AKADEMIA GÓRNICZO-HUTNICZA

im. Stanisława Staszica w Krakowie

WYDZIAŁ INŻYNIERII MECHANICZNEJ I ROBOTYKI

Praca dyplomowa magisterska

Andrzej Szewczyk
Imię i nazwisko

Inżynieria Mechatroniczna
Kierunek studiów

Narzędzie wspierające proces ciągłej integracji poprzez dobór optymalnego zestawu testów.
Temat pracy dyplomowej

Continuous integration tool that supports the process by an optimal test suite selection.
Subject of the thesis

dr inż. Lucjan Miękina	…………………..
Promotor pracy	*Ocena*

Kraków, rok 20...../20.....

Kraków, ……………..

Imię i nazwisko:	Andrzej Szewczyk
Nr albumu:	270039
Kierunek studiów:	Inżynieria Mechatroniczna
Specjalność:	Projektowanie Mechatroniczne

OŚWIADCZENIE

Uprzedzony o odpowiedzialności karnej na podstawie art. 115 ust 1 i 2 ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (tj. Dz.U.z 2006 r. Nr 90, poz. 631 z późn.zm.) : „Kto przywłaszcza sobie autorstwo albo wprowadza w błąd co do autorstwa całości lub części cudzego utworu albo artystycznego wykonania, podlega grzywnie, karze ograniczenia wolności albo pozbawienia wolności do lat 3. Tej samej karze podlega, kto rozpowszechnia bez podania nazwiska lub pseudonimu twórcy cudzy utwór w wersji oryginalnej albo w postaci opracowania, artystyczne wykonanie albo publicznie zniekształca taki utwór, artystyczne wykonanie, fonogram, wideogram lub nadanie”, a także uprzedzony o odpowiedzialności dyscyplinarnej na podstawie art. 211 ust.1 ustawy z dnia 27 lip[ca 2005 r. Prawo o szkolnictwie wyższym (tj. Dz.U. z 2012 r. poz. 572, z późn.zm.) „Za naruszenie przepisów obowiązujących w uczelni oraz za czyny uchybiające godności student ponosi odpowiedzialność dyscyplinarną przed komisją dyscyplinarną albo przed sądem koleżeńskim samorządu studenckiego, zwanym dalej „sądem koleżeńskim”, oświadczam, że niniejszą pracę dyplomową wykonałem osobiście i samodzielnie i że nie korzystałem ze źródeł innych niż wymienione w pracy”.

....

podpis dyplomanta

Kraków, ……………..

Imię i nazwisko:	Andrzej Szewczyk
Nr albumu:	270039
Kierunek studiów:	Inżynieria Mechatroniczna
Specjalność:	Projektowanie Mechatroniczne

OŚWIADCZENIE

Świadomy/a odpowiedzialności karnej za poświadczanie nieprawdy oświadczam, że niniejszą magisterską pracę dyplomową wykonałem/łam osobiście i samodzielnie oraz nie korzystałem/łam ze źródeł innych niż wymienione w pracy.

Jednocześnie oświadczam, że dokumentacja praca nie narusza praw autorskich w rozumieniu ustawy z dnia 4 lutego 1994 roku o prawie autorskim i prawach pokrewnych (Dz. U. z 2006 r. Nr 90 poz. 631 z późniejszymi zmianami) oraz dóbr osobistych chronionych prawem cywilnym. Nie zawiera ona również danych i informacji, które uzyskałem/łam w sposób niedozwolony. Wersja dokumentacji dołączona przeze mnie na nośniku elektronicznym jest w pełni zgodna z wydrukiem przedstawionym do recenzji.

Zaświadczam także, że niniejsza magisterska praca dyplomowa nie była wcześniej podstawą żadnej innej urzędowej procedury związanej z nadawaniem dyplomów wyższej uczelni lub tytułów zawodowych.

………………………………..
podpis dyplomanta

Kraków, ……………..

Imię i nazwisko:	Andrzej Szewczyk
Adres korespondencyjny:	42-100 Kłobuck, ul. Kościuszki 37
Temat pracy dyplomowej magisterskiej:	Narzędzie wspierające proces ciągłej integracji poprzez dobór optymalnego zestawu testów.
Subject of the Master’s thesis:	Continuous integration tool that supports the process by an optimal test suite selection.
Rok ukończenia:	2018
Nr albumu:	270039
Kierunek studiów:	Inżynieria Mechatroniczna
Profil dyplomowania:	Projektowanie Mechatroniczne

OŚWIADCZENIE

Kraków, ...… ……………………………..
data *podpis dyplomanta*

[i] Na podstawie Ustawy z dnia 27 lipca 2005 r. Prawo o szkolnictwie wyższym (Dz.U. 2005 nr 164 poz. 1365) Art. 239. oraz Ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (Dz.U. z 2000 r. Nr 80, poz. 904, z późn. zm.) Art. 15a. "Uczelni w rozumieniu przepisów o szkolnictwie wyższym przysługuje pierwszeństwo w opublikowaniu pracy dyplomowej studenta. Jeżeli uczelnia nie opublikowała pracy dyplomowej w ciągu 6 miesięcy od jej obrony, student, który ją przygotował, może ją opublikować, chyba że praca dyplomowa jest częścią utworu zbiorowego."

Kraków, ……………..

AKADEMIA GÓRNICZO-HUTNICZA
WYDZIAŁ INŻYNIERII MECHANICZNEJ I ROBOTYKI

TEMATYKA MAGISTERSKIEJ PRACY DYPLOMOWEJ
dla studenta II roku studiów stacjonarnych

....
imię i nazwisko studenta

TEMAT PRACY DYPLOMOWEJ MAGISTERSKIEJ:

Narzędzie wspierające proces ciągłej integracji poprzez dobór optymalnego zestawu testów.

SUBJECT OF THE MASTER'S THESIS:

Continuous integration tool that supports the process by an optimal test suite selection.

Promotor pracy:	dr inż. Lucjan Miękina	
Recenzent pracy:	dr hab. inż. Mariusz Giergiel, prof. AGH	.... *Podpis dziekana:*

PLAN PRACY DYPLOMOWEJ
1. Omówienie tematu pracy i sposobu realizacji z promotorem.
2. Zebranie i opracowanie literatury dotyczącej tematu pracy.
3. Zebranie i opracowanie wyników badań.
4. Analiza wyników badań, ich omówienie i zatwierdzenie przez promotora.
5. Opracowanie redakcyjne.

Kraków, ...… ……………………………..
data *podpis dyplomanta*

TERMIN ODDANIA DO DZIEKANATU: .... 20........ r.

....
podpis promotora

Kraków, ……………..

Akademia Górniczo-Hutnicza im. Stanisława Staszica

Wydział Inżynierii Mechanicznej i Robotyki

Kierunek: Inżynieria Mechatroniczna

Profil dyplomowania: Projektowanie Mechatroniczne

Andrzej Szewczyk

Magisterska praca dyplomowa

Narzędzie wspierające proces ciągłej integracji poprzez dobór optymalnego zestawu testów.

Opiekun: dr inż. Lucjan Miękina

STRESZCZENIE

W ostatnich latach miała miejsce gigantyczna digitalizacja życia codziennego każdego z nas. Zastanówmy się ile jest przedmiotów w zasięgu naszego wzroku, w których znajduje się mikroprocesor realizujący pewien ciąg instrukcji na podstawie kodu. Osoby niezwiązane w branżą IT zapewne dojdą do następującego wniosku: skoro urządzenia lub aplikacje, które są na co dzień używane, wymagają dobrze napisanego kodu w celu spełnienia oczekiwań użytkownika, to zapewne potrzeba skończonej liczby programistów, którzy ten kod napiszą. Niestety, nic bardziej mylnego. Aby firma tworząca oprogramowanie mogła odnieść komercyjny sukces, musi ona, oprócz wykwalifikowanych pracowników, posiadać również skuteczny proces, który zapewni, że oprogramowanie dostarczone końcowemu klientowi jest możliwie wysokiej jakości.

Celem niniejszej pracy magisterskiej jest zaprezentowanie narzędzia, które prowadzi nie tylko do zwiększenia jakości końcowego kodu, ale również zmniejsza ryzyko niepowodzenia projektu oraz przyspiesza czas jego realizacji. Systemy ciągłej integracji oparte na regularnym dostarczaniu, budowaniu i automatycznym testowaniu zintegrowanych wersji kodu stały się normą w nowoczesnych firmach programistycznych. Rozwiązanie opisane w tej pracy może dodatkowo usprawnić proces ciągłej integracji, ponieważ natychmiastowe wykonanie optymalnego zestawu testów regresyjnych dobranych na podstawie zmian w kodzie, zapewnia wartościową informację na temat jakości testowanej wersji oprogramowania. Zaprezentowane narzędzie mogłoby być skutecznie wykorzystane w fazie rozwoju dużych systemów informatycznych oraz w międzynarodowych, dużych zespołach programistycznych.

Kraków, the……………..

Akademia Górniczo-Hutnicza im. Stanisława Staszica

Faculty of Mechanical Engineering and Robotics

Field of Study: Mechatronics engineering

Specializations: Mechatronics design

Andrzej Szewczyk

Master's thesis

Continuous integration tool that supports the process by an optimal test suite selection.

Supervisor: dr inż. Lucjan Miękina

SUMMARY

In the recent years, reality of everyday life has undergone the real digital transformation. Let's think about how many devices in our sight have a CPU[1], which is executing a set of instructions based on the code. Most people that are not familiar with the IT[2] industry will draw the following conclusion: since all devices or applications that are used every day require a well-written code in order to satisfy end-user needs, there is a need to have a finite number of programmers to write this code. Still, there is nothing more wrong. If a company that delivers software wants to achieve the success, besides good programmers, this company must have a process, which is aimed at ensuring that the software delivered to an end-customer has the best possible quality.

The purpose of this Master's thesis is to propose a tool that leads not only to quality improvement of the final code, but also mitigates the risks in a project and accelerates time to market. Continuous integration systems that have been built based on regular delivery of automatically pre-tested integrated code builds are a common practice in modern IT companies. The solution described in this thesis could further enhance the continuous integration process, because immediate execution of an optimal regression test suite, which is selected based on changes in the latest code commit, provides valuable information on quality of the software version under test. The proposed tool may be effectively used in the development phase of large-scale IT systems or by global software development teams.

[1] CPU – Central Processing Unit

[2] IT – Information Technology

Table of Contents:

1. Introduction

This chapter is intended to introduce the reader to the content of the thesis. It comprises four sections. Section 1.1. defines the main goal of the thesis and illustrates the steps taken to achieve it. Section 1.2. briefly presents the role of continuous integration in modern software development methodologies. Section 1.3. discusses how my interest in the CI[3] process has aroused, including a mention of two years of my professional experience with the software development process in the multiple international teams. Section 1.4. examines sources of information which refer to the subject of the thesis.

1.1. Goal and plan of the thesis

The main goal of the thesis is to propose a continuous integration tool, which besides the general advantages of adopting the CI principles, places additional emphasis on measuring system-wide impact of local changes. The proposed continuous integration tool defines a coherent sequence of steps containing the following items: analysis of local changes between code versions, selecting tailor-made test suite for the changes, execution of the selected test cases, execution of the integration sanity check, providing immediate feedback on code quality.

The steps taken to accomplish the above goal are discussed in details in the further part of the work that is organized as follows.

Chapter 2. provides an overview of the most common software development models. Not only is there a detailed description of each model included, there are also the pros and cons of the three main approaches presented.

The next part of the thesis focuses on the role of continuous integration process and testing activities in the software development life cycle. There are best practices adequately highlighted and general definitions of test levels and test types included.

Chapter 4. describes software tools that are used in the thesis. There is a brief description of their capabilities presented. Furthermore, there is an additional emphasis put on the features that have been used for the purpose of achieving the main goal of the thesis. This chapter contains a wide range of tools including programming languages, various frameworks or software management systems.

The requirements of the unit under test are specified in chapter 5. The unit under test is an application for which changes in the code between commits are compared. The

[3] CI – Continuous Integration

way the application works is presented in the block diagrams for clarity and ease of explanation.

In chapter 6. the testing activities mentioned in chapter 4. are illustrated based on the testing process artifacts that have been implemented in the thesis. Test cases for each test level are described with a detailed explanation of the testing frameworks features that are used in order to improve testing capabilities Similarly to the previous section, chapter 6. includes a block diagram serving as an example of the process of test cases definition.

Chapter 7. gives an overview of the algorithm intended to select an optimal test suite. There are inputs and outputs of the proposed algorithm presented. This chapter contains also an example of an optimal test suite, which is the product of the algorithm execution for a given change made to the code.

In chapter 8. there is a detailed explanation of the CI tool design provided. There are also several examples of both successful and unsuccessful execution of the CI pipeline included. Furthermore, in this section, there is a description of the essential Jenkins features, which may be extremely useful in the process of building and testing software.

Chapter 9. finishes the thesis by providing a summary of the presented work. In the final part, there are also possibilities for further improvements presented.

1.2. Why continuous integration is important

Agile methodologies are currently one of the most well-known and frequently used software development life cycle models. It is difficult to find a job offer for a programmer that does not contain any of the following words: Agile, Scrum, pair programming, CI, CD[4] and much more associated with the agile methodologies.

In a nutshell, there are two main goals of CI: to automatically generate a software build and to provide developers with immediate feedback about quality of the recent code build. This approach to the software development process claims to be more human friendly than traditional development methods.

At this place, I would like to make a reference to the Manifesto for Agile Software Development. This manifesto was proclaimed by the seventeen signatories during a meeting in Snowbird, Utah between 11th and 13th of February 2001. This meeting is deemed to be the beginning of a revolution in software development. Since then, the eXtreme Programming (XP) enthusiasts are setting the pace for development of state-of-

[4] CD – Continuous Delivery

the-art digital technologies. The above-mentioned agile methodologies are the most common ideas behind the XP approach.

The following declaration has been made at this meeting:

"We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- Individuals and interactions over processes and tools,
- Working software over comprehensive documentation,
- Customer collaboration over contract negotiation,
- Responding to change over following a plan.

That is, while there is value in the items on the right, we value the items on the left more. [1]"

How does CI relate to the above declaration? Since collaboration with a customer is very important, the simplest way to collaborate with customers is to present the outcome of our work on a regular basis. The point is that, in order to show the customer a product, the code that is running inside this product has to work as the customer expects. The question arises, is it possible to deliver new functionalities, which were implemented in the code written a few hours ago, to the customer with a high degree of certainty in terms of its good quality. Of course, it is possible to achieve this. The only thing that needs to be done before handing a product off to a customer is to find issues in the code and fix them long before the customer will find these issues.

In the preceding subparagraph, is has been said that a product can be delivered to a customer as soon as an evidence on quality of the working code is provided. These are just some of the requirements for a software development process that allows to release new functionalities within the tight deadlines:

- The build process shall be done automatically following the strictly defined procedure,
- The test environment shall emulate the normal operating condition for a product as close as possible,
- Test scripts shall be written before the code is written and those tests should be automated,
- Integration and testing of the code shall happen several times a day,
- The simplest solutions shall be implemented to meet today's problems,
- Generation of business stories should be used to define the functionality,

- Shared code ownerships amongst the developers shall be promoted,
- An on-site customer for continual feedback and to define functional acceptance testing shall play a crucial role in the project. Those acceptance tests may be automated and built-in in the CI pipeline while being supervised by the customer.

With the above requirements, there are numerous iterations of software builds, each requiring testing. Once a developer starts writing every test case, it has to be ensured that it can be automated. Every time a change is introduced in the code, it shall be tested at the component level and then integrated with the existing code, which is then fully integration-tested using the full set of test cases. This gives continuous integration, by which it is meant that changes are incorporated continuously into the software build [2].

1.3. Motivation to take Continuous Integration topic up

This paragraph is a quick summary of my engineering career path. I started my career here in AGH University of Science and Technology as a mechatronics engineering student, then in parallel to the studies, I have been an intern in Industrial Turbomachinery Systems department in Woodward Poland sp. z o.o. for 14 months, finally I ended up as an embedded software verification engineer in Aircraft Turbine Systems department in the same company. Oddly enough, I have never got a chance to work on a project that has been developed based on one of the agile methodologies. Due to the strict certification process, the projects I was involved in are mostly developed using the V-model software development life cycle. Each level in the V-model was split into small activities that were realized using the customized waterfall models.

The interesting thing is that, while I was working on particular projects, I did not consider some project activities as problems that are mainly caused by some weaknesses of the V-model or waterfall model itself. Currently, I am aware that most of these problems can be readily addressed by introducing agile methodologies into the project activities. The breakthrough that has led to changes in my outlook on the software development process was participation in the ISTQB[5] Foundation Level course. I passed the certification and gained the practical knowledge of the fundamental concepts of software testing including people in roles such as testers, test analysts, test engineers, test consultants, test managers, user acceptance testers and software developers. The

[5] ISTQB - International Software Testing Qualifications Board

scope of ISTQB Foundation Level covers all software development practices including Waterfall, Agile, DevOps and Continuous Delivery [3].

There is huge potential for taking advantage of agile methodologies, particularly the CI process, to enhance the process of embedded software development for aerospace applications. Thus, I have decided to set up and validate my own continuous integration tool. I did my best to provide a solution that can solve many engineering difficulties I have encountered in the past. Furthermore, while I was working on the CI tool that is subject to this thesis, I was facing numerous challenges. Solving many of these problems was very demanding and time consuming. Thus, it should not be a surprise to anyone that, to complete a software project on time and on budget, there is a need to have a CI expert onboard, who is responsible for development and maintenance of the whole CI infrastructure dedicated to the project.

1.4. Review of technological know-how for the CI process

In the recent years, continuous integration has become a very popular cure for the common dysfunction of many software teams. Thus, it is obvious that the CI subject matter experts want to make use of their expertise for the commercial purposes. As a result, there is a lot of training courses teaching how to incorporate ideas of the CI process to an organization. These courses are often tailor-made and suit the needs of a particular organization. Participation is such course itself can be very expensive.

On the other hand, there are some people who are still willing to share their knowledge for free. While I was working on this thesis, I found much useful information about the different aspects of continuous integration. There are a lot of articles on the software development process in the internet, which in my opinion are the most valuable source of information. The authors of these articles have been beginning from the same starting point as I have. Similarly, they had to develop a continuous integration tool from scratch. Most of the stories presented by them start with some simple CI solutions that got the management approval. Over the course of time, the more time was spent on the CI tools development, the more buy-in was received. Finally, these guys have led to fundamental changes in the entire process of software development. There are many examples that I am referring to, be the most flourishing one, which has significantly increased my interest in CI can be found at the following link: https://bulldogjob.pl [4].

In one of the preceding subparagraphs, the costs of implementation of agile methodologies in an organization was considered. It has to be kept in mind that these costs are certainly much higher. In many cases, software teams that are adopting agile are bound undergo a deep restructuring to start thinking in an agile way. Based on the below examples of successful implementation of the CI practices in the world's largest high-tech companies, undoubtedly it is worth making financial investments to use the agile mindset on a daily based.

Here are 4 trail blazing companies that exemplify the possibilities of DevOps[6] [5]:

- Amazon – on average, engineers are deploying code every 11.7 seconds.
- Netflix – known for its commitment to automation and open source.
- Facebook – known for its accelerated development lifecycle that meets consumers' expectations of software by bi-weekly app updates, effectively served notice and constant, rapid refreshes for mobile apps.
- IBM - It is estimated that 70-80% of teams are pure agile teams.

[6] DevOps - Development and Operations. Focuses on culture that aims at raising awareness of potential benefits for an organization, which are the result of continuous deployment (consists of continuous delivery, continuous integration, and continuous testing) with the lean management principles.

2. Software development process

Software development process, also known as a software development life cycle, is the process of dividing software development work into phases. The phases contain manageable chunks of tasks that can be assigned to individuals responsible for a certain activity within the software development life cycle. Depending on available resources, type of a software project or a product and last but not least, software development model, different activities can be can be defined. The most common are:

- Requirements definition,
- Software architecture design,
- Code development,
- Software build process definition,
- Testing,
- Debugging and bug fixing,
- Deployment,
- Configuration management,
- Maintenance.

In the next part of this paragraph, the above activities are going to be presented with an emphasis on their roles and time frames in software development models. Each of the activities may either have its own separate phase or be aggregated with another activities into one larger phase.

2.1. Waterfall model

The waterfall model is one of the earliest models that have been defined. This model has a natural timeline and all tasks are executed in a sequence. At the top of the waterfall there is a study of user requirements, then system requirement are defined. The waterfall flows down through the various project tasks. Once the design is ready, development starts, which in turn flows into build. In the final step testing activities are conducted.

The apparent risk that arises in the waterfall approach is a likely event of finding bugs in the testing phase close to the end of the project life cycle. In general, with this model it is difficult to get feedback passed to any preceding phase in the waterfall. There are also additional difficulties if there is a need to carry out numerous iterations for a particular phase.

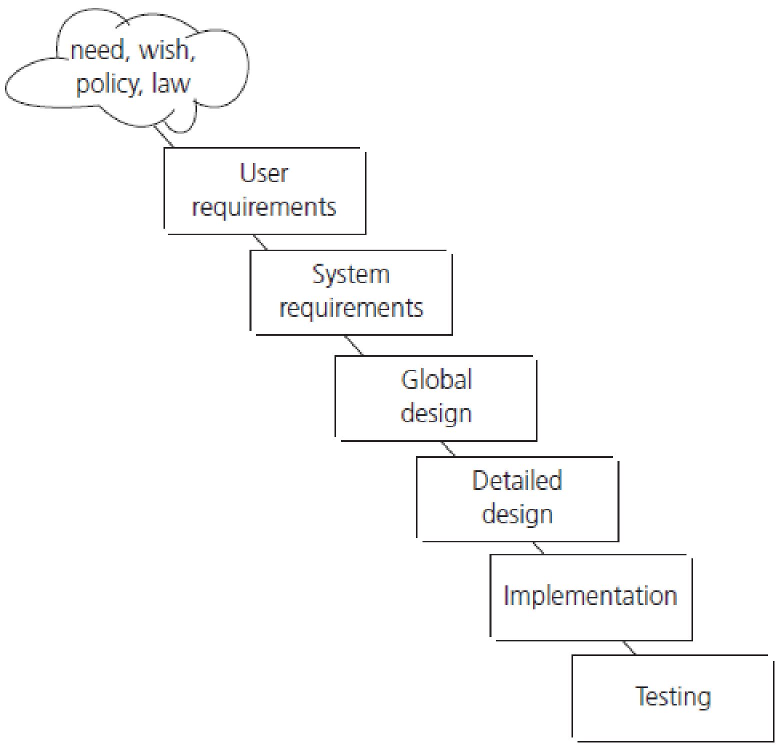


Fig. 2.1. Waterfall model [2].

Nonetheless, in some cases the pros of the waterfall model outweigh the cons. This model is efficient for small software projects that are carried out by a few engineers. The other possibility of using the waterfall approach are those projects that have well-defined requirement at the very beginning and these requirements are highly unlikely to be changed at any stage of the project.

2.2. V - model

The V-model has been designed to address some of the problems that are being experienced using the waterfall approach. Bugs are found too late in the life cycle, as testing is not involved in the early stages of the project. The other problem associated with late testing is added lead time complicated to estimate. The fundamental principle provided by the V-model is to begin testing as early as possible in the life cycle. The model is also aimed at showing that testing is not only an execution-based activity and it defines a variety of testing activities that need to be performed before the end of the code development phase. Ideally, these activities should be carried out in parallel with development activities.

The V-model illustrates how testing activities, which are validation[7] and verification[8], can be integrated into each phase of the life cycle. Each phase of the V-model has its own test level comprises of a group of testing activities that are organized and managed together. Validation testing takes place especially during the early stages (e.g. reviewing the user requirements), and late in the life cycle (e.g. during user acceptance testing). Verification tasks exist mostly in the intermediate stages of the V-model. However, in practice, a V-model may have more, fewer or different levels of development and testing.

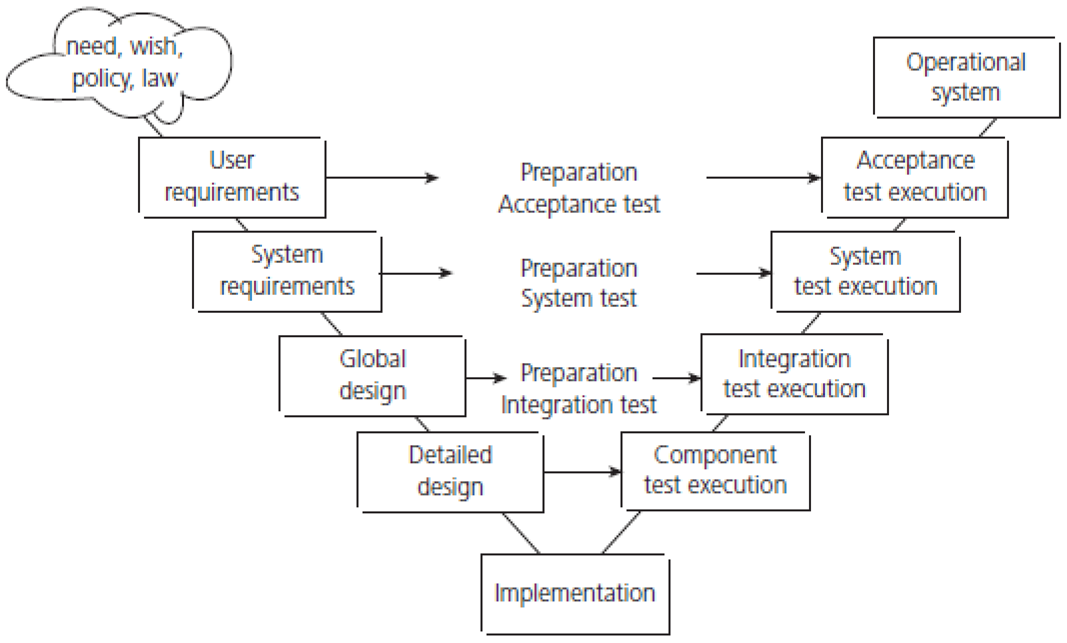


Fig. 2.2. V - model [2].

The V-model approach is a highly disciplined software development model. It promotes precise design, careful development, and comprehensive documentation necessary to build stable software products. Hence, this approach is widely used in applications that require high degree of reliability, which is standard in medical or aerospace industry.

2.3. Iterative life cycles – agile methodologies

The main purpose of iterative or incremental life cycles is cycling through a number of smaller, effective and result-driven life cycles phases for the same project instead of one large development activity.

[7] Validation - aims at giving an evidence that the requirements for a specific use or application have been fulfilled.
[8] Verification - aims at giving an evidence that the specific requirements have been fulfilled.

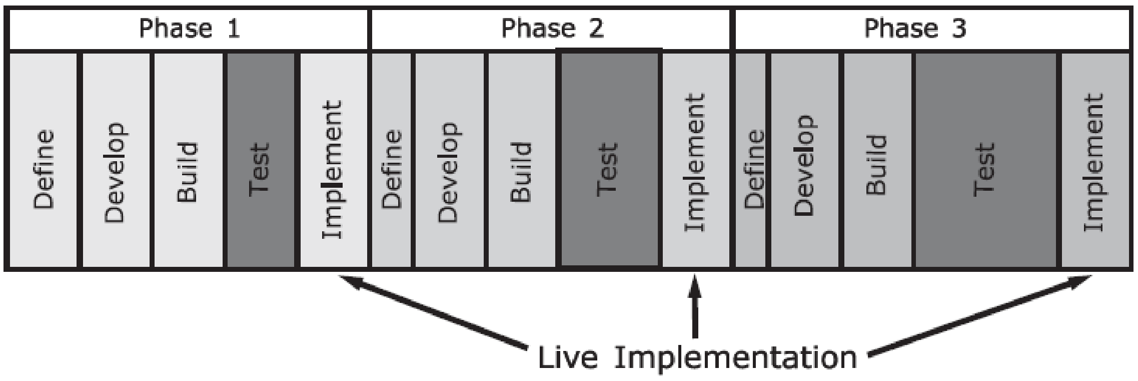


Fig. 2.3. Iterative development model [2].

A common feature of the iterative approach is that the delivery is divided into increments or builds. The main advantage of this approach is that iterative development can give early market presence with critical functionality. As a result, the customer can provide the development team with feedback on the product. If the early versions are not satisfactory, the user or system requirements can be redefined with little impact on the project time line. Besides the business value and fitness-for-use of the product that are continuously improved in the subsequent deliveries to the customer, each increment adds a portion of functionality in the overall project requirements.

From the testing perspective, subsequent increments require testing for the new functionality, testing of the existing functionality and integration testing of new and existing code. Agile enthusiasts tend to say that working software is the primary measure of progress. Thus, regression testing plays a crucial role in all iterations after the first one. In some versions of the incremental approach, each phase follows a ‘mini V-model’ with its own design, coding and testing activities.

There are couple examples of incremental development models:

- Rapid Application Development (RAD),
- eXtreme Programming (XP) ,
- Agile methodologies.

For the characteristic of the above models with an emphasis on agile methodologies see section 1.2.

The iterative approach provides greater flexibility throughout the development process, because it allows the software to quickly respond to changes in market requirements or business environment.

3. CI and testing throughout software life cycle

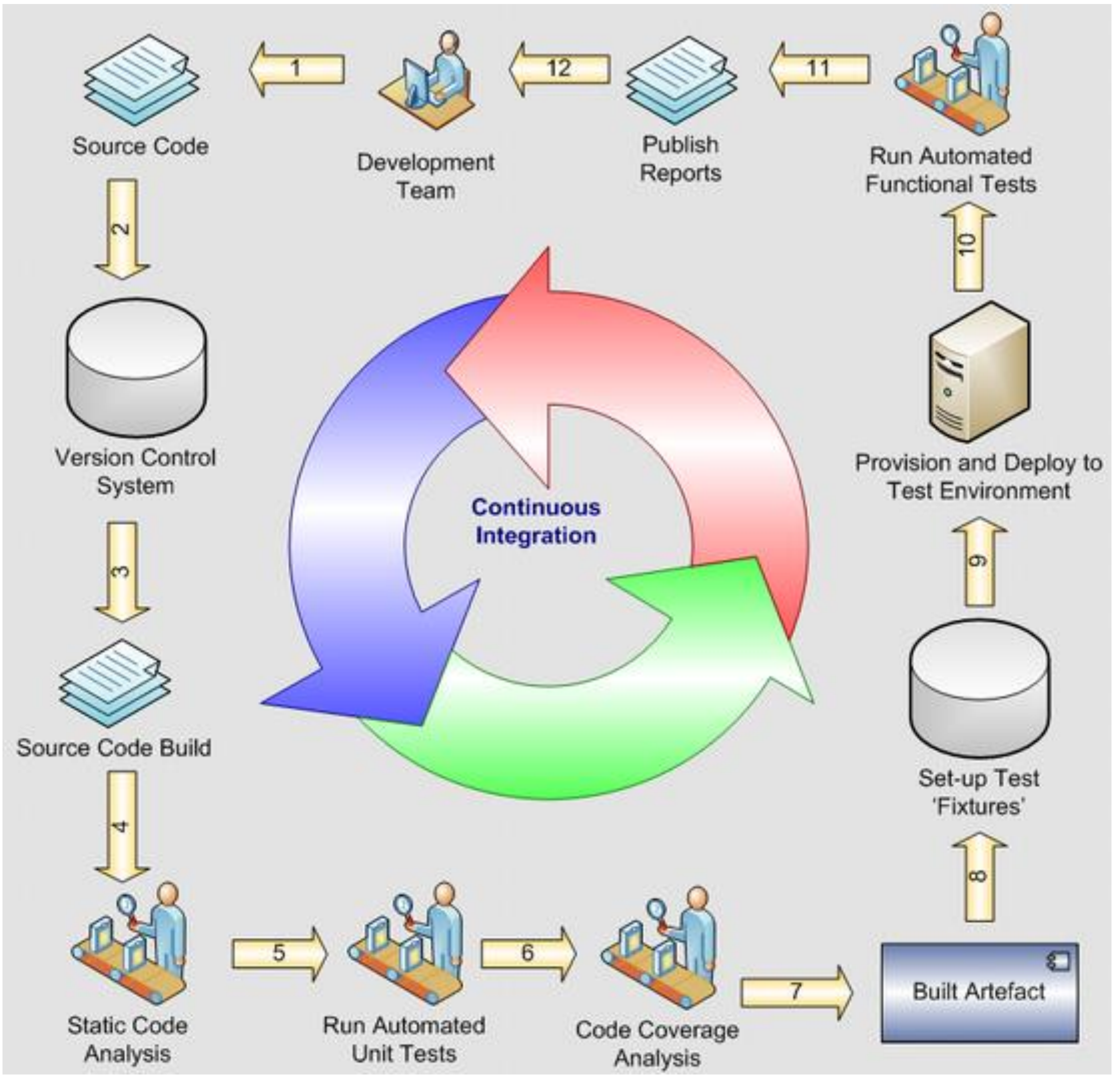


Fig. 3.1. Phases in a continuous integration pipeline [6].

The continuous integration pipeline presented above illustrates a wide variety of activities, such as checking out and building new versions of code (phases 1. to 4.), running tests (phases 5. to 11.), and providing the development team with feedback on quality of the latest code version (phase 12.). If all testing activities have been executed successfully and no bugs have been found, the software build under test can enter the software deployment phase.

The comparison of the phases in a continuous integration pipelines against the phases of the software development models described in chapter 2. leads to a conclusion that CI activities are present in every phase of the software life cycle, except for the requirements definition phase. However, in parallel to user and system requirements definition, the requirements for a CI server may be defined.

3.1. Common principles and best practices of CI process

The objectives of CI have been already mentioned in paragraph 1.2. They are: build and test software automatically and provide developers with immediate feedback about quality of the recent code build. To achieve these objectives, continuous integration relies on the following principles [7]:

- Maintain a code repository.
- Automate the build,
- Make the build self-testing,
- Every commit should be built on an integration machine,
- Keep the build fast,
- Test in a clone of the production environment,
- Make it easy for anyone to get the latest executable version,
- Everyone can see the results of the latest build,
- Automate deployment.

The starting point when implementing continuous integration is an assumption that a single command should have the capability of building the system. Once system is built, all subsequent phases of a CI pipeline should run automatically one after another. To put the principles of Continuous Integration into practice, the following requirements for the CI server should be fulfilled [7]:

- The CI server monitors the repository and checks out changes when they occur,
- The CI server builds the system and runs unit and integration tests. The unit and integration test suites shall be extensive,
- The CI server releases deployable artefacts for testing,
- The CI server assigns a build label to the version of the code it just built to make it easy to get the latest deliverables or reproduce the software build in the future,
- If the build or tests fail, the CI server alerts the team,
- The team fixes the issue at the earliest opportunity,
- Continue to continually integrate and test throughout the project,
- The CI pipeline shall be fast and easily maintainable.

If the CI server meets the above criteria, it is intended to produce benefits such as: reduction of overhead across the development and deployment process, reduction in the time and effort for integrations of different code changes what enables the development team to focus on adding features, improvement of collaboration between team members so recent code is always shared or earlier detection and prevention of defects.

3.2. Testing activities in software development process

In general, testing is the process that comprises all life cycle activities, both static and dynamic, associated with planning, preparation and evaluation of software products and related work products. The objective of testing is to ascertain that a software product satisfies specified requirements (verification) as well as demonstrate that the software product is fit for purpose (validation).

Regardless of the software development model, there are several characteristics of good testing. From the high level point of view, these principles can be defined as follows:

- Each test level[9] shall have test objectives specific to that level,
- For every development activity, there shall be a corresponding testing activity,
- Testers shall be involved in a project early in the development life cycle (e.g. as soon as drafts of the documents are available),
- The analysis and design of tests for a specific test level shall begin during the corresponding development activity.

3.3. Test levels

The differentiation between particular tests level has been introduced to clearly separate various test activities and assign these activities to a specific phase of a project. The process of defining the various test levels is aimed at identifying missing areas in the testing approach and preventing overlapping and repetition. However, in some cases an intended overlap may be put in place to address and mitigate specific risks. Four test levels are distinguished:

- Component (unit) testing,
- Integration testing,
- System testing,
- Acceptance testing.

To assign the above test level to a specific project phase, refer to the definition of the V-model (see section 2.2., figure 2.2.).

[9] Test level- a group of testing activities that are organized and manager together. A test level is linked to the responsibilities in a project [2].

Based on the scope of this thesis and the test levels that matter most in the Continuous Integration process, unit testing and integration testing are going to be set out in detail.

Unit testing searches for defects and verifies functionality of software modules that are separately testable (e.g. programs, objects, classes). To isolate a component under test from the rest of system dependencies, stubs and drivers are used to replace the missing software and create the interface between the software components. A stub is a skeletal or special-purpose implementation of a software component. Stubs are called from the software components and they replace the behavior of a component they are simulating. Drivers, unlike stubs, call a component to be tested. A driver is a software component or test tool that replaces a component that takes care of control or/and the calling of a component or system.

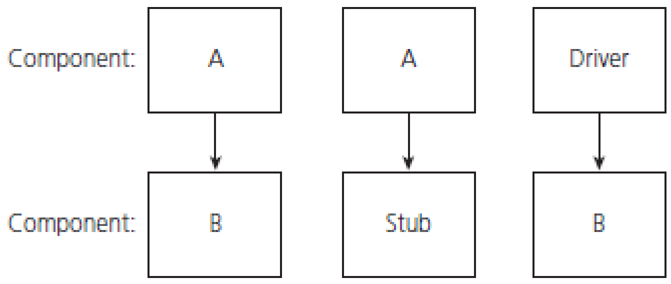


Fig. 3.2. Stubs and drivers [2].

Component testing may include testing functional characteristics, non-functional requirements such as performance, robustness testing or resource-behavior, as well as structural testing (e.g. statement or decision coverage). Typically, unit tests are written in a unit test framework or debugging tool that can access the code being tested by the programmer who wrote the code.

Agile methodologies support test-driven development which is an approach to component testing to prepare and automate test cases before implementing the code. Test-driven development is highly iterative and it defines cycles of writing test cases, then implementing small pieces of code, building and integrating the code. Components tests are executed unit they pass.

The second test level that is of interest to the thesis is integration testing. It is performed to expose defects in the interactions between integrated components or systems. There is more than one level of integration testing, depending on test objects and varying sizes of systems containing integrated components. There might be component and system integration testing distinguished. Component integration testing is intended to test the interactions between software components and is carried out after unit testing. System integration testing is designed to test the interactions between different systems and may be carried out after system testing.

Integration testing does not lead to isolation of failure to specific lines in the code. Thus, to find cause of failure and identify the buggy component, the integration sequence and the number of integration steps shall be pre-defined. At each stage of integration, testers focus solely on integration itself. It means that they are not interested in testing functionality of the components, but they are testing interactions between the components to give evidence if an integrated system meets specified requirements.

Integration testing should investigate both functional and non-functional requirements of the system. Sometimes, testers may also deal with incomplete or undocumented features. Typically, functional testing is carried out based on specification-based (black-box) techniques in a controlled test environment. Due to the fact that integration testing requires a dedicated testware and test data, integration tests are written and executed by the independent testing team responsible for development of the whole test infrastructure. The test environment shall be as close as possible to the final target or production environment.

3.4. Test types

A test type is a group of testing activities aimed at testing a component or system focused on a specific test objective [2]. The particular test objectives could be: functional testing, non-functional testing, the structure or architecture testing, confirmation or regression testing. Each of the test types may be performed at all test levels.

Functional testing verifies if the requirements specification on what a system or component does is correctly implemented in the code. Specification-based testing is often referred as black-box testing, because the code is executed without a direct access to that code. A test case has as set of inputs that launch the code. Then the test case checks outputs of the code execution. The outputs should have values that are defined in the system or component specification in the case of the software is used under specified conditions that have been set by the inputs.

Testing of software product characteristics, which is called non-functional testing, focuses on measuring how well something is done. A particular requirement is tested to assess the code implementation under a specific scale of measurement (for example the scale of measurement can be time to respond). The characteristics that are commonly verified in non-functional testing are reliability, usability, efficiency, maintainability or portability. These tests are likewise executed in the black-box test environment.

In structural testing unlike black-box testing, a test case has direct access to the code. Testing of software structure or architecture is often referred as white-box testing. This type of testing is most often used in measuring the coverage of a set of structural elements. At component level structural testing may measure code coverage, at component integration level it may be used for checking a calling hierarchy, whereas at system integration level control flow in the code may be verified by a structural test.

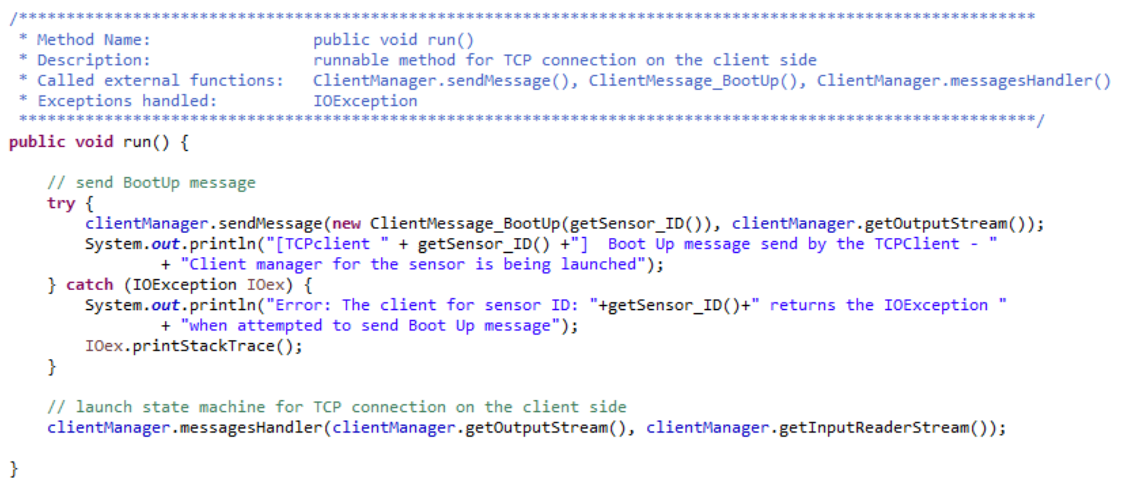
Testing related to changes is either conformation or regression testing. Confirmation testing is re-testing of a component or system in which there has been a defect found in the previous build. The goal of confirmation testing is to verify if a new version of the software fixes the bug. Regression testing, like confirmation testing, involves executing the test cases that have been executed before. However, in this case, a new build is verified with a view to determining if the last version of the software has not introduced or uncovered a different defect elsewhere in the code. Regression tests are executed every single time the software changes, either as a result of bug fixes or new functionality. It is advisable to have a regression test suite at each level of testing.

4. Software toolset

To achieve the goal of the thesis and propose a Continuous Integration tool that supports the process by an optimal test suite selection there was a need to use the various programming languages, multiple testing frameworks, a CI automation server, a version control system, an integrated development environment and a software project management tool.

4.1. Programming languages

There are two programming languages used in the thesis. The unit under test is implemented in Java SE 8. The script that selects an optimal test suite is implemented in Python 2.7.

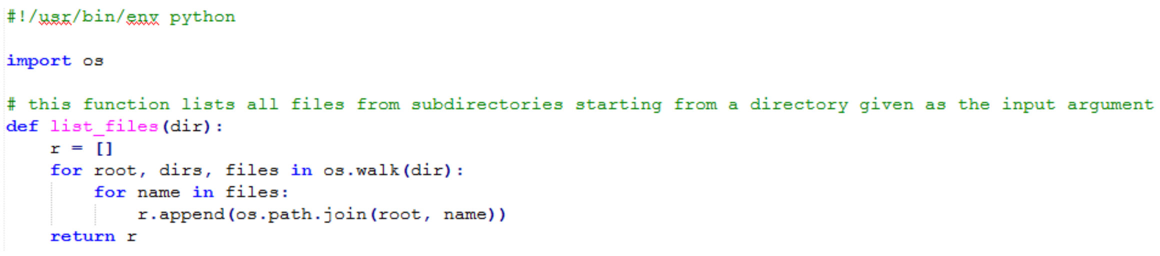


```java
/*********************************************************************************************************
 * Method Name:                public void run()
 * Description:                runnable method for TCP connection on the client side
 * Called external functions:  ClientManager.sendMessage(), ClientMessage_BootUp(), ClientManager.messagesHandler()
 * Exceptions handled:         IOException
 *********************************************************************************************************/
public void run() {

    // send BootUp message
    try {
        clientManager.sendMessage(new ClientMessage_BootUp(getSensor_ID()), clientManager.getOutputStream());
        System.out.println("[TCPclient " + getSensor_ID() +"]  Boot Up message send by the TCPClient - "
                + "Client manager for the sensor is being launched");
    } catch (IOException IOex) {
        System.out.println("Error: The client for sensor ID: "+getSensor_ID()+" returns the IOException "
                + "when attempted to send Boot Up message");
        IOex.printStackTrace();
    }

    // launch state machine for TCP connection on the client side
    clientManager.messagesHandler(clientManager.getOutputStream(), clientManager.getInputReaderStream());

}
```

Fig. 4.1. Java code snippet - a method implementation.

Java Platform, Standard Edition (Java SE) is a computing platform for development and deployment of portable code for desktop and server environments. Java SE defines a set of general-purpose APIs, for example Java APIs for the Java Class Library. It also includes the Java Language Specification and the Java Virtual Machine Specification [8]. The best known implementation of Java SE is Java Development Kit (JDK™). The JDK™ used in the thesis as a development environment for building the unit under test using the Java programming language is JDK 8u161.

A package in Java is a group of related types that provide access protection and name space management. The types refer to classes, interfaces, enumerations, and annotation types. The unit under test includes several packages. Naturally, it calls elementary classes from java.lang or classes for reading and writing (input and output) from java.io. However, the essential packages that enable the application to provide its basic functionality such as web-based networking or multiple threads to run tasks concurrently are java.net and java.util.concurrent.



```python
#!/usr/bin/env python

import os

# this function lists all files from subdirectories starting from a directory given as the input argument
def list_files(dir):
    r = []
    for root, dirs, files in os.walk(dir):
        for name in files:
            r.append(os.path.join(root, name))
    return r
```

Fig. 4.2. Python code snippet - a method definition.

Python, unlike Java that is neither interpreted nor compiled, is a pure interpreted programming language. What it means is that instructions are executed directly and freely, without previously compiling a program into machine-language instructions. As a result, particularly in general-purpose programming, additional flexibility over compiled programming languages may be achieved.

One of the main reasons for the choice of Python is that it allows to implement a wider range of functionalities in fewer lines in comparison with other programming languages. The additional advantages of Python are speed, efficiency, widespread library support and programmer-friendly data structures. The most challenging requirement for the optimal test suite selector is the desired functionality to crunch a great deal of data varying in size in the limited amount of time. Python features that are fit for this purpose are extremely flexible lists and the NumPy library. Furthermore, Python supports interaction with git repositories by means of the GitPython library thus all requirements could have been implemented in a single script.

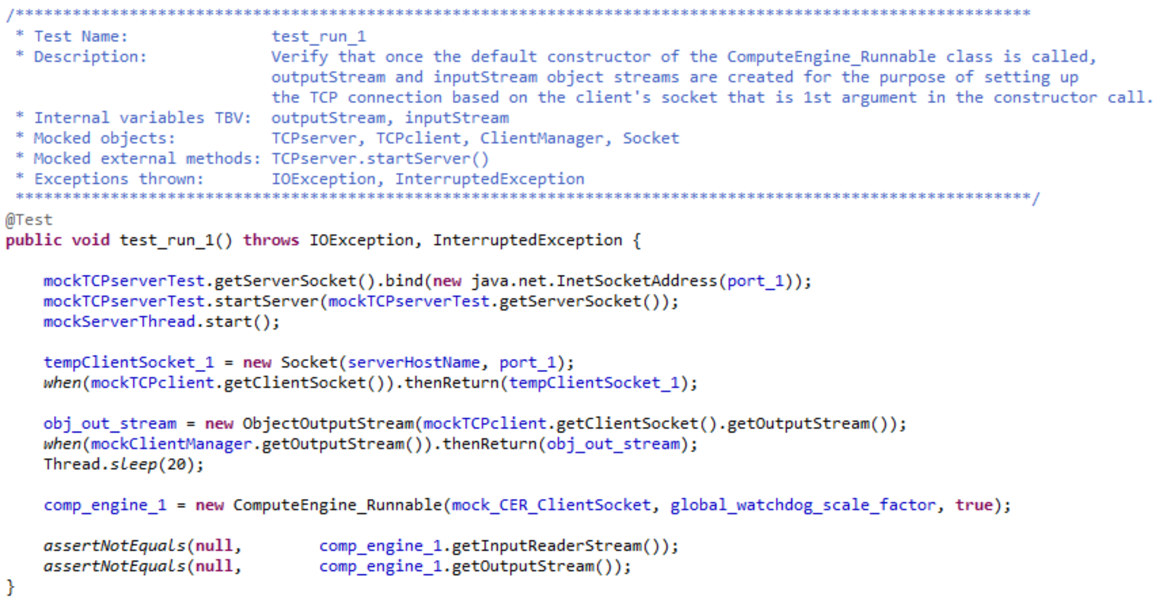
4.2. Testing Frameworks

Given the scope of the thesis, there is a need to implement unit and integration tests for the application under test. The tests might have been developed in dedicated testing frameworks, separate for each test level. However, multiple testing frameworks would result in greater complexity of the optimal test suite selector. Therefore, a comprehensive testing framework has been chosen that supports both unit and integration tests.

JUnit is an open source framework, which can be used for writing and running tests. Basically, a JUnit test is written in Java programming language. However, the JUnit test framework extends Java by providing some interfaces for the testing purposes. The most commonly used features for quick and easy generation of test cases and test data are:

- Annotations that can annotate variables, parameters, packages, methods to enhance source code readability and structure. The following annotation can be found in the unit tests: Before, After, Test, Mock, Spy.
- Assertions that provide methods useful in determining pass or fail status of a test case. There are various types of assertions like Boolean, null, identical etc.
- Test Runners and Test Suites for running tests,
- JUnit Expected Exception Test that traces the exception and also checks whether the code is throwing expected exception or not.

JUnit is a perfect choice for both unit and integration tests, because it is also a regression testing framework that can be easily integrated with an IDE and a software project management tool. Nonetheless, JUnit was originally meant to write and run unit tests, hence some features for integration testing are limited. Still, the existing features are sufficient for the purpose of achieving the goal of the thesis.



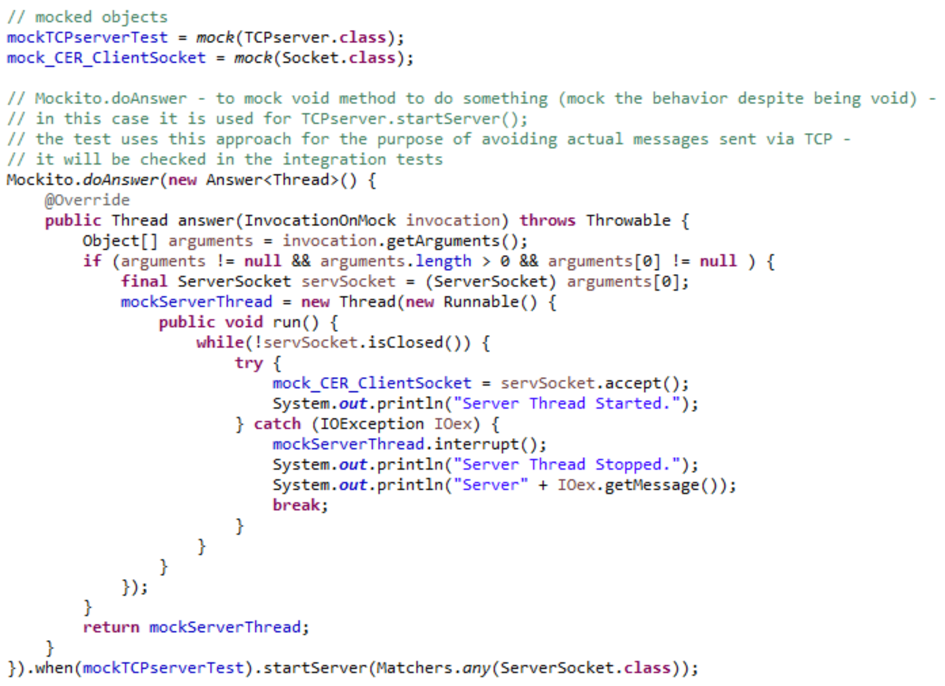
```
/************************************************************************************************************
 * Test Name:              test_run_1
 * Description:            Verify that once the default constructor of the ComputeEngine_Runnable class is called,
                           outputStream and inputStream object streams are created for the purpose of setting up
                           the TCP connection based on the client's socket that is 1st argument in the constructor call.
 * Internal variables TBV: outputStream, inputStream
 * Mocked objects:         TCPserver, TCPclient, ClientManager, Socket
 * Mocked external methods: TCPserver.startServer()
 * Exceptions thrown:      IOException, InterruptedException
 ***********************************************************************************************************/
@Test
public void test_run_1() throws IOException, InterruptedException {

    mockTCPserverTest.getServerSocket().bind(new java.net.InetSocketAddress(port_1));
    mockTCPserverTest.startServer(mockTCPserverTest.getServerSocket());
    mockServerThread.start();

    tempClientSocket_1 = new Socket(serverHostName, port_1);
    when(mockTCPclient.getClientSocket()).thenReturn(tempClientSocket_1);

    obj_out_stream = new ObjectOutputStream(mockTCPclient.getClientSocket().getOutputStream());
    when(mockClientManager.getOutputStream()).thenReturn(obj_out_stream);
    Thread.sleep(20);

    comp_engine_1 = new ComputeEngine_Runnable(mock_CER_ClientSocket, global_watchdog_scale_factor, true);

    assertNotEquals(null,       comp_engine_1.getInputReaderStream());
    assertNotEquals(null,       comp_engine_1.getOutputStream());
}
```

Fig. 4.3. JUnit code snippet - a sample test case.

As mentioned in paragraph 3.3., a unit test requires a mocking framework. Mockito lets the software developer write extensive tests with a clean and simple API. Moreover, the tests are very readable and produce clean verification errors. The following features of Mockito are used in the unit tests:

- mock() - creates a mock object of given class or interface. Mocks in Mockito allow to stub invocations,
- spy() - creates a spy based on class instead of an object. Spies in Mockito allow to stub invocations as well as use real method invocations,

- when() - enables stubbing methods,
- thenReturn() - Sets a return value to be returned when the method is called.
- doAnswer() - stubs a void method with generic Answer,
- machers() - allows flexible verification or stubbing [9].



```java
// mocked objects
mockTCPserverTest = mock(TCPserver.class);
mock_CER_ClientSocket = mock(Socket.class);

// Mockito.doAnswer - to mock void method to do something (mock the behavior despite being void) -
// in this case it is used for TCPserver.startServer();
// the test uses this approach for the purpose of avoiding actual messages sent via TCP -
// it will be checked in the integration tests
Mockito.doAnswer(new Answer<Thread>() {
    @Override
    public Thread answer(InvocationOnMock invocation) throws Throwable {
        Object[] arguments = invocation.getArguments();
        if (arguments != null && arguments.length > 0 && arguments[0] != null ) {
            final ServerSocket servSocket = (ServerSocket) arguments[0];
            mockServerThread = new Thread(new Runnable() {
                public void run() {
                    while(!servSocket.isClosed()) {
                        try {
                            mock_CER_ClientSocket = servSocket.accept();
                            System.out.println("Server Thread Started.");
                        } catch (IOException IOex) {
                            mockServerThread.interrupt();
                            System.out.println("Server Thread Stopped.");
                            System.out.println("Server" + IOex.getMessage());
                            break;
                        }
                    }
                }
            });
        }
        return mockServerThread;
    }
}).when(mockTCPserverTest).startServer(Matchers.any(ServerSocket.class));
```

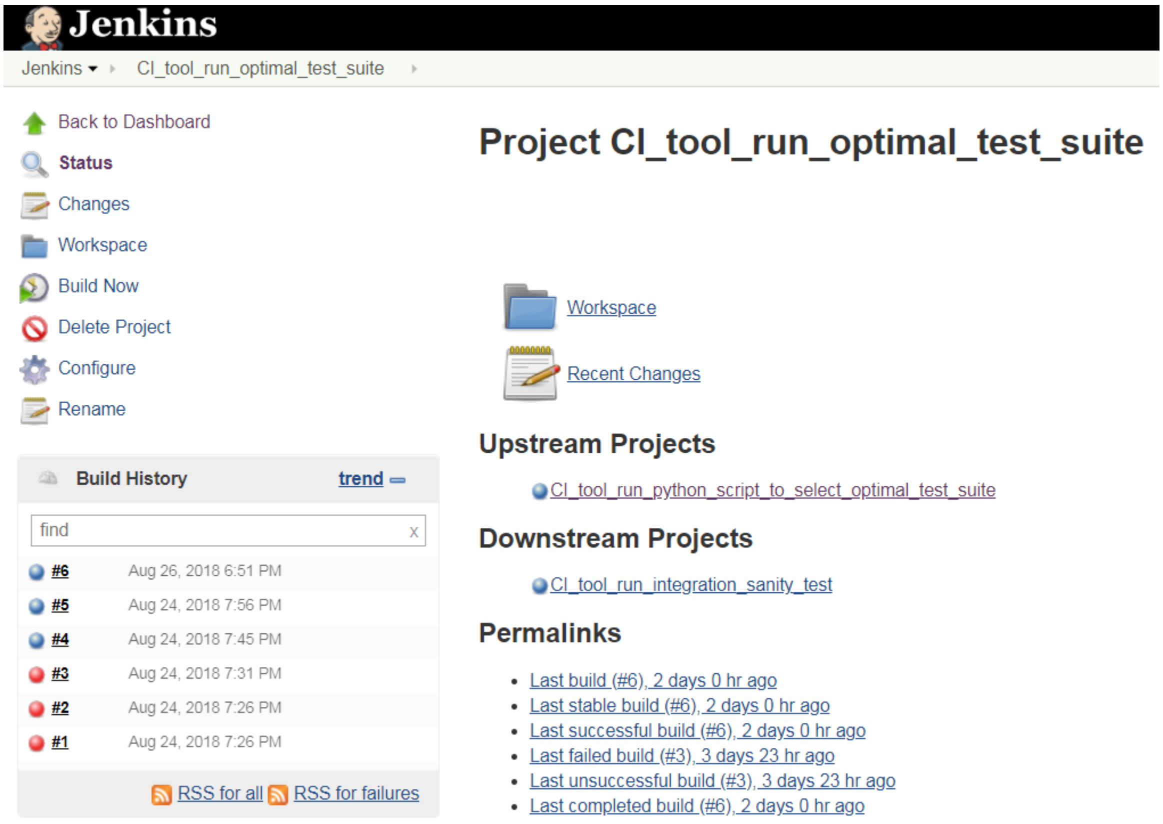
Fig. 4.4. JUnit code snippet - use of Mockito features to mock dependencies.

4.3. Continuous Integration automation server

The CI server used in the thesis is Jenkins. Basically, Jenkins is open source automation server written in Java SE 8 as a self-contained Java-based program available on every operating system. Jenkins configuration is very easy via its web interface.

There are several beneficial features of this continuous integration environment that enable Jenkins to automate the non-human part of the software development process. Jenkins is the leading CI automaton server with more than 1000 plugins in the Update Center. The plugins enable a software developer to integrate practically every tool in the continuous integration and continuous delivery toolchain. Due to Jenkins extensibility via plugins, there are nearly infinite possibilities for what Jenkins can do.

The Continuous Integration tool for an optimal test suite selection uses the various programming languages, testing frameworks and other tools. Therefore, Jenkins appears to be the ideal choice since it offers a simple way to create a continuous integration environment for almost any combination of languages, source code repositories and software project management tools using pipelines. A pipeline is a suite of plugins that defines the entire build process, which typically includes stages for building an application, testing it and then delivering it. By modeling a series of related tasks, pipelines add a powerful set of automation tools to Jenkins [10].


Jenkins

Jenkins ▾ › CI_tool_run_optimal_test_suite ›

Back to Dashboard
Status
Changes
Workspace
Build Now
Delete Project
Configure
Rename

Build History trend

find

#6	Aug 26, 2018 6:51 PM
#5	Aug 24, 2018 7:56 PM
#4	Aug 24, 2018 7:45 PM
#3	Aug 24, 2018 7:31 PM
#2	Aug 24, 2018 7:26 PM
#1	Aug 24, 2018 7:26 PM

RSS for all RSS for failures

Project CI_tool_run_optimal_test_suite

Workspace

Recent Changes

Upstream Projects

CI_tool_run_python_script_to_select_optimal_test_suite

Downstream Projects

CI_tool_run_integration_sanity_test

Permalinks

- Last build (#6), 2 days 0 hr ago
- Last stable build (#6), 2 days 0 hr ago
- Last successful build (#6), 2 days 0 hr ago
- Last failed build (#3), 3 days 23 hr ago
- Last unsuccessful build (#3), 3 days 23 hr ago
- Last completed build (#6), 2 days 0 hr ago

Fig. 4.5. Jenkins web interface for a project.

4.4. Version control system

The purpose of using a version control system is tracking changes in computer files and coordinating work on those files among multiple people. In the project, which is the subject of this thesis, all files that make any contribution to the thesis are stored in a git repository. The used git server is github, which makes it possible to display the contents of a Git repository via the web.

Various project artifacts are divided into independent groups of files that are stored on multiple branches. Basically, a branch contains the duplication of an object under revision control. Branching also generally enables all project artifacts and consequently a small pieces of each artifact to be developed in parallel by almost infinite number of team members in their respective areas of expertise. Once the changes such as new features or fixes for bugs are committed to the remote repository, these changes may be later merged after testing with the parent branch.

The remote repository is accessible with HTTPS the following web URL: https://github.com/AndSze/CI_tool_for_an_optimal_test_suite_selection.git. There are five branches created for the purpose of storing project artifacts:

- master - development branch for code of the application and all tests. It contains also project build configuration files used by Maven,
- develop - temporarily development branch, used for testing the CI tool,
- production - contains the code that has been successfully built and tested by the CI tool pipeline. The code on this branch is always up and running and can be delivered to a customer at any time,
- tests_selector - contains Python scripts used for an optimal test suite selection,
- JenkinsJobs – contains the Jenkins jobs the CI tool comprises of. There are also required plugins added to this branch that are necessary to execute the CI tool build pipeline since they do not belong to the set of plugins included by default.
- documents - contains the official documents that are required to get Master's Degree. This paper is also tracked on the documents branch, so anyone can read it or check out the history of changes made to the .docx file.

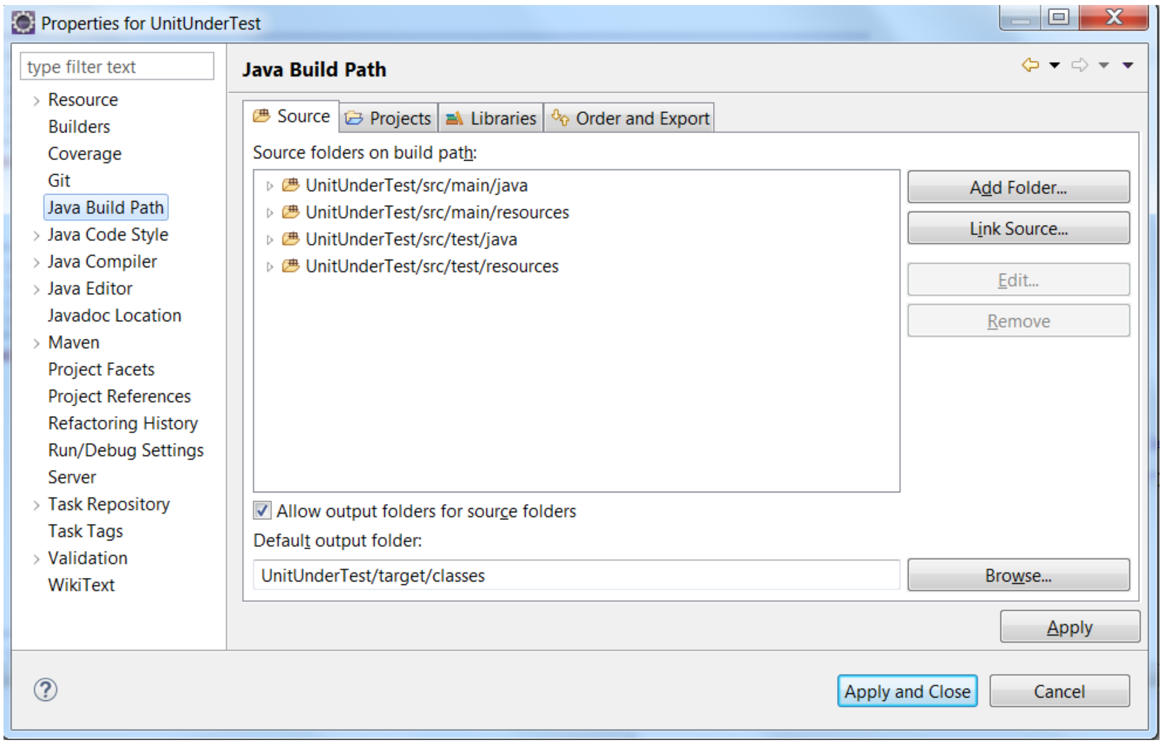
4.5. Integrated development environment

Integrated Development Environment (IDE) provides software developers with a possibility to manage workspaces, build, launch and debug applications and finally to share artifacts with a team and version the code. The IDE used in the thesis is Eclipse Oxygen Release (4.7.3a). Similarly to previously described software tools, Eclipse contains an extensible plug-in system for customizing the environment.

Eclipse is free and open-source software most widely used for software project development in Java programming language. A project in Eclipse includes a set of software development tools that are built together based on the source code implementation of the programs for a certain software package and software framework. The project after being built is a deliverable application.

From the thesis perspective, Eclipse does not only deliver the Java editor, but it also provides an opportunity to write JUnit tests and run the unit tests with Maven. Furthermore, Eclipse has a the Java Project Wizard that enables a software developer to define how the project should be built. The required build configurations in Eclipse are the default JRE and compiler compliance. Optional build configurations may be whether to optimize the code or include debug information.

To define the structure of a project, Eclipse uses source folders. During building Java project, source folders added to the Java build path are translated by the compiler to .class files that will be written to the output folder. The source folders mechanism allows, for example, to separate test from the application as shown in figure 4.6. Within a source folder, a more detailed structuring can be obtained by using packages.


Properties for UnitUnderTest

type filter text

Resource
Builders
Coverage
Git
Java Build Path
Java Code Style
Java Compiler
Java Editor
Javadoc Location
Maven
Project Facets
Project References
Refactoring History
Run/Debug Settings
Server
Task Repository
Task Tags
Validation
WikiText

Java Build Path

Source | Projects | Libraries | Order and Export

Source folders on build path:

UnitUnderTest/src/main/java
UnitUnderTest/src/main/resources
UnitUnderTest/src/test/java
UnitUnderTest/src/test/resources

Add Folder...
Link Source...
Edit...
Remove

[✓] Allow output folders for source folders

Default output folder:

UnitUnderTest/target/classes

Browse...

Apply

Apply and Close
Cancel

Fig. 4.6. Java Build Path configuration in Eclipse.

4.6. Software project management tool

Software project management tool may be a misleading name since this chapter is intended exclusively for the characteristic of the technical toolset used in the thesis. However, in case of software project management, tasks of managing the project do not focus solely on managing the team of engineers working on this project. The objective of software project management tool used for technical purpose is to define how software is built and describe software dependencies.

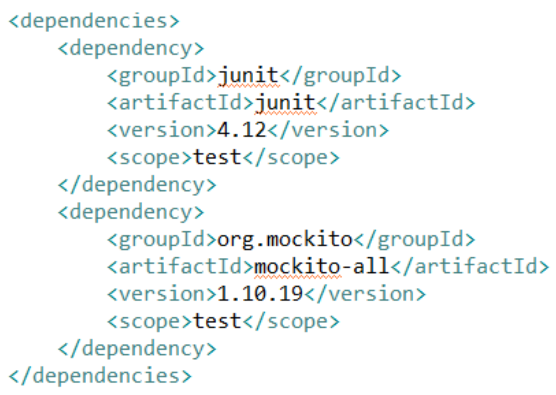
It was said in the above subparagraph that an IDE may also define build configuration for a project. A software project management tool would additionally enhance the build process, because it comes with pre-defined targets for performing certain well-defined tasks. These tasks may be, besides the above mentioned build process and dependencies management, description of external modules, components, all required plugins, the build order, directories and compilation of code and its packaging.

The software project management tool used in the thesis is Maven. It is delivered by Apache Software Foundation. Maven uses a concept of a project object model (POM), which is stored in pom.xml. POM provides all the configuration for a single project. In Maven, all work is done by plugins that will be executed during the build providing a set of goals such as compile, test, validate, install or deploy.

There are four main features of Maven used in the thesis. All features are configured under the following Maven project descriptors in pom.xml:

- <properties></properties> - Maven properties are value placeholder for properties that are accessible anywhere within a POM.
- <dependencies></ dependencies > - Maven dependency is an element that identifies individual artifacts such as software libraries or modules.
- <profiles></profiles> - Maven profile is an element that changes settings depending on the environment where it is being built.
- <build></build> - Maven build is an element that declares project's directory structure and manages plugins.

For example, the dependencies section of pom.xml is presented in figure 4.7. The purpose of the below Maven dependencies is to select a version of the testing frameworks used in the thesis: JUnit and Mockito.



```
<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.mockito</groupId>
        <artifactId>mockito-all</artifactId>
        <version>1.10.19</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Fig. 4.7. POM code snippet – dependencies section of pom.xml.

5. Unit Under Test

The unit under test is an application written in Java 8 SE. The high level concept of the application includes a data server and nodes that send data to the server. Data is transmitted between the server and nodes over the TCP/IP protocol using TCP sockets. Both TCP server and TCP clients program can read from and write to a socket.

5.1. Overview of the application

TCP server is a data server. TCP client is a software simulation of a smog sensor that sends its measurements to the TCP server. TCP server has two watchdogs: the first one determines the point in time when measurement data should be received. The second one has much longer time to expiration and it determines the point in time when a measurement history should be received. TCP client has one watchdog that is synchronized with the shorter server watchdog after each TCP connection.

The inputs for TCP server are port, list of sensors, number of sensors, measurements limit and watchdog scale factor. The list of sensors contains configuration settings for each sensor that are going to be sent to a particular sensor during initialization. The number of sensors defines how many threads are running in parallel on the TCP server side to handle TCP connection sessions with a particular sensor. The measurements limit defines number of measurement data messages that should be followed by a measurement history. The watchdog scale factor is going to be described in the next paragraph.

The outputs of TCP server are serialized files saved on a PC that is running the TCP server. There are at least two files expected from every sensor within a single TCP connection session. The first file includes the current configuration and state of a sensor, whereas the second file contains the measurements received from a sensor. There may be more files saved within a session if a measurement history is received or during the sensor initialization phase when the TCP server sends settings to a sensor.

There is only one TCP server, hence the TCPserver class is a static Java class. What it means is that there can be only one instance of the TCPserver class that acts like an entirely separate class. As a result, there is only one server socket. TCP server has to be launched prior to TCP client, because it listens for an incoming TCP connection. Basically, once TCPserver is started, it is going to start its watchdogs and hangs in a blocking function until it receives an incoming client request. Once a message from a TCP client is received, there is a separated thread started that is responsible for handling the TCP connection between the server socket and the client socket that sent that message. Once the TCP connection session is successful, the thread waits until all sensors finish their connection sessions. If all messages containing configuration data, measurement data and acknowledgments from all sensors have been processed on the TCP server side, the shorter watchdog is kicked. Every once in a while, when the longer server watchdog is close to expire, server threads would expect an additional message containing measurement history. In this case, after the successful TCP session both watchdogs are kicked. The detailed description of TCP server implementation is shown in figures 5.2. and 5.3.

The inputs for TCP clients are port, server host name and sensor ID. TCP clients have no direct outputs, because sensors send their data that is then further processed on the TCP server side.

There are multiple TCP clients, hence every TCP client has its unique socket. Basically, once a TCP client is started, it is going to transmit a message to the TCP server that should have been already launched. After the initialization phase and successful configuration, the TCP connection is ended and the client watchdog starts to count down. Once the client watchdog is close to expire, next TCP connection session is launched. If the session is successful what means that all messages expected by the TCP server have been sent, the watchdog is kicked and synchronized with the server watchdog and it counts down until it reaches the threshold for launching TCP connection session again. The detailed description of TCP client implementation is shown in figures 5.4 and 5.5.

The entire communication between the TCP server and TCP clients is watchdog-driven. If a client watchdog for any sensor expires, this sensor goes to the dead state and it is unable neither to send nor receive messages anymore. If any of server watchdogs expires, the application will crash.

5.2. Application code refactoring to make it testable

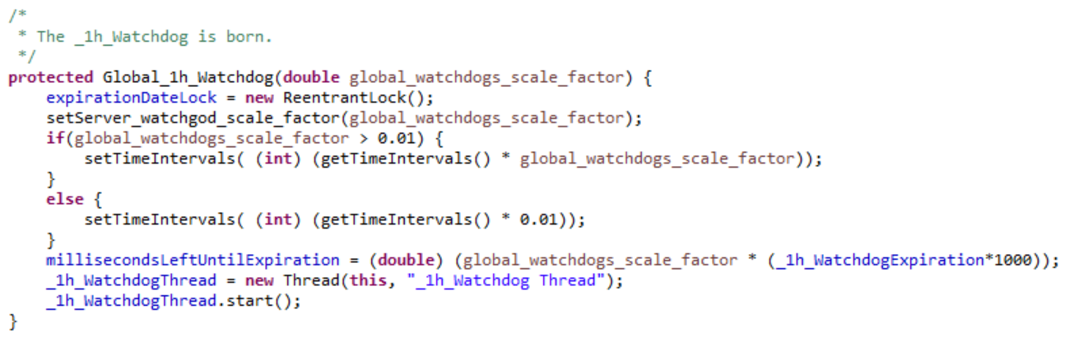
There have been a lot of minor changes made to many small pieces of the application code for the purpose of being able to verify those pieces of code. However, there is no need to put emphasis on those changes, since it is a general practice in the phase of writing unit tests.

It was a deliberate strategy to refer the server watchdogs as shorter and longer in the previous paragraph. Let's figure out, what should be the frequency between the points in time when measurement data or measurement history is received. A reasonable period of time for consecutive smog measurements is one hour. Measurement history, that contains all measurement data since the last message with measurement history has been sent, should be sent once every 24 hours.

Taking into account to the above periods of time, is it a dead end for the plan to write tests for the application? Is it worth taking an effort to write an integration test that is going to verify if a measurement history is received every 24 hours and the longer server watchdog has been kicked?

Obviously, execution of an integration test cannot last 24 hours. Fortunately, there is a solution to this kind of problem. The proposed solution creates an interface to the piece of code that is responsible for determining the points in time for setting up TCP connection sessions. The watchdog scale factor, which has been already mentioned in the previous paragraph, accelerates TCP connection execution cycles by decreasing default values of time left to expiration for the server watchdogs. Watchdog scale factors can be within the range of values between 0.001 to 1.0. If the watchdog scale factor equals 0.001, measurement data will be sent every 3.6 seconds, whereas measurement history will be sent every 86.4 seconds. If the watchdog scale factor has its default value of 1.0 that is going to be unchangeable in the production code, measurement data is expected by TCP server every 60 minutes, whereas measurement history is expected every 24 hours

Figure 5.1. shows that once the constructor of Global_1h_Watchdog is called, the parameter that determines a value of time left to expiration for the watchdog (millisecondsLeftUntilExpiration) is multiplied by global_watchdogs_scale_factor.



```
/*
 * The _1h_Watchdog is born.
 */
protected Global_1h_Watchdog(double global_watchdogs_scale_factor) {
    expirationDateLock = new ReentrantLock();
    setServer_watchgod_scale_factor(global_watchdogs_scale_factor);
    if(global_watchdogs_scale_factor > 0.01) {
        setTimeIntervals( (int) (getTimeIntervals() * global_watchdogs_scale_factor));
    }
    else {
        setTimeIntervals( (int) (getTimeIntervals() * 0.01));
    }
    millisecondsLeftUntilExpiration = (double) (global_watchdogs_scale_factor * (_1h_WatchdogExpiration*1000));
    _1h_WatchdogThread = new Thread(this, "_1h_Watchdog Thread");
    _1h_WatchdogThread.start();
}
```

Fig. 5.1. Interface in the application code that accelerates its execution cycles.

5.3. Block diagrams of application dataflow

Figure 5.2. illustrates how the TCP server code works. Figure 5.3. shows dataflow of ComputeEngine_Runnable task, which contains TCP server state machine for handling TCP connection. ComputeEngine_Runnable implements the Java Runnable interface since instances of ComputeEngine_Runnable are intended to be executed by a thread.

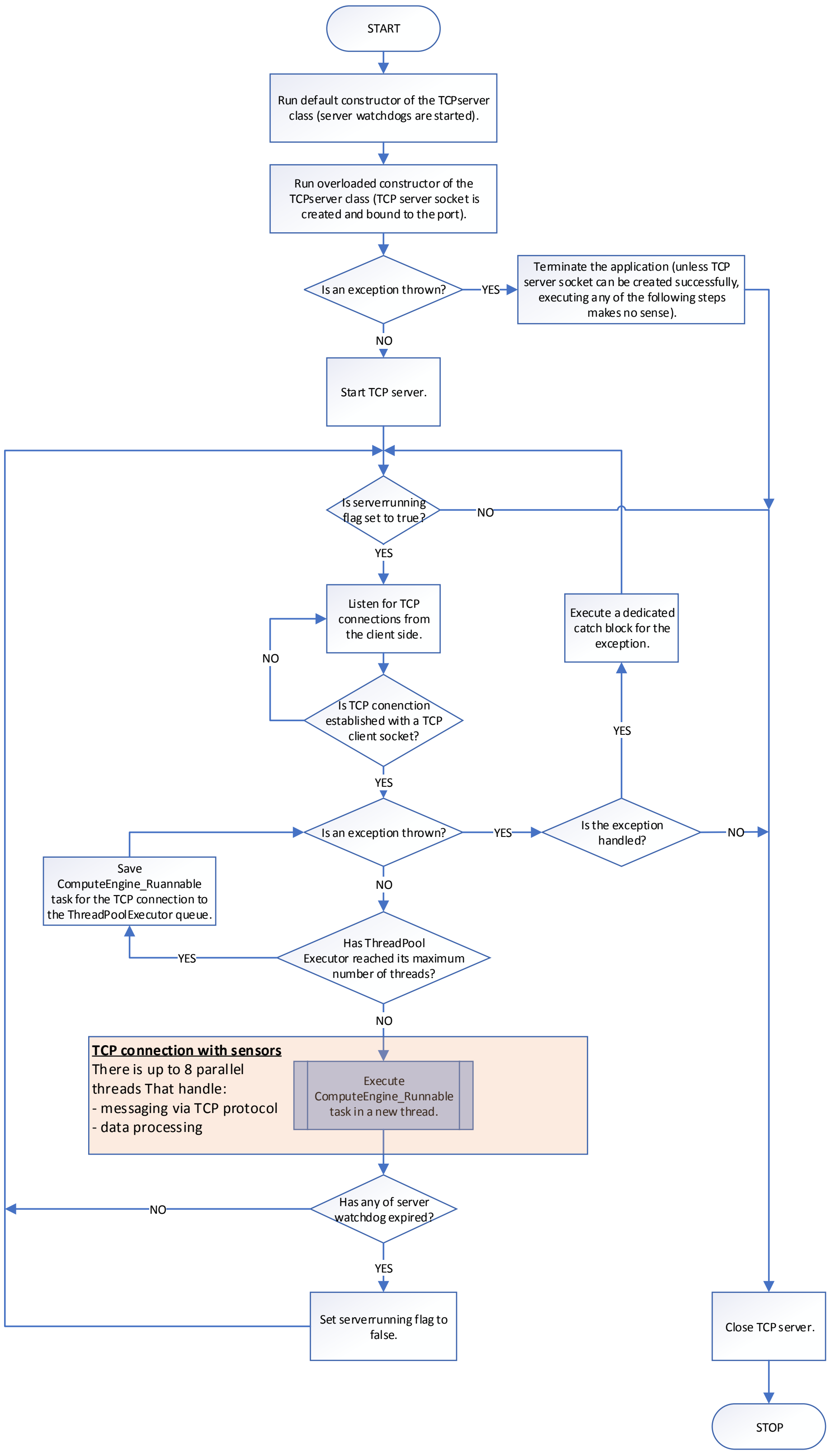

START
Run default constructor of the TCPserver class (server watchdogs are started).
Run overloaded constructor of the TCPserver class (TCP server socket is created and bound to the port).
Is an exception thrown?
YES
Terminate the application (unless TCP server socket can be created successfully, executing any of the following steps makes no sense).
NO
Start TCP server.
Is serverrunning flag set to true?
NO
YES
Listen for TCP connections from the client side.
NO
Is TCP conenction established with a TCP client socket?
YES
Execute a dedicated catch block for the exception.
YES
Is an exception thrown?
YES
Is the exception handled?
NO
Save ComputeEngine_Ruannable task for the TCP connection to the ThreadPoolExecutor queue.
NO
Has ThreadPool Executor reached its maximum number of threads?
YES
NO
TCP connection with sensors
There is up to 8 parallel threads That handle:
- messaging via TCP protocol
- data processing
Execute ComputeEngine_Runnable task in a new thread.
Has any of server watchdog expired?
NO
YES
Set serverrunning flag to false.
Close TCP server.
STOP

Fig. 5.2. Block diagram of application logic on the TCP server side.

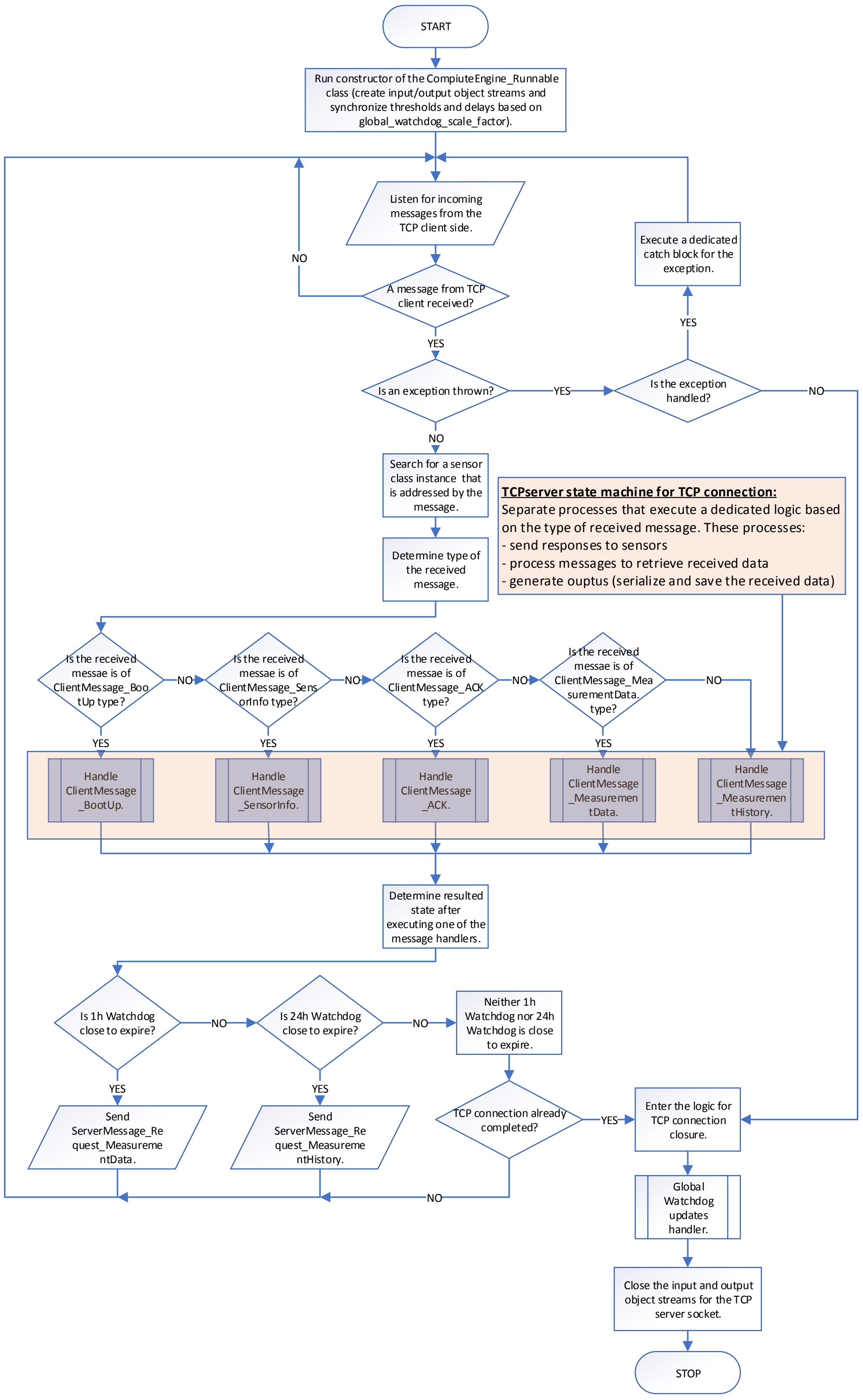

START
Run constructor of the CompiuteEngine_Runnable class (create input/output object streams and synchronize thresholds and delays based on global_watchdog_scale_factor).
Listen for incoming messages from the TCP client side.
A message from TCP client received?
NO
YES
Is an exception thrown?
YES
NO
Is the exception handled?
YES
NO
Execute a dedicated catch block for the exception.
Search for a sensor class instance that is addressed by the message.
Determine type of the received message.
TCPserver state machine for TCP connection:
Separate processes that execute a dedicated logic based on the type of received message. These processes:
- send responses to sensors
- process messages to retrieve received data
- generate ouptus (serialize and save the received data)
Is the received messae is of ClientMessage_BootUp type?
Is the received messae is of ClientMessage_SensorInfo type?
Is the received messae is of ClientMessage_ACK type?
Is the received messae is of ClientMessage_MeasurementData. type?
NO
NO
NO
NO
YES
YES
YES
YES
Handle ClientMessage _BootUp.
Handle ClientMessage _SensorInfo.
Handle ClientMessage _ACK.
Handle ClientMessage _MeasurementData.
Handle ClientMessage _MeasurementHistory.
Determine resulted state after executing one of the message handlers.
Is 1h Watchdog close to expire?
NO
Is 24h Watchdog close to expire?
NO
Neither 1h Watchdog nor 24h Watchdog is close to expire.
YES
YES
Send ServerMessage_Request_MeasurementData.
Send ServerMessage_Request_MeasurementHistory.
TCP connection already completed?
YES
NO
Enter the logic for TCP connection closure.
Global Watchdog updates handler.
Close the input and output object streams for the TCP server socket.
STOP

Fig. 5.3. Block diagram of ComputeEngine_Runnable class instances execution.

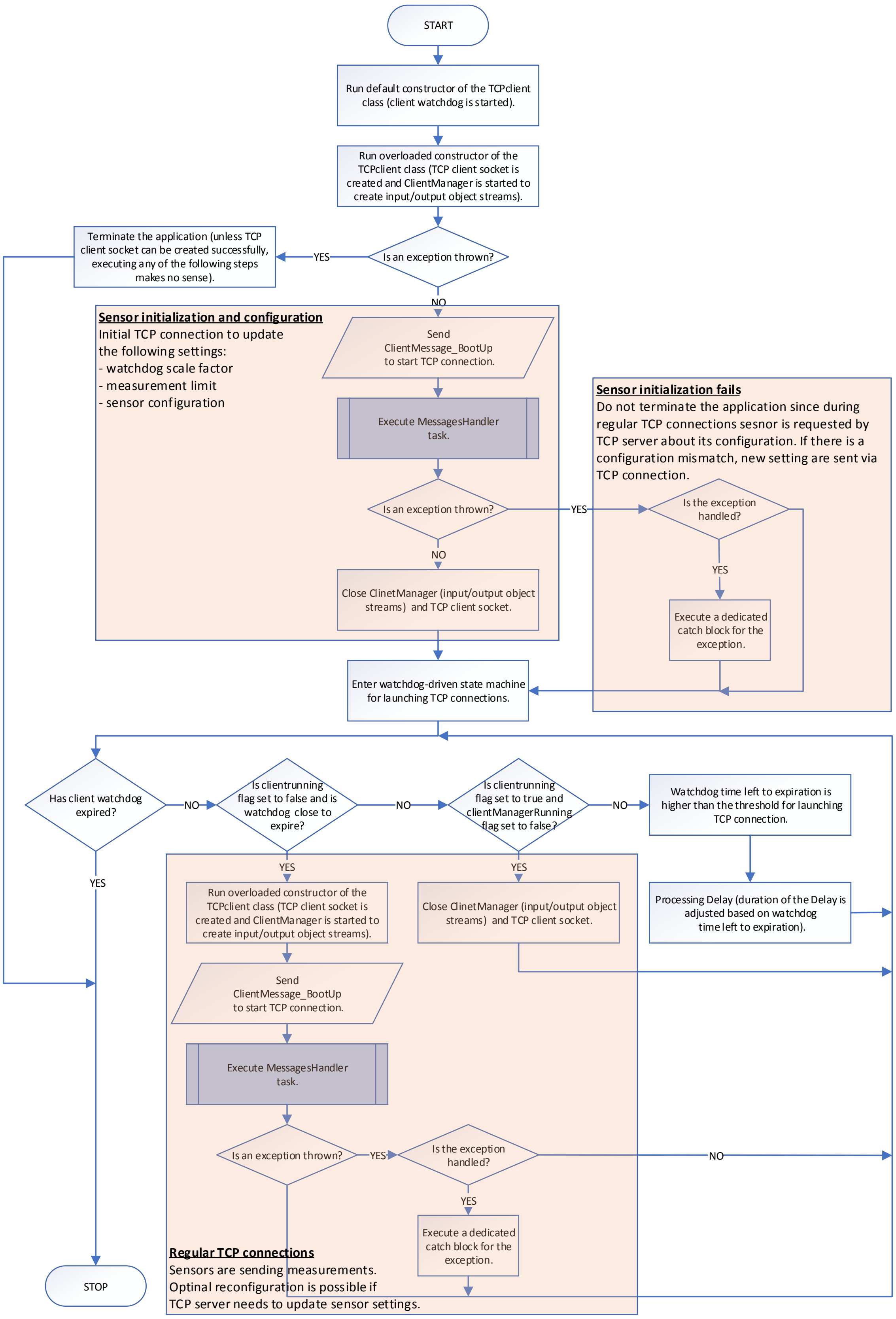

START
Run default constructor of the TCPclient class (client watchdog is started).
Run overloaded constructor of the TCPclient class (TCP client socket is created and ClientManager is started to create input/output object streams).
Is an exception thrown?
YES
Terminate the application (unless TCP client socket can be created successfully, executing any of the following steps makes no sense).
NO
Sensor initialization and configuration
Initial TCP connection to update the following settings:
- watchdog scale factor
- measurement limit
- sensor configuration
Send ClientMessage_BootUp to start TCP connection.
Execute MessagesHandler task.
Is an exception thrown?
YES
NO
Close ClinetManager (input/output object streams) and TCP client socket.
Sensor initialization fails
Do not terminate the application since during regular TCP connections sesnor is requested by TCP server about its configuration. If there is a configuration mismatch, new setting are sent via TCP connection.
Is the exception handled?
YES
Execute a dedicated catch block for the exception.
Enter watchdog-driven state machine for launching TCP connections.
Has client watchdog expired?
NO
Is clientrunning flag set to false and is watchdog close to expire?
NO
Is clientrunning flag set to true and clientManagerRunning flag set to false?
NO
Watchdog time left to expiration is higher than the threshold for launching TCP connection.
Processing Delay (duration of the Delay is adjusted based on watchdog time left to expiration).
YES
YES
YES
Run overloaded constructor of the TCPclient class (TCP client socket is created and ClientManager is started to create input/output object streams).
Close ClinetManager (input/output object streams) and TCP client socket.
Send ClientMessage_BootUp to start TCP connection.
Execute MessagesHandler task.
Is an exception thrown?
YES
Is the exception handled?
NO
YES
Execute a dedicated catch block for the exception.
Regular TCP connections
Sensors are sending measurements.
Optinal reconfiguration is possible if TCP server needs to update sensor settings.
STOP

Fig. 5.4. Block diagram of application logic on the TCP client side.

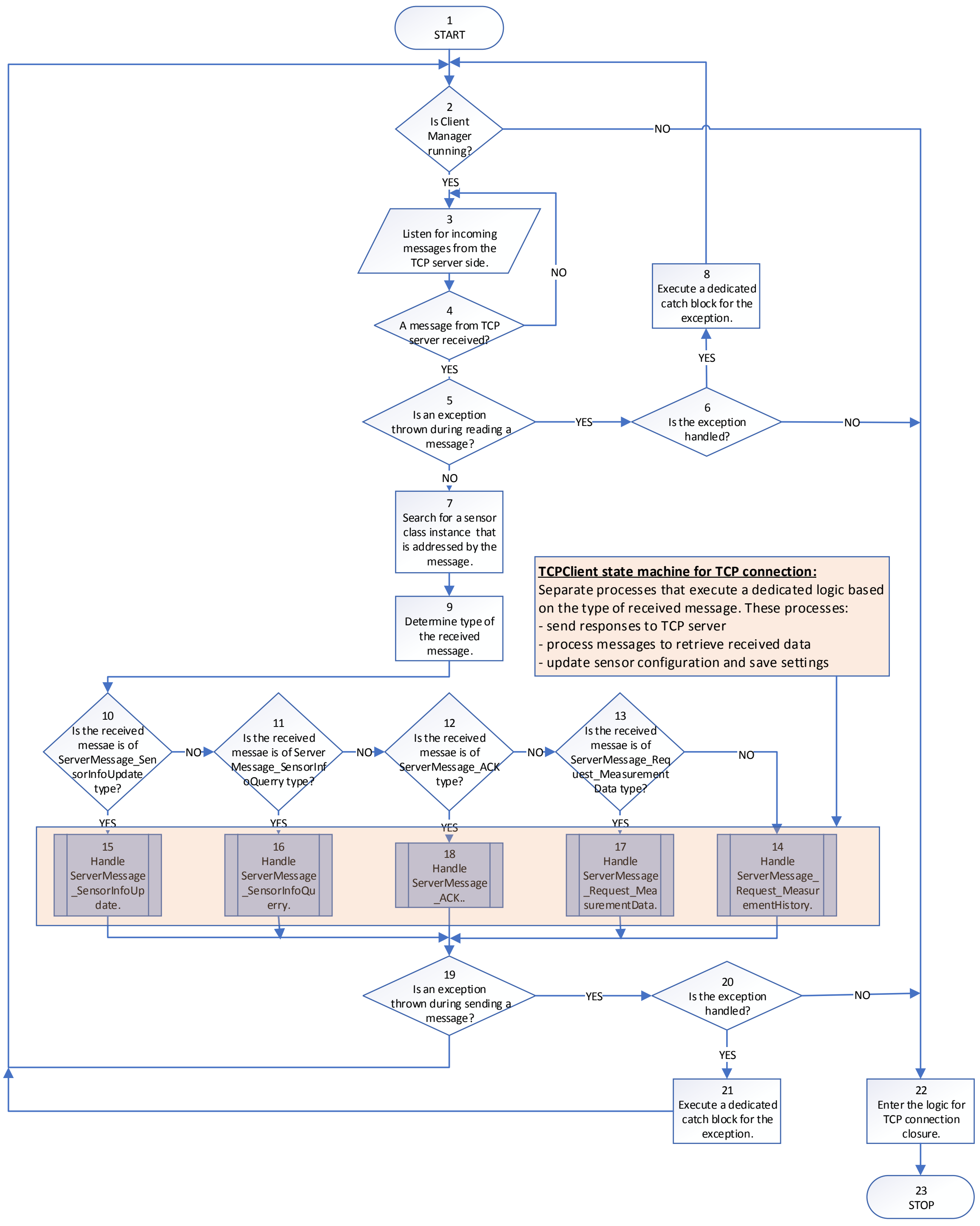

1 START
2 Is Client Manager running?
NO
YES
3 Listen for incoming messages from the TCP server side.
NO
4 A message from TCP server received?
YES
8 Execute a dedicated catch block for the exception.
YES
5 Is an exception thrown during reading a message?
YES
6 Is the exception handled?
NO
NO
7 Search for a sensor class instance that is addressed by the message.
TCPClient state machine for TCP connection:
Separate processes that execute a dedicated logic based on the type of received message. These processes:
- send responses to TCP server
- process messages to retrieve received data
- update sensor configuration and save settings
9 Determine type of the received message.
10 Is the received messae is of ServerMessage_SensorInfoUpdate type?
NO
11 Is the received messae is of ServerMessage_SensorInfoQuerry type?
NO
12 Is the received messae is of ServerMessage_ACK type?
NO
13 Is the received messae is of ServerMessage_Request_MeasurementData type?
NO
YES
YES
YES
YES
15 Handle ServerMessage_SensorInfoUpdate.
16 Handle ServerMessage_SensorInfoQuerry.
18 Handle ServerMessage_ACK..
17 Handle ServerMessage_Request_MeasurementData.
14 Handle ServerMessage_Request_MeasurementHistory.
19 Is an exception thrown during sending a message?
YES
20 Is the exception handled?
NO
YES
21 Execute a dedicated catch block for the exception.
22 Enter the logic for TCP connection closure.
23 STOP

Fig. 5.5. Block diagram of messagesHandler function in ClientManager class.

Figure 5.4. illustrates how the TCP client code works. On the TCP client side, TCPclient is the class that implements the Java Runnable Interface. Every instance of runnable TCPclient class has its ClientManager that implements messagesHandler function showed in figure 5.5. MessagesHandler contains TCP client state machine for handling TCP connection. The difference in implementation of interfaces that run threads between TCP server and TCP client is a result of the fact that TCPserver class is static.

6. Testware

There have been many testing artifacts created for the purpose of the application code verification. Generally, the prepared testware includes test plan definition, tests design, test inputs and expected results definition, test execution, test environment configuration and documentation.

The test plan involves an extensive set of unit tests and a comprehensive integration test that verifies if the application works as expected by executing a sanity check. There have been an emphasis put on unit tests with a view to achieving the main objective of the thesis, which is an optimal test suite selection. The entire unit test suite contains 173 tests. Given the fact that many of these tests are very complex, it can be said without a doubt that the test suite is able to verify the code thoroughly and provide a valuable feedback on its quality.

6.1. Test design

As mentioned in chapter 4., testing frameworks that have been used for tests development are JUnit and Mockito. The process of tests development for the application was no mean feat.

The complexity of application requires the use of sophisticated features of the testing frameworks. It is mostly caused by the fact that the TCP protocol defines a sequence of steps required to establish a connection and send data over the network. From the test design perspective, the most demanding task was figuring out a solution how to handle TCP sockets functionality that causes the application to hang until a particular response is received. For example, TCP server hangs in startServer function until Object Output Stream is created on the TCP client side. Then Object Output Stream waits for Object Input Stream from the server before proceeding. The second challenge was designing tests for an application that generates hardly any outputs but only TCP messages. Last but not least, it was very time consuming to stub/mock multiple dependencies that are needed to transmit data without exceptions.

Let’s take MessagesHandlerTest as an example. This test in order to be executable requires the creation of 4 threads. These threads implement the following functionalities:

- mockServerThread – avoiding remote deadlock,
- testThread_server - listening for messages on the server side,

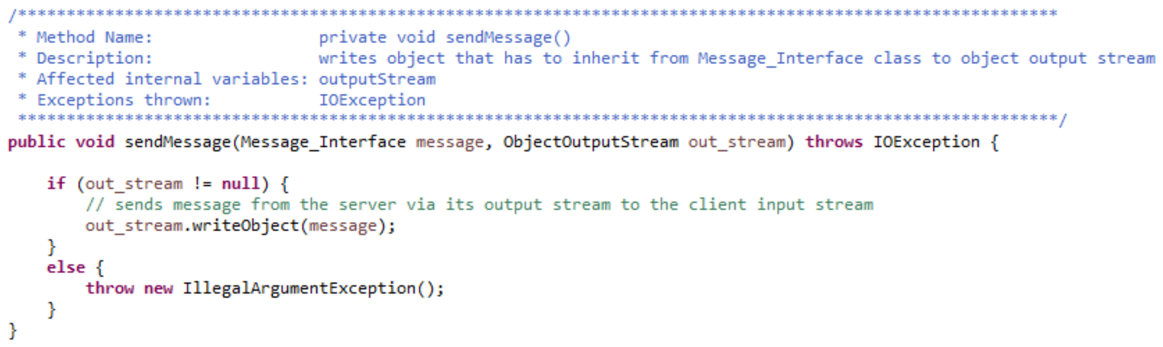
- testThread_client - listening for messages on the client side and resending particular responses for the received messages,
- Main test thread – executing test logic, setting inputs, validating outputs.

6.2. Tests inputs and expected results

Every function in the application has its function header that may include the following information:

- Method Name,
- Description,
- Affected internal variables,
- Affected external variables,
- Local variables,
- Called internal functions,
- Called external functions,
- Returned value,
- Exceptions thrown,
- Exceptions handled.

If any of the above parameters are not applicable to a function, the header of this function is limited only to the valid fields. The function header for sendMessage function is given in figure 6.1.



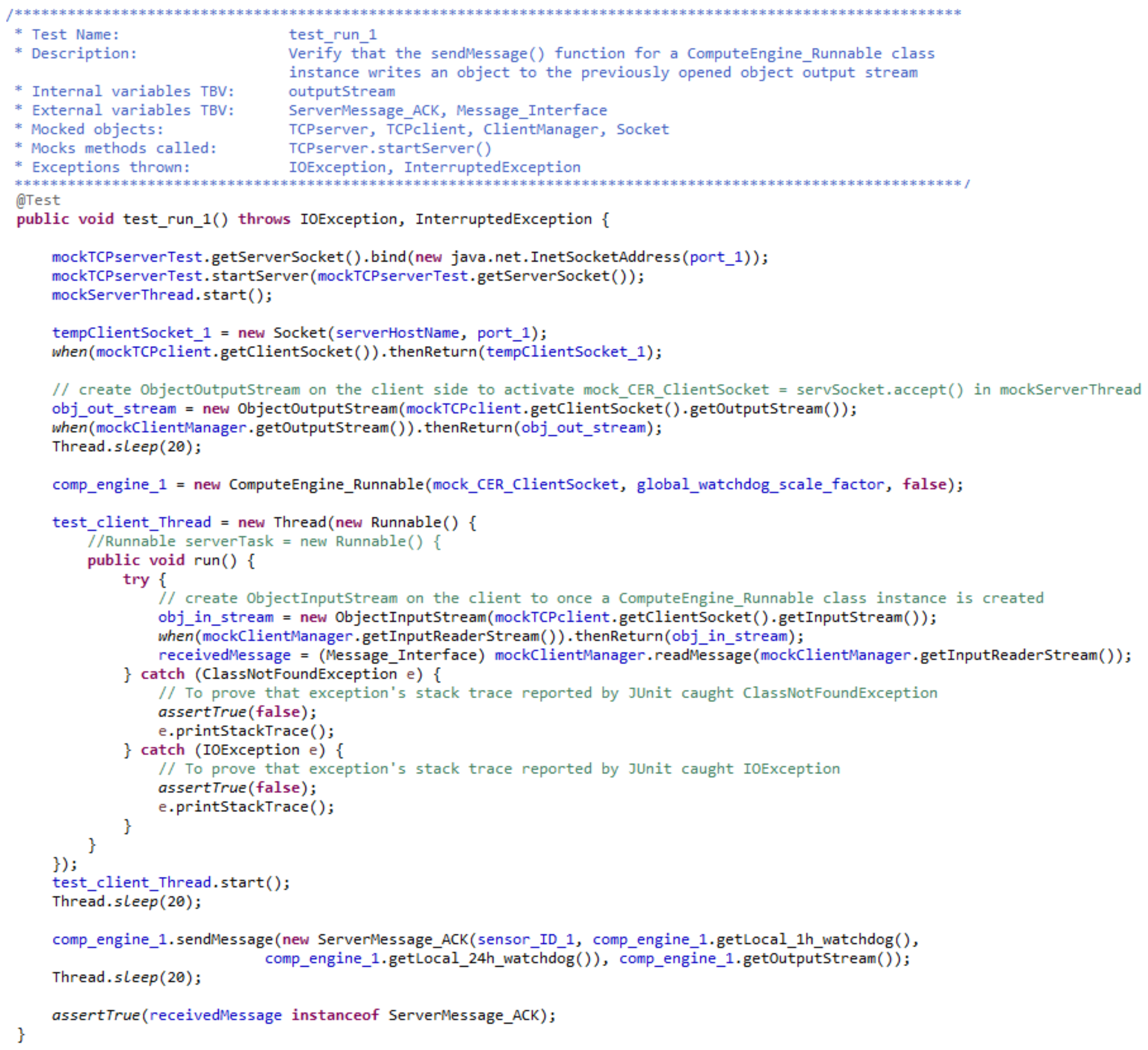
```
/***********************************************************************************************
 * Method Name:                 private void sendMessage()
 * Description:                 writes object that has to inherit from Message_Interface class to object output stream
 * Affected internal variables: outputStream
 * Exceptions thrown:           IOException
 ***********************************************************************************************/
public void sendMessage(Message_Interface message, ObjectOutputStream out_stream) throws IOException {

    if (out_stream != null) {
        // sends message from the server via its output stream to the client input stream
        out_stream.writeObject(message);
    }
    else {
        throw new IllegalArgumentException();
    }
}
```

Fig. 6.1. SendMessage function definition and header – TCP server side.

It is worth mentioning that the process of filling the fields in a function header based on reading the code of a function is also a testing activity. It is called static code analysis, which is the analysis of computer software that is performed without actually executing programs. The information derived from the analysis may detect incorrect implementation of individual statements or declarations and highlight possible coding errors.

The most important fields in a function header in terms of defining test inputs and expected results are affected internal variables, affected internal variables, local variables, returned value and exceptions thrown. Based on the sendMessage function header that affects outputStream internal variable and throws IOException, there is a need to write minimum 2 tests.

The first test (SendMessageTest.test_run1) verifies if sendMessage function writes an object to an object output stream. The code of SendMessageTest.test_run1 is available in figure 6.2. Test inputs are ServerMessage_ACK that is sent using the server socket over the TCP network and output stream, which actually writes this message to an outputStream. Expected test result is ServerMessage_ACK that is received on the TCP client side. In order to enable reading messages on the TCP client side, there is a need to create test_client_Thread. Moreover, there are 4 mocks of the following class object created: TCPserver, TCPclient, ClientManager and Socket.

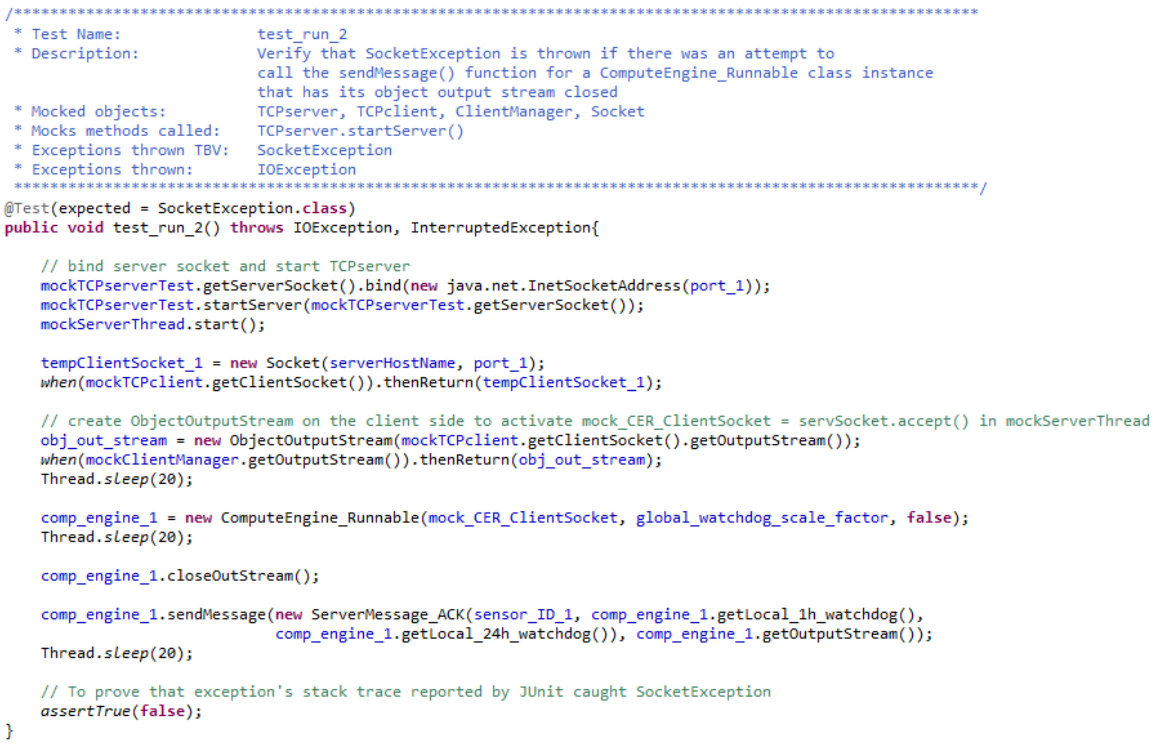


```
/*****************************************************************************************************************
 * Test Name:                   test_run_1
 * Description:                 Verify that the sendMessage() function for a ComputeEngine_Runnable class
                                instance writes an object to the previously opened object output stream
 * Internal variables TBV:      outputStream
 * External variables TBV:      ServerMessage_ACK, Message_Interface
 * Mocked objects:              TCPserver, TCPclient, ClientManager, Socket
 * Mocks methods called:        TCPserver.startServer()
 * Exceptions thrown:           IOException, InterruptedException
 *****************************************************************************************************************/
 @Test
 public void test_run_1() throws IOException, InterruptedException {

     mockTCPserverTest.getServerSocket().bind(new java.net.InetSocketAddress(port_1));
     mockTCPserverTest.startServer(mockTCPserverTest.getServerSocket());
     mockServerThread.start();

     tempClientSocket_1 = new Socket(serverHostName, port_1);
     when(mockTCPclient.getClientSocket()).thenReturn(tempClientSocket_1);

     // create ObjectOutputStream on the client side to activate mock_CER_ClientSocket = servSocket.accept() in mockServerThread
     obj_out_stream = new ObjectOutputStream(mockTCPclient.getClientSocket().getOutputStream());
     when(mockClientManager.getOutputStream()).thenReturn(obj_out_stream);
     Thread.sleep(20);

     comp_engine_1 = new ComputeEngine_Runnable(mock_CER_ClientSocket, global_watchdog_scale_factor, false);

     test_client_Thread = new Thread(new Runnable() {
         //Runnable serverTask = new Runnable() {
         public void run() {
             try {
                 // create ObjectInputStream on the client to once a ComputeEngine_Runnable class instance is created
                 obj_in_stream = new ObjectInputStream(mockTCPclient.getClientSocket().getInputStream());
                 when(mockClientManager.getInputReaderStream()).thenReturn(obj_in_stream);
                 receivedMessage = (Message_Interface) mockClientManager.readMessage(mockClientManager.getInputReaderStream());
             } catch (ClassNotFoundException e) {
                 // To prove that exception's stack trace reported by JUnit caught ClassNotFoundException
                 assertTrue(false);
                 e.printStackTrace();
             } catch (IOException e) {
                 // To prove that exception's stack trace reported by JUnit caught IOException
                 assertTrue(false);
                 e.printStackTrace();
             }
         }
     });
     test_client_Thread.start();
     Thread.sleep(20);

     comp_engine_1.sendMessage(new ServerMessage_ACK(sensor_ID_1, comp_engine_1.getLocal_1h_watchdog(),
                             comp_engine_1.getLocal_24h_watchdog()), comp_engine_1.getOutputStream());
     Thread.sleep(20);

     assertTrue(receivedMessage instanceof ServerMessage_ACK);
 }
```

Fig. 6.2. SendMessageTest.test_run1 unit test code.

Given the fact that sendMessage function throws IOException, an additional test was written to verify if SocketException is thrown if there was an attempt to send a message over the TCP network using sendMessage function when output stream has been closed. Test inputs are ServerMessage_ACK and output stream which has been previously closed by calling closeOutStream function. Expected test result is SocketException. To doublecheck that exception's stack trace reported by JUnit caught SocketException an additional assertion statement is added: assertTrue(false). Unless the exception was thrown, this assertion will cause test failure.

JUnit has a dedicated feature to test exceptions. In order to verify exceptions, the @Test annotation is extended with an optional parameter "expected”. The code of SendMessageTest.test_run2 is available in figure 6.3.



```
/*************************************************************************************************************
 * Test Name:                test_run_2
 * Description:              Verify that SocketException is thrown if there was an attempt to
                             call the sendMessage() function for a ComputeEngine_Runnable class instance
                             that has its object output stream closed
 * Mocked objects:           TCPserver, TCPclient, ClientManager, Socket
 * Mocks methods called:     TCPserver.startServer()
 * Exceptions thrown TBV:    SocketException
 * Exceptions thrown:        IOException
 *************************************************************************************************************/
@Test(expected = SocketException.class)
public void test_run_2() throws IOException, InterruptedException{

    // bind server socket and start TCPserver
    mockTCPserverTest.getServerSocket().bind(new java.net.InetSocketAddress(port_1));
    mockTCPserverTest.startServer(mockTCPserverTest.getServerSocket());
    mockServerThread.start();

    tempClientSocket_1 = new Socket(serverHostName, port_1);
    when(mockTCPclient.getClientSocket()).thenReturn(tempClientSocket_1);

    // create ObjectOutputStream on the client side to activate mock_CER_ClientSocket = servSocket.accept() in mockServerThread
    obj_out_stream = new ObjectOutputStream(mockTCPclient.getClientSocket().getOutputStream());
    when(mockClientManager.getOutputStream()).thenReturn(obj_out_stream);
    Thread.sleep(20);

    comp_engine_1 = new ComputeEngine_Runnable(mock_CER_ClientSocket, global_watchdog_scale_factor, false);
    Thread.sleep(20);

    comp_engine_1.closeOutStream();

    comp_engine_1.sendMessage(new ServerMessage_ACK(sensor_ID_1, comp_engine_1.getLocal_1h_watchdog(),
                             comp_engine_1.getLocal_24h_watchdog()), comp_engine_1.getOutputStream());
    Thread.sleep(20);

    // To prove that exception's stack trace reported by JUnit caught SocketException
    assertTrue(false);
}
```

Fig. 6.3. SendMessageTest.test_run2 unit test code.

6.3. Test plan

In this paragraph, there will be the high level approach to writing unit tests presented by the example of messagesHandler function. Generally, the used test methodology is decision coverage testing, which is designed to ensure that each of the possible branches from each decision point is executed at least once and thereby ensuring that all reachable code is executed. Then, outcomes of code execution are compared with the expected results for each branch.

Table 6.1. Test cases for messagesHandler function.

Transition between code blocks	Test name and test purpose
1.-2.-3.	MessagesHandlerTest.test_run_1: Verify that once the messagesHandler() function is called while the isClientManagerRunning flag is equal to true, the client manager class instance that runs the messagesHandler() function is able to read messages send from TCPserver.
1.-2.-22.	MessagesHandlerTest.test_run_2: Verify that once the messagesHandler() function is called while the isClientManagerRunning flag is set to false, the client manager class instance that runs the messagesHandler() function is NOT able to read messages send from TCPserver.
1.-2.-3.-4.	MessagesHandlerTest.test_run_3: Verify that once the messagesHandler() function is called, it hangs in the readMessage() function until it gets a new message from TCPserver. It is verified also that the state machine of messagesHandler() function is executed for all messages received from TCPserver, but every time the currently processing message is different from the previous.
4.-5.-7.	MessagesHandlerTest.test_run_4: Verify that once the messagesHandler() function receives any message from TCPserver that has different sensor ID than the sensor ID that is written to the Client Manager class instance, the state machine of messagesHandler() function sends ClientMessage_SensorInfo with the new sensor ID to confirm that this sensor ID was set intentionally. Verify also that the new sensor ID is also set for the sensor in Client_Sensors_LIST.
3.	ReadMessageTest.test_run_1: Verify that once the readMessage() function is called, an incoming message from TCPserver is read from an input object stream.
4.-5.-6.-8.	ReadMessageTest.test_run_2: Verify that SocketException is thrown if the client socket for an input object stream was closed while the readMessage() function was expecting for a new message from TCPserver.
4.-5.-6.-8.	ReadMessageTest.test_run_3: Verify that ClassCastException is thrown if the message received by the readMessage() function is not a message of the Message_Interface type.
10.-15.-30.	SendMessageTest.test_run_1: Verify that the sendMessage() function for the ClientManager class instance writes an object to the previously opened object output stream for a client socket.
19.-20.-21.	SendMessageTest.test_run_2: Verify that SocketException is thrown if there was an attempt to call the sendMessage() function for a client manager class instance that has its object output stream closed.

19.-20.-21.	SendMessageTest.test_run_3: Verify that IllegalArgumentException is thrown if there was an attempt to call the sendMessage() function for a client manager class instance without initializing its object output stream.

Table 6.1. includes test cases for messagesHandler function. The test cases are defined based on decision coverage testing strategy. First column in the table contains numbers of blocks in figure 5.5. These numbers are used for the purpose of illustrating transition between code blocks that are verified by a particular test case. Second column contains test name and test purpose with detailed description of test logic and values to be verified. The remaining transitions for the blue blocks in figure 5.5. that contain separate processes are covered by additional 19 test runs, which are not included in table 6.1.

6.4. Sanity check

Sanity check is a system integration test that examines outputs of the application. There is only one integration test, because this test is able to verify, assuming that there are no bugs found by the unit test suite, if the application works as expected.

The purpose of ApplicationSanityCheckIT is to verify that once the application is started it is endlessly sending messages via TCP connection between the TCP server and multiple TCP clients in parallel threads. Expected outputs are the serialized files that are saved after each TCP connection cycle on a PC disc that is running TCP server.

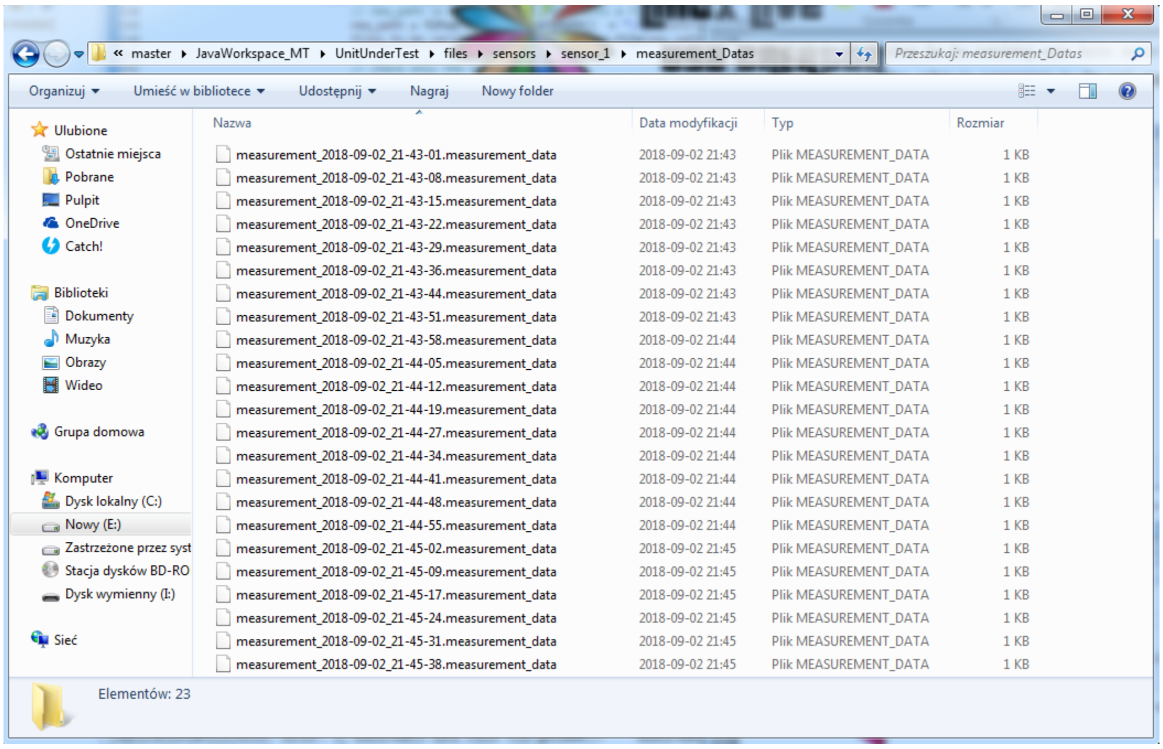
ApplicationSanityCheckIT inputs set comprises of the TCP server and TCP client inputs. Those inputs are listed below with their values that were used in the test execution:

- port = 8765,
- number_of_sensors = 5,
- watchdog_scale_factor = 0.002,
- measurements_limit = 24,
- serverHostName = "localhost".

There is one additional input for ApplicationSanityCheckIT:successful _TCPconenctions_threshold = 3. This threshold defines number of the whole cycles, in other words number of received measurement history data, after it is claimed that the test execution is passed. Since time of tests execution should not be too long, it is believed that checking that 72 measurement data messages have been received, the shorter watchdog has been kicked 72 times, 78 serialized sensor info files have been saved, 3 measurement history messages have been received and the longer watchdog has been kicked 3 times is a sufficient amount of information to prove that the application code is good enough to be merged with the production code.

In the figures 6.4. – 6.6. there are serialized files resulting from TCP connection between the TCP server and smog sensors. Figure 6.7. shows the test execution results in Eclipse as well as a snippet from Eclipse console that contains a log of application runtime and ApplicationSanityCheckIT logic. Time of the sanity check execution for the watchdog scale factor set to 0.002 equals 520 seconds. It is less than 9 minutes, what in comparison to the real application execution time is a tremendous saving of time.

In figure 6.4. it is shown that there are maximum 23 measurement data files serialized at one time due to the fact that once all measurements are received followed by a measurement history message, which contains the measurements from previous cycle, the measurement data files are immediately deleted in order not to duplicate data.

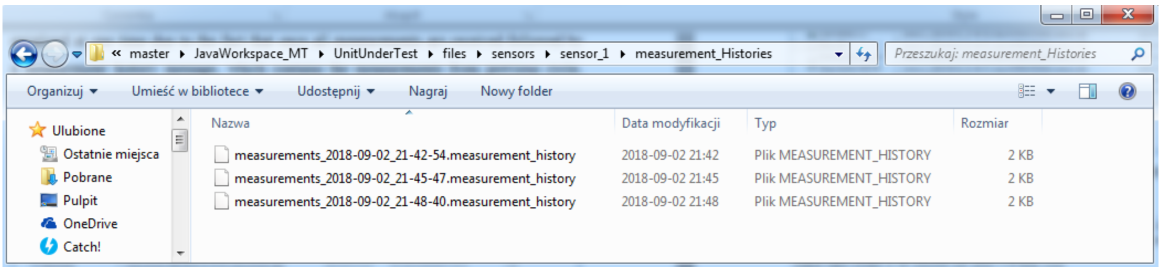


« master ▸ JavaWorkspace_MT ▸ UnitUnderTest ▸ files ▸ sensors ▸ sensor_1 ▸ measurement_Datas

Nazwa	Data modyfikacji	Typ	Rozmiar
measurement_2018-09-02_21-43-01.measurement_data	2018-09-02 21:43	Plik MEASUREMENT_DATA	1 KB
measurement_2018-09-02_21-43-08.measurement_data	2018-09-02 21:43	Plik MEASUREMENT_DATA	1 KB
measurement_2018-09-02_21-43-15.measurement_data	2018-09-02 21:43	Plik MEASUREMENT_DATA	1 KB
measurement_2018-09-02_21-43-22.measurement_data	2018-09-02 21:43	Plik MEASUREMENT_DATA	1 KB
measurement_2018-09-02_21-43-29.measurement_data	2018-09-02 21:43	Plik MEASUREMENT_DATA	1 KB
measurement_2018-09-02_21-43-36.measurement_data	2018-09-02 21:43	Plik MEASUREMENT_DATA	1 KB
measurement_2018-09-02_21-43-44.measurement_data	2018-09-02 21:43	Plik MEASUREMENT_DATA	1 KB
measurement_2018-09-02_21-43-51.measurement_data	2018-09-02 21:43	Plik MEASUREMENT_DATA	1 KB
measurement_2018-09-02_21-43-58.measurement_data	2018-09-02 21:44	Plik MEASUREMENT_DATA	1 KB
measurement_2018-09-02_21-44-05.measurement_data	2018-09-02 21:44	Plik MEASUREMENT_DATA	1 KB
measurement_2018-09-02_21-44-12.measurement_data	2018-09-02 21:44	Plik MEASUREMENT_DATA	1 KB
measurement_2018-09-02_21-44-19.measurement_data	2018-09-02 21:44	Plik MEASUREMENT_DATA	1 KB
measurement_2018-09-02_21-44-27.measurement_data	2018-09-02 21:44	Plik MEASUREMENT_DATA	1 KB
measurement_2018-09-02_21-44-34.measurement_data	2018-09-02 21:44	Plik MEASUREMENT_DATA	1 KB
measurement_2018-09-02_21-44-41.measurement_data	2018-09-02 21:44	Plik MEASUREMENT_DATA	1 KB
measurement_2018-09-02_21-44-48.measurement_data	2018-09-02 21:44	Plik MEASUREMENT_DATA	1 KB
measurement_2018-09-02_21-44-55.measurement_data	2018-09-02 21:44	Plik MEASUREMENT_DATA	1 KB
measurement_2018-09-02_21-45-02.measurement_data	2018-09-02 21:45	Plik MEASUREMENT_DATA	1 KB
measurement_2018-09-02_21-45-09.measurement_data	2018-09-02 21:45	Plik MEASUREMENT_DATA	1 KB
measurement_2018-09-02_21-45-17.measurement_data	2018-09-02 21:45	Plik MEASUREMENT_DATA	1 KB
measurement_2018-09-02_21-45-24.measurement_data	2018-09-02 21:45	Plik MEASUREMENT_DATA	1 KB
measurement_2018-09-02_21-45-31.measurement_data	2018-09-02 21:45	Plik MEASUREMENT_DATA	1 KB
measurement_2018-09-02_21-45-38.measurement_data	2018-09-02 21:45	Plik MEASUREMENT_DATA	1 KB

Elementów: 23

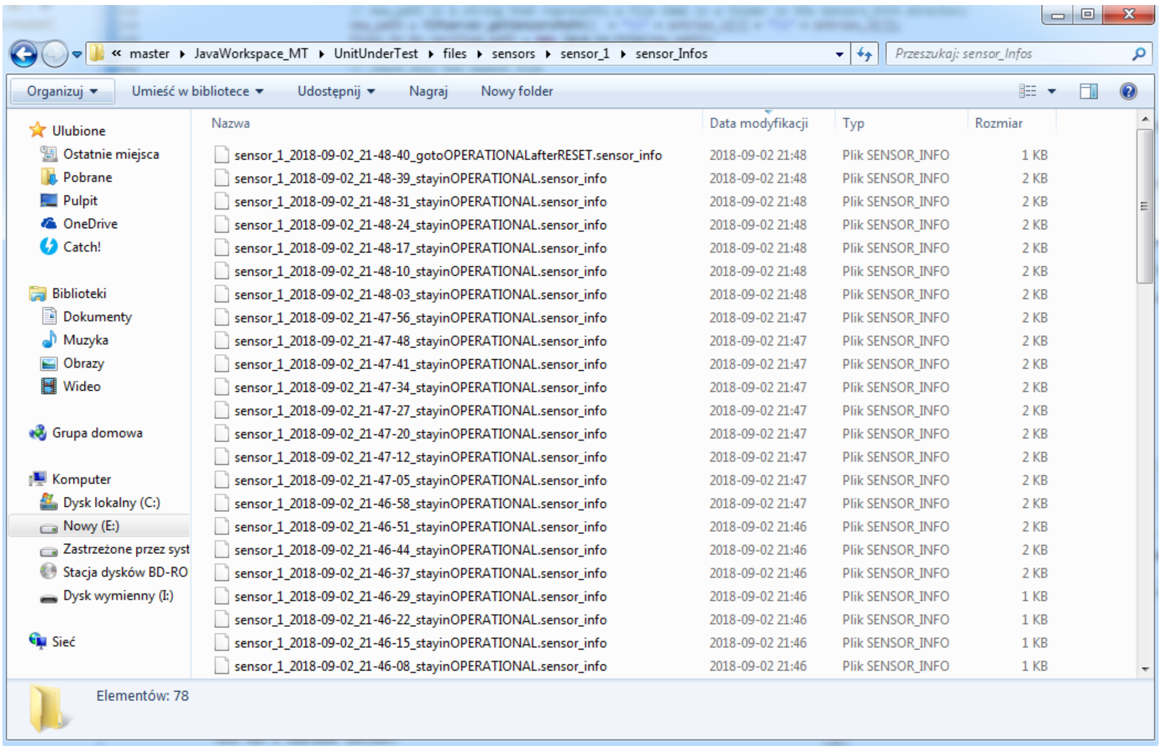
Fig. 6.4. ApplicationSanityCheckIT expected results – measurement data files saved by TCP server.



« master ▸ JavaWorkspace_MT ▸ UnitUnderTest ▸ files ▸ sensors ▸ sensor_1 ▸ measurement_Histories

Nazwa	Data modyfikacji	Typ	Rozmiar
measurements_2018-09-02_21-42-54.measurement_history	2018-09-02 21:42	Plik MEASUREMENT_HISTORY	2 KB
measurements_2018-09-02_21-45-47.measurement_history	2018-09-02 21:45	Plik MEASUREMENT_HISTORY	2 KB
measurements_2018-09-02_21-48-40.measurement_history	2018-09-02 21:48	Plik MEASUREMENT_HISTORY	2 KB

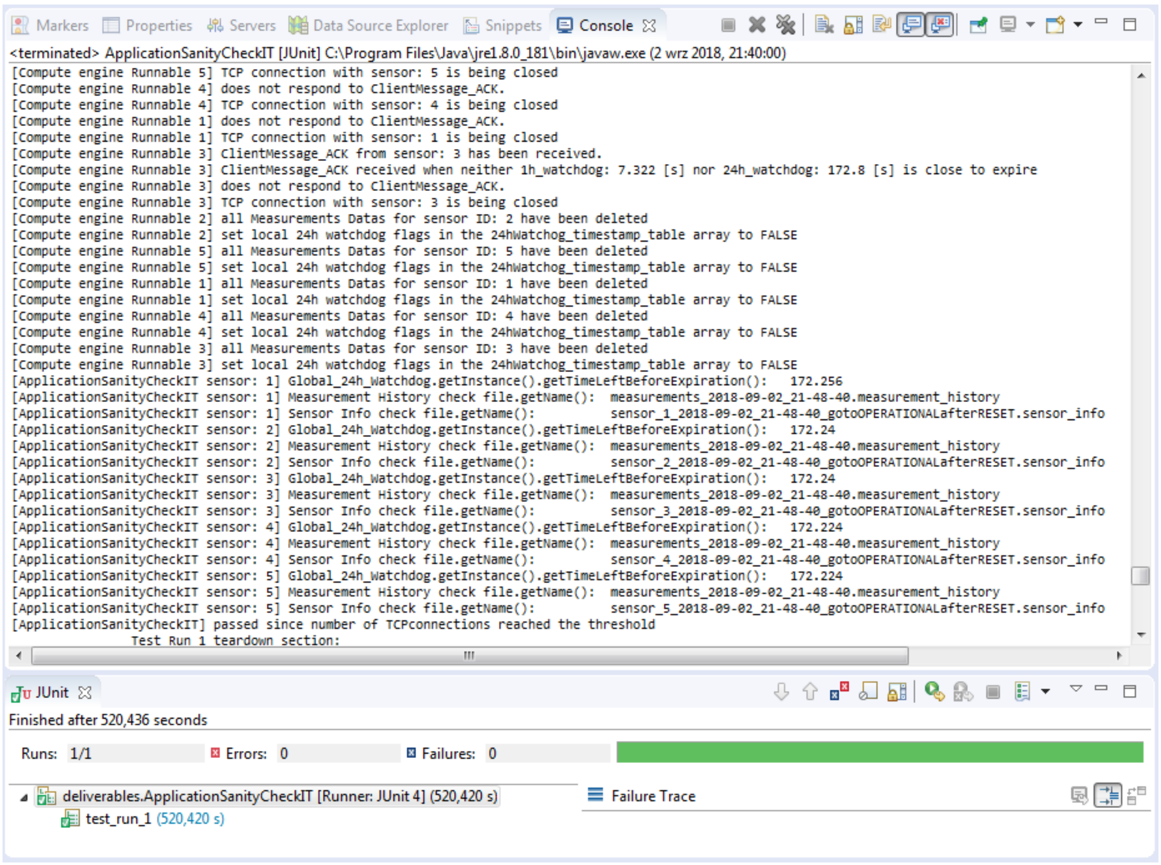
Fig. 6.5. ApplicationSanityCheckIT expected results – measurement history files saved by TCP server.



Nazwa	Data modyfikacji	Typ	Rozmiar
sensor_1_2018-09-02_21-48-40_gotoOPERATIONALafterRESET.sensor_info	2018-09-02 21:48	Plik SENSOR_INFO	1 KB
sensor_1_2018-09-02_21-48-39_stayinOPERATIONAL.sensor_info	2018-09-02 21:48	Plik SENSOR_INFO	2 KB
sensor_1_2018-09-02_21-48-31_stayinOPERATIONAL.sensor_info	2018-09-02 21:48	Plik SENSOR_INFO	2 KB
sensor_1_2018-09-02_21-48-24_stayinOPERATIONAL.sensor_info	2018-09-02 21:48	Plik SENSOR_INFO	2 KB
sensor_1_2018-09-02_21-48-17_stayinOPERATIONAL.sensor_info	2018-09-02 21:48	Plik SENSOR_INFO	2 KB
sensor_1_2018-09-02_21-48-10_stayinOPERATIONAL.sensor_info	2018-09-02 21:48	Plik SENSOR_INFO	2 KB
sensor_1_2018-09-02_21-48-03_stayinOPERATIONAL.sensor_info	2018-09-02 21:48	Plik SENSOR_INFO	2 KB
sensor_1_2018-09-02_21-47-56_stayinOPERATIONAL.sensor_info	2018-09-02 21:47	Plik SENSOR_INFO	2 KB
sensor_1_2018-09-02_21-47-48_stayinOPERATIONAL.sensor_info	2018-09-02 21:47	Plik SENSOR_INFO	2 KB
sensor_1_2018-09-02_21-47-41_stayinOPERATIONAL.sensor_info	2018-09-02 21:47	Plik SENSOR_INFO	2 KB
sensor_1_2018-09-02_21-47-34_stayinOPERATIONAL.sensor_info	2018-09-02 21:47	Plik SENSOR_INFO	2 KB
sensor_1_2018-09-02_21-47-27_stayinOPERATIONAL.sensor_info	2018-09-02 21:47	Plik SENSOR_INFO	2 KB
sensor_1_2018-09-02_21-47-20_stayinOPERATIONAL.sensor_info	2018-09-02 21:47	Plik SENSOR_INFO	2 KB
sensor_1_2018-09-02_21-47-12_stayinOPERATIONAL.sensor_info	2018-09-02 21:47	Plik SENSOR_INFO	2 KB
sensor_1_2018-09-02_21-47-05_stayinOPERATIONAL.sensor_info	2018-09-02 21:47	Plik SENSOR_INFO	2 KB
sensor_1_2018-09-02_21-46-58_stayinOPERATIONAL.sensor_info	2018-09-02 21:47	Plik SENSOR_INFO	2 KB
sensor_1_2018-09-02_21-46-51_stayinOPERATIONAL.sensor_info	2018-09-02 21:46	Plik SENSOR_INFO	2 KB
sensor_1_2018-09-02_21-46-44_stayinOPERATIONAL.sensor_info	2018-09-02 21:46	Plik SENSOR_INFO	2 KB
sensor_1_2018-09-02_21-46-37_stayinOPERATIONAL.sensor_info	2018-09-02 21:46	Plik SENSOR_INFO	2 KB
sensor_1_2018-09-02_21-46-29_stayinOPERATIONAL.sensor_info	2018-09-02 21:46	Plik SENSOR_INFO	1 KB
sensor_1_2018-09-02_21-46-22_stayinOPERATIONAL.sensor_info	2018-09-02 21:46	Plik SENSOR_INFO	1 KB
sensor_1_2018-09-02_21-46-15_stayinOPERATIONAL.sensor_info	2018-09-02 21:46	Plik SENSOR_INFO	1 KB
sensor_1_2018-09-02_21-46-08_stayinOPERATIONAL.sensor_info	2018-09-02 21:46	Plik SENSOR_INFO	1 KB

Elementów: 78

Fig. 6.6. ApplicationSanityCheckIT expected results – files containing sensor information (setting and state) saved by TCP server.



```
<terminated> ApplicationSanityCheckIT [JUnit] C:\Program Files\Java\jre1.8.0_181\bin\javaw.exe (2 wrz 2018, 21:40:00)
[Compute engine Runnable 5] TCP connection with sensor: 5 is being closed
[Compute engine Runnable 4] does not respond to ClientMessage_ACK.
[Compute engine Runnable 4] TCP connection with sensor: 4 is being closed
[Compute engine Runnable 1] does not respond to ClientMessage_ACK.
[Compute engine Runnable 1] TCP connection with sensor: 1 is being closed
[Compute engine Runnable 3] ClientMessage_ACK from sensor: 3 has been received.
[Compute engine Runnable 3] ClientMessage_ACK received when neither 1h_watchdog: 7.322 [s] nor 24h_watchdog: 172.8 [s] is close to expire
[Compute engine Runnable 3] does not respond to ClientMessage_ACK.
[Compute engine Runnable 3] TCP connection with sensor: 3 is being closed
[Compute engine Runnable 2] all Measurements Datas for sensor ID: 2 have been deleted
[Compute engine Runnable 2] set local 24h watchdog flags in the 24hWatchog_timestamp_table array to FALSE
[Compute engine Runnable 5] all Measurements Datas for sensor ID: 5 have been deleted
[Compute engine Runnable 5] set local 24h watchdog flags in the 24hWatchog_timestamp_table array to FALSE
[Compute engine Runnable 1] all Measurements Datas for sensor ID: 1 have been deleted
[Compute engine Runnable 1] set local 24h watchdog flags in the 24hWatchog_timestamp_table array to FALSE
[Compute engine Runnable 4] all Measurements Datas for sensor ID: 4 have been deleted
[Compute engine Runnable 4] set local 24h watchdog flags in the 24hWatchog_timestamp_table array to FALSE
[Compute engine Runnable 3] all Measurements Datas for sensor ID: 3 have been deleted
[Compute engine Runnable 3] set local 24h watchdog flags in the 24hWatchog_timestamp_table array to FALSE
[ApplicationSanityCheckIT sensor: 1] Global_24h_Watchdog.getInstance().getTimeLeftBeforeExpiration():   172.256
[ApplicationSanityCheckIT sensor: 1] Measurement History check file.getName():  measurements_2018-09-02_21-48-40.measurement_history
[ApplicationSanityCheckIT sensor: 1] Sensor Info check file.getName():          sensor_1_2018-09-02_21-48-40_gotoOPERATIONALafterRESET.sensor_info
[ApplicationSanityCheckIT sensor: 2] Global_24h_Watchdog.getInstance().getTimeLeftBeforeExpiration():   172.24
[ApplicationSanityCheckIT sensor: 2] Measurement History check file.getName():  measurements_2018-09-02_21-48-40.measurement_history
[ApplicationSanityCheckIT sensor: 2] Sensor Info check file.getName():          sensor_2_2018-09-02_21-48-40_gotoOPERATIONALafterRESET.sensor_info
[ApplicationSanityCheckIT sensor: 3] Global_24h_Watchdog.getInstance().getTimeLeftBeforeExpiration():   172.24
[ApplicationSanityCheckIT sensor: 3] Measurement History check file.getName():  measurements_2018-09-02_21-48-40.measurement_history
[ApplicationSanityCheckIT sensor: 3] Sensor Info check file.getName():          sensor_3_2018-09-02_21-48-40_gotoOPERATIONALafterRESET.sensor_info
[ApplicationSanityCheckIT sensor: 4] Global_24h_Watchdog.getInstance().getTimeLeftBeforeExpiration():   172.224
[ApplicationSanityCheckIT sensor: 4] Measurement History check file.getName():  measurements_2018-09-02_21-48-40.measurement_history
[ApplicationSanityCheckIT sensor: 4] Sensor Info check file.getName():          sensor_4_2018-09-02_21-48-40_gotoOPERATIONALafterRESET.sensor_info
[ApplicationSanityCheckIT sensor: 5] Global_24h_Watchdog.getInstance().getTimeLeftBeforeExpiration():   172.224
[ApplicationSanityCheckIT sensor: 5] Measurement History check file.getName():  measurements_2018-09-02_21-48-40.measurement_history
[ApplicationSanityCheckIT sensor: 5] Sensor Info check file.getName():          sensor_5_2018-09-02_21-48-40_gotoOPERATIONALafterRESET.sensor_info
[ApplicationSanityCheckIT] passed since number of TCPconnections reached the threshold
                Test Run 1 teardown section:
```

```
Finished after 520,436 seconds
Runs: 1/1        Errors: 0        Failures: 0

deliverables.ApplicationSanityCheckIT [Runner: JUnit 4] (520,420 s)
    test_run_1 (520,420 s)
```

Fig. 6.7. ApplicationSanityCheckIT successful execution and snippet from of output.

7. Optimal test suite selection

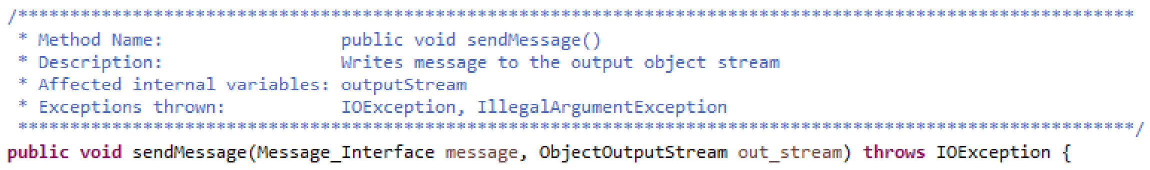
The idea behind the selection of an optimal test suite is to analyze changes that have been made to the code and then, based on these changes, select a group of tests to be executed. The selected test suite needs to be tailor-made for the changes. Ideally, the test cases that are going to be executed shall cover every piece of code that may be affected intentionally or unintentionally by the changes introduced between the commits that are submitted to an analysis.

The script that handles an optimal test suite selection is written in Python. Besides the advantages of Python itself that have been already listed in paragraph 4.1., Python supports interaction with git repositories. One of the most popular Python libraries used for the aforementioned purpose is GitPython. Not only does GitPython allow programmers to access a git repository by a pure Python language, it also implements the faster, but more resource intensive git command interface.

7.1. Inputs for the proposed algorithm

Similarly to the test inputs that have been described in paragraph 6.2., a source of information for the algorithm that selects an optimal test suite are function headers. From the test suite selection perspective, the most important fields in a function header are:

- Called internal functions,
- Called external functions.



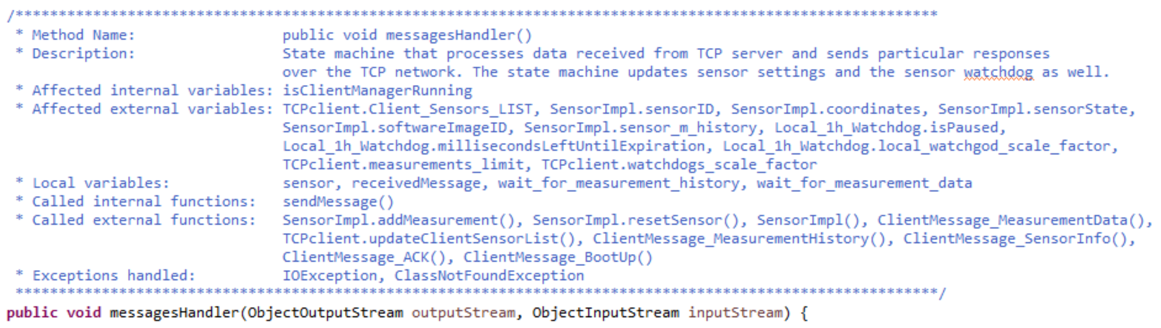
```
/*****************************************************************************************************
 * Method Name:                 public void sendMessage()
 * Description:                 Writes message to the output object stream
 * Affected internal variables: outputStream
 * Exceptions thrown:           IOException, IllegalArgumentException
 *****************************************************************************************************/
public void sendMessage(Message_Interface message, ObjectOutputStream out_stream) throws IOException {
```

Fig. 7.1. SendMessage function header – TCP client side.

Let us assume that sendMessage function has been changed in the latest code version. Undoubtedly, a test suite to be executed shall contain the test cases for sendMessage function. Besides these obligatory test cases, the test suite shall include all test cases that are likely to expose any defects introduced to the code by updates to sendMessage function.

First of all, it is assumed that changes to a function definition may affect only those pieces of code that directly call the function. An analysis, which would assess the potential impact of changes to a function to a piece of code that is not directly related to the function, would exceed the scope of the thesis.

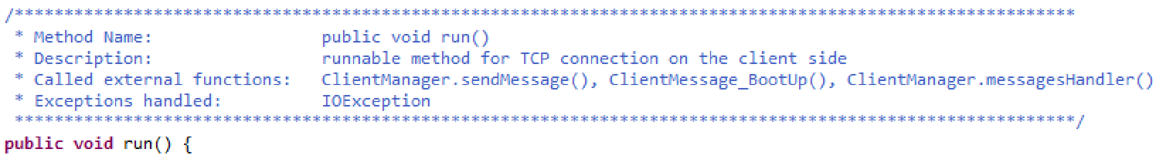
There are two possible scenarios that would trigger the algorithm to add a test case to an optimal test suite selected for analyzing the impact of changes to the code. In the first case scenario, the test cases for a function are going to be executed if this function calls the changed method, which is implemented in the same module. The algorithm reads function headers for all methods from a module that contains the changed code. It searches for the called internal functions field and checks if this filed refers to the changed function. If it does, test cases for the method that owns the function header are added to an optimal test suite. Figure 7.2. shows the header of messagesHandler function that includes sendMessage function in the called internal functions field. Thereby, if sendMessage function changes, test cases for messagesHandler function are executed.



```
/**********************************************************************************************
 * Method Name:                 public void messagesHandler()
 * Description:                 State machine that processes data received from TCP server and sends particular responses
                                over the TCP network. The state machine updates sensor settings and the sensor watchdog as well.
 * Affected internal variables: isClientManagerRunning
 * Affected external variables: TCPclient.Client_Sensors_LIST, SensorImpl.sensorID, SensorImpl.coordinates, SensorImpl.sensorState,
                                SensorImpl.softwareImageID, SensorImpl.sensor_m_history, Local_1h_Watchdog.isPaused,
                                Local_1h_Watchdog.millisecondsLeftUntilExpiration, Local_1h_Watchdog.local_watchgod_scale_factor,
                                TCPclient.measurements_limit, TCPclient.watchdogs_scale_factor
 * Local variables:             sensor, receivedMessage, wait_for_measurement_history, wait_for_measurement_data
 * Called internal functions:   sendMessage()
 * Called external functions:   SensorImpl.addMeasurement(), SensorImpl.resetSensor(), SensorImpl(), ClientMessage_MeasurementData(),
                                TCPclient.updateClientSensorList(), ClientMessage_MeasurementHistory(), ClientMessage_SensorInfo(),
                                ClientMessage_ACK(), ClientMessage_BootUp()
 * Exceptions handled:          IOException, ClassNotFoundException
 **********************************************************************************************/
public void messagesHandler(ObjectOutputStream outputStream, ObjectInputStream inputStream) {
```

Fig. 7.2. Function header for messagesHandler function that calls the modified internal method.

In case of the match for a modified function and the called external functions field the algorithm is roughly the same. The only difference is that there is a need to analyze function headers for all methods in the entire application. The called external functions field in a function header contains not only function names, but also the source code module, in which a function is defined. Figure 7.3. shows the function header of run function defined in TCPclient class that calls sendMessage function defined in ClientManager class source code module.



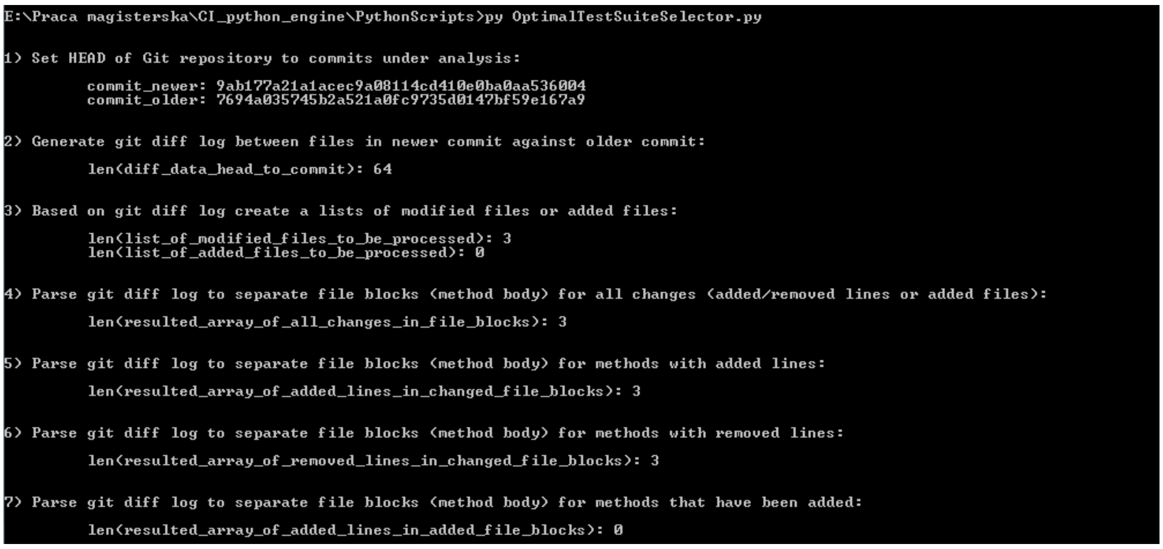
```
/***************************************************************************************************
 * Method Name:                 public void run()
 * Description:                 runnable method for TCP connection on the client side
 * Called external functions:   ClientManager.sendMessage(), ClientMessage_BootUp(), ClientManager.messagesHandler()
 * Exceptions handled:          IOException
 ***************************************************************************************************/
public void run() {
```

Fig. 7.3. Function header for run function calls the modified external method.

It is also worth mentioning that in this case, the algorithm is going to check packages prior to submitting a test case to execution since it is possible that there are more methods that have exactly the same name. The only chance to differentiate, which test cases shall be executed, is to verify if called external functions in a function header belong to the same package. The functions with the same name, but implemented in different packages – respectively tcpserver and tcpclient packages, are presented in figures 6.1. and 7.1.

7.2. Implementation of the proposed algorithm

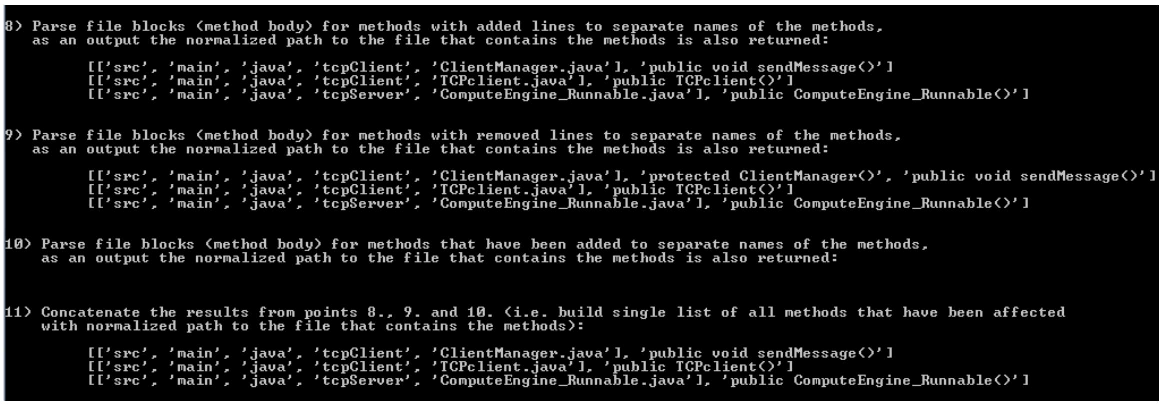
The script in Python contains 15 steps towards selecting an optimal test suite. In the first step commits under analysis are selected. By default the newest and the previous commits are selected. However, there is a possibility to compare the code between any commits. In step 2. there is git diff log file generated. This file highlights all changes that have been made to the code in newer commit against older commit. Then, the algorithm searches for source code modules that contain the changes and creates lists of modified files or added files. In case of a new source code file was added, it needs to be processed separately, because there is a difference in number of files in newer commit against older commit. In step 4. the git diff log file is parsed in order to separate all changes by grouping the modified code lines into file blocks. Then, each of the file block, which contains changes limited to scope of the affected function, is assigned to one of the following groups: a group of file blocks for methods with added lines, a group of file blocks for methods with removed lines or a group of file blocks that have been added.



```
E:\Praca magisterska\CI_python_engine\PythonScripts>py OptimalTestSuiteSelector.py

1) Set HEAD of Git repository to commits under analysis:

        commit_newer: 9ab177a21a1acec9a08114cd410e0ba0aa536004
        commit_older: 7694a035745b2a521a0fc9735d0147bf59e167a9

2) Generate git diff log between files in newer commit against older commit:

        len(diff_data_head_to_commit): 64

3) Based on git diff log create a lists of modified files or added files:

        len(list_of_modified_files_to_be_processed): 3
        len(list_of_added_files_to_be_processed): 0

4) Parse git diff log to separate file blocks (method body) for all changes (added/removed lines or added files):

        len(resulted_array_of_all_changes_in_file_blocks): 3

5) Parse git diff log to separate file blocks (method body) for methods with added lines:

        len(resulted_array_of_added_lines_in_changed_file_blocks): 3

6) Parse git diff log to separate file blocks (method body) for methods with removed lines:

        len(resulted_array_of_removed_lines_in_changed_file_blocks): 3

7) Parse git diff log to separate file blocks (method body) for methods that have been added:

        len(resulted_array_of_added_lines_in_added_file_blocks): 0
```

Fig. 7.4. The algorithm for an optimal test suite selection results – console output for steps 1.-7.

In the next steps, each file block belonging to one of the aforementioned groups is decomposed with a view to extracting method names that have been changed between code versions under analysis. Each group of file blocks is processed independently in consecutive steps from 8. to 10. Having in mind that function names are not a unique identifier and may be duplicated, the resulting lists of either methods with added or removed or new methods contain normalized path to modules, in which the affected methods are defined. In step 11. the lists that have been generated in steps from 8. to 10. are concatenated into a single list containing all affected methods.

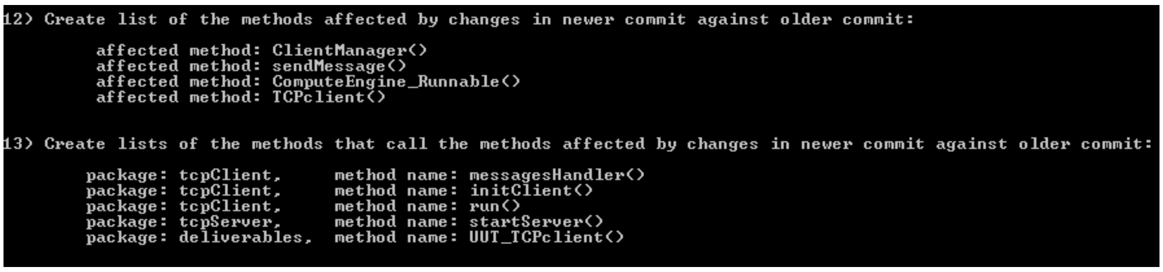


```
8) Parse file blocks (method body) for methods with added lines to separate names of the methods,
   as an output the normalized path to the file that contains the methods is also returned:

        [['src', 'main', 'java', 'tcpClient', 'ClientManager.java'], 'public void sendMessage()']
        [['src', 'main', 'java', 'tcpClient', 'TCPclient.java'], 'public TCPclient()']
        [['src', 'main', 'java', 'tcpServer', 'ComputeEngine_Runnable.java'], 'public ComputeEngine_Runnable()']

9) Parse file blocks (method body) for methods with removed lines to separate names of the methods,
   as an output the normalized path to the file that contains the methods is also returned:

        [['src', 'main', 'java', 'tcpClient', 'ClientManager.java'], 'protected ClientManager()', 'public void sendMessage()']
        [['src', 'main', 'java', 'tcpClient', 'TCPclient.java'], 'public TCPclient()']
        [['src', 'main', 'java', 'tcpServer', 'ComputeEngine_Runnable.java'], 'public ComputeEngine_Runnable()']

10) Parse file blocks (method body) for methods that have been added to separate names of the methods,
    as an output the normalized path to the file that contains the methods is also returned:


11) Concatenate the results from points 8., 9. and 10. (i.e. build single list of all methods that have been affected
    with normalized path to the file that contains the methods):

        [['src', 'main', 'java', 'tcpClient', 'ClientManager.java'], 'public void sendMessage()']
        [['src', 'main', 'java', 'tcpClient', 'TCPclient.java'], 'public TCPclient()']
        [['src', 'main', 'java', 'tcpServer', 'ComputeEngine_Runnable.java'], 'public ComputeEngine_Runnable()']
```

Fig. 7.5. The algorithm for an optimal test suite selection results – console output for steps 8.-11.

The steps that have been already described focus more on collecting data from the git repository. Steps that are going to be described concentrate entirely on selecting an optimal test suite and submitting the selected test suite for execution. These steps follow the procedure presented in the preceding paragraph.

In step 12. the resulted list from step 11. is processed to divide each affected method into a single element as well as get rid of modifies such as public, protected and the return type. In step 13. the algorithm reads function headers and searches for names of the modified methods in the internal and external called function fields. If there is a match, the name and package of a function that calls the modified methods are added to the list of methods that call methods affected by changes in the code. Result of the script execution for the example of modifications to sendMessage function and their potential impact on messagesHandler and run functions that have been described in paragraph 7.1. is illustrated in figure 7.6.

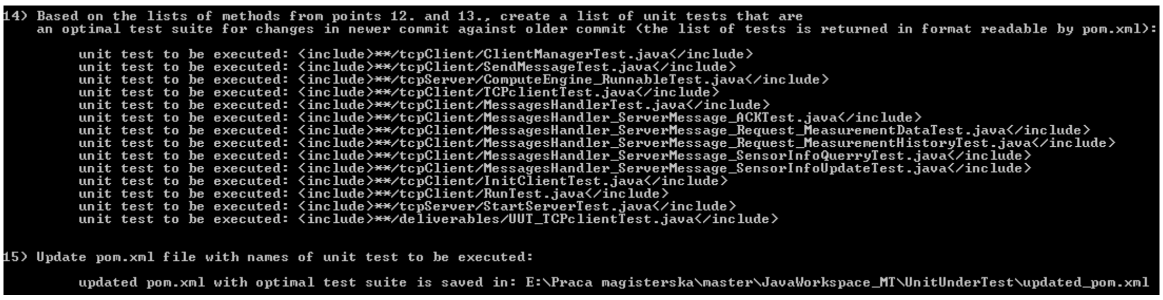


```
12) Create list of the methods affected by changes in newer commit against older commit:

         affected method: ClientManager()
         affected method: sendMessage()
         affected method: ComputeEngine_Runnable()
         affected method: TCPclient()

13) Create lists of the methods that call the methods affected by changes in newer commit against older commit:

        package: tcpClient,     method name: messagesHandler()
        package: tcpClient,     method name: initClient()
        package: tcpClient,     method name: run()
        package: tcpServer,     method name: startServer()
        package: deliverables,  method name: UUT_TCPclient()
```

Fig. 7.6. The algorithm for an optimal test suite selection results – console output for steps 12.-13.

In the last section of the Python script, a list of unit tests that have been selected for execution is created in step 14. There is a naming convention for unit tests in Maven that has been applied for all unit tests in the unit under test. The rule is a name of unit test shall begin with the name of function it verifies, which shall be followed by ‘Test’. As a result, the unit test for sendMessage function is named sendMessageTest. Due to JUnit limitation in number of tests runs that can be implemented in a single source code module, there are some unit tests that use an extended name of the first part containing the function name. The functions that require more than 10 test runs in order to verify their functionality are those functions that implement the state machines for TCP connection. The naming convention that has been applied for these unit tests is: the name of function followed by dash and the type of message that is handled by a piece of code that is verified followed by 'Test'. For example, a unit test for the case of the state machine on the TCP client side, which is messageHandler function, that processes ServerMessage_ACK is called messageHandler_ServerMessage_ACKTest.

Based on the lists of methods generated in steps 12. and 13. The list comprises of those test cases that verify each method listed before. Elements in the lists have a format that is compatible with the POM and can be read by Maven. In the last step pom.xml is updated with the list of unit tests to be executed and saved as updated_pom.xml.

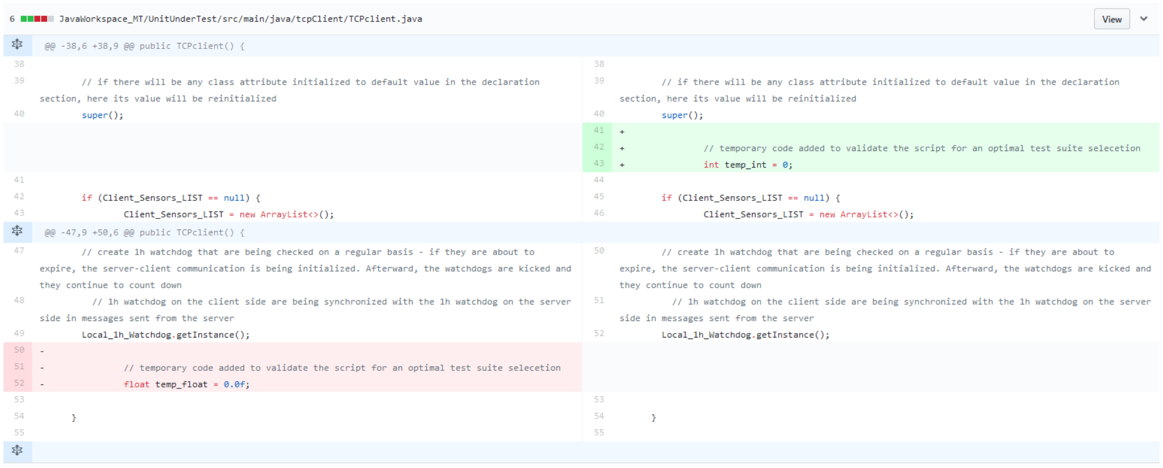


```
14) Based on the lists of methods from points 12. and 13., create a list of unit tests that are
    an optimal test suite for changes in newer commit against older commit (the list of tests is returned in format readable by pom.xml):

        unit test to be executed: <include>**/tcpClient/ClientManagerTest.java</include>
        unit test to be executed: <include>**/tcpClient/SendMessageTest.java</include>
        unit test to be executed: <include>**/tcpServer/ComputeEngine_RunnableTest.java</include>
        unit test to be executed: <include>**/tcpClient/TCPclientTest.java</include>
        unit test to be executed: <include>**/tcpClient/MessagesHandlerTest.java</include>
        unit test to be executed: <include>**/tcpClient/MessagesHandler_ServerMessage_ACKTest.java</include>
        unit test to be executed: <include>**/tcpClient/MessagesHandler_ServerMessage_Request_MeasurementDataTest.java</include>
        unit test to be executed: <include>**/tcpClient/MessagesHandler_ServerMessage_Request_MeasurementHistoryTest.java</include>
        unit test to be executed: <include>**/tcpClient/MessagesHandler_ServerMessage_SensorInfoQuerryTest.java</include>
        unit test to be executed: <include>**/tcpClient/MessagesHandler_ServerMessage_SensorInfoUpdateTest.java</include>
        unit test to be executed: <include>**/tcpClient/InitClientTest.java</include>
        unit test to be executed: <include>**/tcpClient/RunTest.java</include>
        unit test to be executed: <include>**/tcpServer/StartServerTest.java</include>
        unit test to be executed: <include>**/deliverables/UUT_TCPclientTest.java</include>


15) Update pom.xml file with names of unit test to be executed:

        updated pom.xml with optimal test suite is saved in: E:\Praca magisterska\master\JavaWorkspace_MT\UnitUnderTest\updated_pom.xml
```

Fig. 7.7. The algorithm for an optimal test suite selection results – console output for steps 14.-15.

7.3. Outputs of the proposed algorithm

Although the results of the algorithm execution have been already presented in the preceding paragraph, this section is added for the purpose of validating the results. There are screenshots from github attached below to document the changes that have been made to ClientManager, sendMessage, ComputeEngine_Runnable and TCPclient methods between the code versions under analysis. In conclusion, comparing the code changes illustrated in figures 7.8., 7.9. and 7.10. with the Python script outputs presented in figures 7.4., 7.5., 7.6. and 7.7. and the function headers in figures 7.1., 7.2. and 7.3., it can be stated that the algorithm is working correctly.

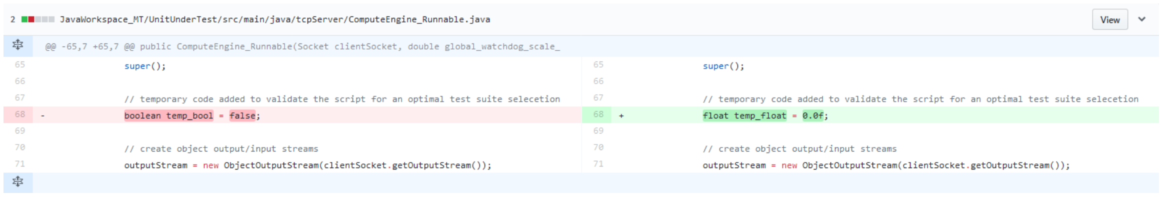


```
6 JavaWorkspace_MT/UnitUnderTest/src/main/java/tcpClient/TCPclient.java                    View

@@ -38,6 +38,9 @@ public TCPclient() {
```

```
        // if there will be any class attribute initialized to default value in the declaration section, here its value will be reinitialized
        super();

        if (Client_Sensors_LIST == null) {
                Client_Sensors_LIST = new ArrayList<>();
@@ -47,9 +50,6 @@ public TCPclient() {
        // create 1h watchdog that are being checked on a regular basis - if they are about to expire, the server-client communication is being initialized. Afterward, the watchdogs are kicked and they continue to count down
          // 1h watchdog on the client side are being synchronized with the 1h watchdog on the server side in messages sent from the server
        Local_1h_Watchdog.getInstance();
-
-               // temporary code added to validate the script for an optimal test suite selecetion
-               float temp_float = 0.0f;

      }

```

```
        // if there will be any class attribute initialized to default value in the declaration section, here its value will be reinitialized
        super();
+
+               // temporary code added to validate the script for an optimal test suite selecetion
+               int temp_int = 0;

        if (Client_Sensors_LIST == null) {
                Client_Sensors_LIST = new ArrayList<>();

        // create 1h watchdog that are being checked on a regular basis - if they are about to expire, the server-client communication is being initialized. Afterward, the watchdogs are kicked and they continue to count down
          // 1h watchdog on the client side are being synchronized with the 1h watchdog on the server side in messages sent from the server
        Local_1h_Watchdog.getInstance();

      }

```

Fig. 7.8. Github side-by-side diff for the code of TCPclass.java in commits under analysis.

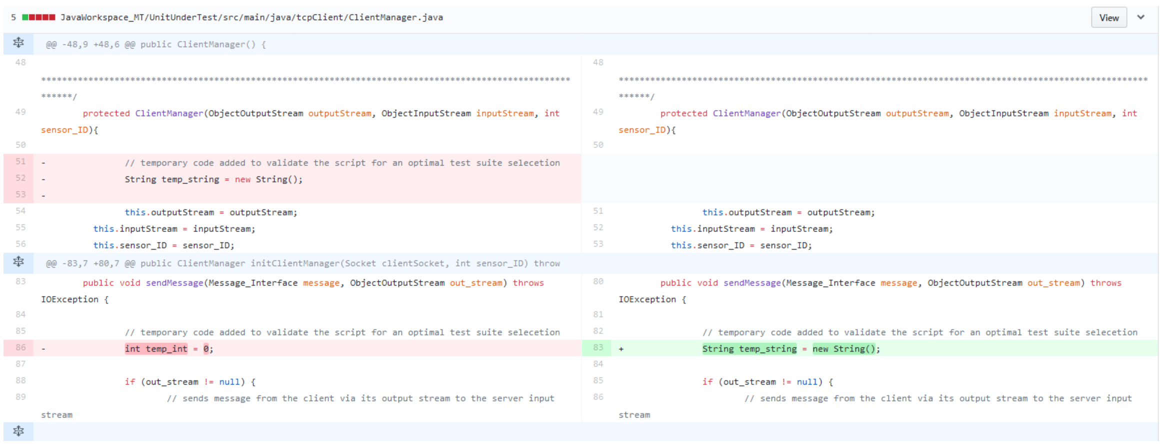


```
2 JavaWorkspace_MT/UnitUnderTest/src/main/java/tcpServer/ComputeEngine_Runnable.java                    View

@@ -65,7 +65,7 @@ public ComputeEngine_Runnable(Socket clientSocket, double global_watchdog_scale_
```

```
               super();

               // temporary code added to validate the script for an optimal test suite selecetion
-              boolean temp_bool = false;

               // create object output/input streams
               outputStream = new ObjectOutputStream(clientSocket.getOutputStream());
```

```
               super();

               // temporary code added to validate the script for an optimal test suite selecetion
+              float temp_float = 0.0f;

               // create object output/input streams
               outputStream = new ObjectOutputStream(clientSocket.getOutputStream());
```

Fig. 7.9. Github side-by-side diff for the code of ComputeEngine_Runnable.java in commits under analysis.



```
5 JavaWorkspace_MT/UnitUnderTest/src/main/java/tcpClient/ClientManager.java                    View

@@ -48,9 +48,6 @@ public ClientManager() {
```

```
*********************************************************************************************************/
        protected ClientManager(ObjectOutputStream outputStream, ObjectInputStream inputStream, int sensor_ID){

-               // temporary code added to validate the script for an optimal test suite selecetion
-               String temp_string = new String();
-
                this.outputStream = outputStream;
            this.inputStream = inputStream;
            this.sensor_ID = sensor_ID;
@@ -83,7 +80,7 @@ public ClientManager initClientManager(Socket clientSocket, int sensor_ID) throw
        public void sendMessage(Message_Interface message, ObjectOutputStream out_stream) throws IOException {

                // temporary code added to validate the script for an optimal test suite selecetion
-               int temp_int = 0;

                if (out_stream != null) {
                        // sends message from the client via its output stream to the server input stream
```

```
*********************************************************************************************************/
        protected ClientManager(ObjectOutputStream outputStream, ObjectInputStream inputStream, int sensor_ID){

                this.outputStream = outputStream;
            this.inputStream = inputStream;
            this.sensor_ID = sensor_ID;

        public void sendMessage(Message_Interface message, ObjectOutputStream out_stream) throws IOException {

                // temporary code added to validate the script for an optimal test suite selecetion
+               String temp_string = new String();

                if (out_stream != null) {
                        // sends message from the client via its output stream to the server input stream
```

Fig. 7.10. Github side-by-side diff for the code of ClientManager.java in commits under analysis.

8. CI tool

The main goal of the proposed continuous integration tool is to integrate application, tests and Python script that have been already described in detail. However, the automation server used in the thesis additionally enhances the capabilities of the aforementioned project artifacts. Not only is Jenkins capable of responding automatically to events to a github repository, it can also deploy the code to an another branch or repository if build finishes successfully.

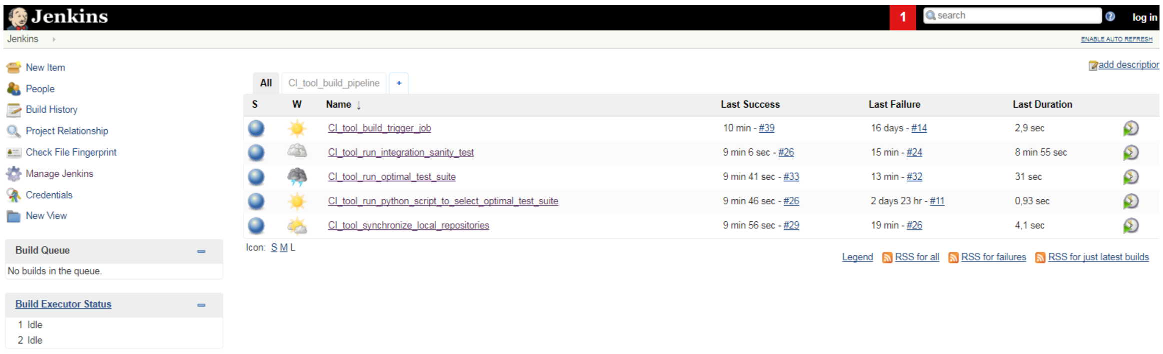
8.1. Jenkins build jobs

Build jobs are the essential feature of the Jenkins build process. A Jenkins build job can be viewed as a particular task or group of tasks in the build process. What makes Jenkins such a wonderful CI/CD tool lies in the fact that there are many available options to create a build job in many different ways. It makes it possible to set up a tailor-made job that fits for the purpose of any CI process requirement.

There are five jobs in Jenkins defined. These jobs are a series of related tasks, which are going to be executed sequentially. The entire build process is defined in CI_tool_build_pipeline that comprises the following steps:

- CI_tool_build_trigger_job – grants the Jenkins server access to the github repository. If a push event is detected, execution of the pipeline will start.
- CI_tool_synchronize_local_repositories – an analysis of changes in the code is performed locally to accelerate generation of the git diff log file and execution of the Python script. In order to carry out the analysis on local repositories, these repositories are synchronized to include all commits that have been pushed to the remote repository. This job is triggered if CI_tool_build_trigger_job finishes successfully.
- CI_tool_run_Python_script_to_select_optimal_test_suite – execution of the Python script for an optimal test suite selection. The detailed description of the optimal test suite selector is available in chapter 7. This job is triggered if CI_tool_synchronize_local_repositories finishes successfully.
- CI_tool_run_optimal_test_suite – execution of an optimal test suite that has been selected in the previous step. This job is triggered if CI_tool_run_Python_script_to_select_optimal_test_suite finishes successfully.

- CI_tool_run_integration_sanity_test – execution of the sanity check. This job is triggered if CI_tool_run_Python_script_to_select_optimal_test_suite finishes successfully regardless of the result of CI_tool_run_optimal_test_suite.



Jenkins | 1 | search | log in

Jenkins › ENABLE AUTO REFRESH

- New Item
- People
- Build History
- Project Relationship
- Check File Fingerprint
- Manage Jenkins
- Credentials
- New View

Build Queue
No builds in the queue.

Build Executor Status
1 Idle
2 Idle

add description

All | CI_tool_build_pipeline | +

S	W	Name ↓	Last Success	Last Failure	Last Duration	
●	☀	CI_tool_build_trigger_job	10 min - #39	16 days - #14	2,9 sec	
●	☁	CI_tool_run_integration_sanity_test	9 min 6 sec - #26	15 min - #24	8 min 55 sec	
●	⛈	CI_tool_run_optimal_test_suite	9 min 41 sec - #33	13 min - #32	31 sec	
●	☀	CI_tool_run_python_script_to_select_optimal_test_suite	9 min 46 sec - #26	2 days 23 hr - #11	0,93 sec	
●	⛅	CI_tool_synchronize_local_repositories	9 min 56 sec - #29	19 min - #26	4,1 sec	

Icon: S M L

Legend | RSS for all | RSS for failures | RSS for just latest builds

Fig. 8.1. Jenkins view containing all Jenkins jobs that define the CI tool build process.

If both CI_tool_run_optimal_test_suite and CI_tool_run_integration_sanity_test finish successfully, there is a post-build action defined in the last job that merges the changes introduced in the commit that had triggered the Jenkins pipeline execution to the production branch on the github remote repository.

8.2. Execution of tests in Jenkins

The Python script selects an optimal test suite and updates the POM file with a list of test cases to be executed. In the next step these test cases shall be executed. However, with a view to executing only the chosen test cases, there is a need to use some features of Maven. Thus, pom.xml includes additional configuration options. The Python script updates these settings and creates a new POM file referred as updated_pom.xml. The Jenkins job that runs an optimal test suite is configured to use this POM file instead of the default one. Basically, in the last two jobs Jenkins specifies the goal of a Maven build and acts as a caller of pure Maven commands that are interpreted by the Maven plugin.

As depicted in figure 8.2. Java project of the TCP server/client application is on line with the default structure for a Java project in Maven. Generally, it is extremely beneficial to follow the default setup for every single feature defined in Maven. This approach leads to a minimal amount of configuration and easy syntax of Maven commands.

The presented below use of the standard directory layout in a way that allows both unit tests and integration tests execution in a Maven build was proposed by Petri Kainulainen and is available on his web page [12].

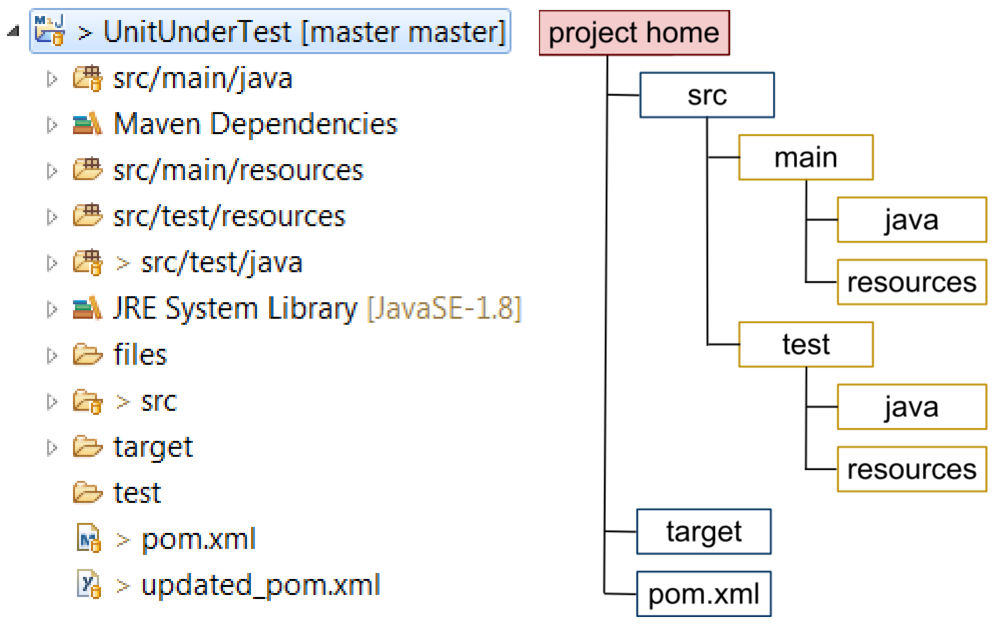


Fig. 8.2. Unit under test project tree in comparison to the default structure for a Java project in Maven [11].

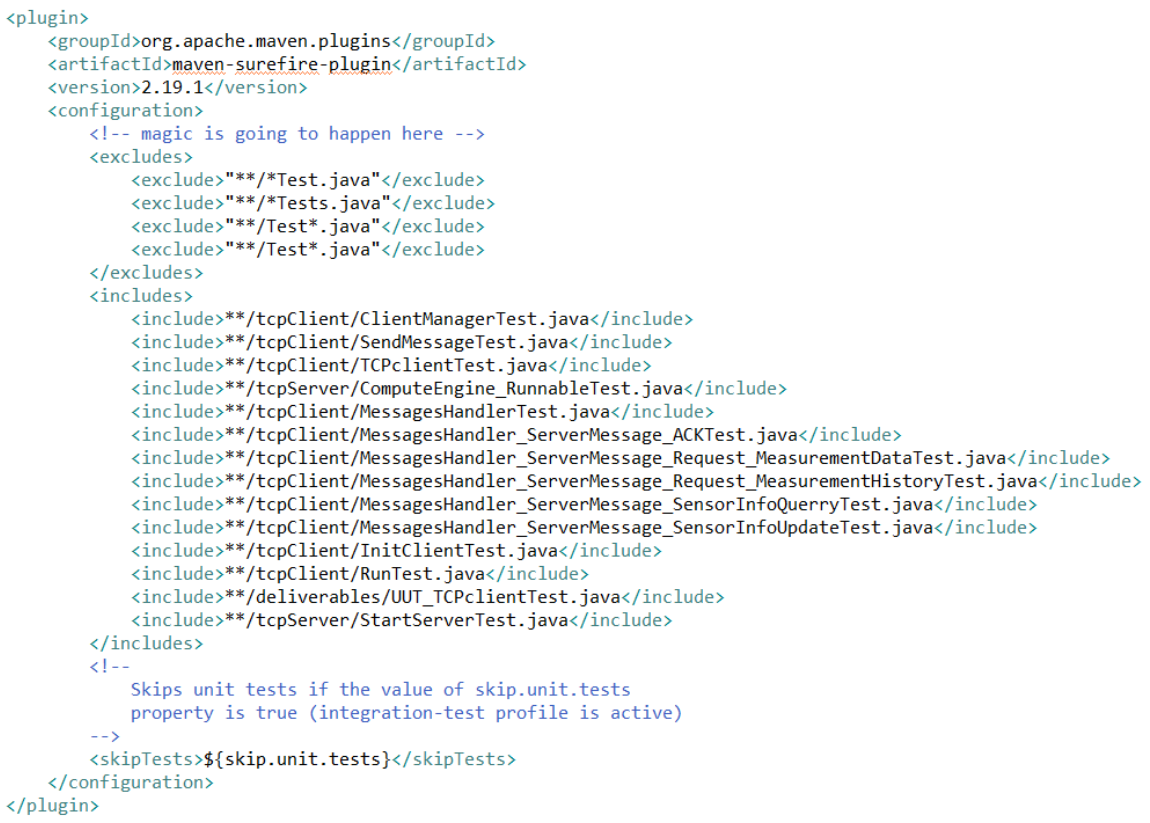
The first step to incorporate both types of testing aims at creating Maven profiles for unit and integration tests. The dev profile is used in the development environment, and this profile is executed when the active profile is not specified. The integration-test profile is used for running the integration tests. Respectively, the dev profile configures the name of the directory that contains the properties file which contains the configuration used in the development environment, whereas the integration-test profile configures the name of the directory with the properties file used by the integration tests. These profiles added to pom.xml are presented below.

```xml
<profiles>
    <profile>
        <id>dev</id>
        <activation>
            <activeByDefault>true</activeByDefault>
        </activation>
        <properties>
            <!--
                Only unit tests are run when the dev profile is active
            -->
            <skip.integration.tests>true</skip.integration.tests>
            <skip.unit.tests>false</skip.unit.tests>
            <packaging.type>pom</packaging.type>
        </properties>
    </profile>
    <profile>
        <id>integration-test</id>
        <properties>
            <!--
                Only integration tests are run when the integration-test profile is active
            -->
            <skip.integration.tests>false</skip.integration.tests>
            <skip.unit.tests>true</skip.unit.tests>
            <packaging.type>jar</packaging.type>
        </properties>
    </profile>
</profiles>
```

Fig. 8.3. Definition of profiles in pom.xml.

The Maven Surefire Plugin is used to run unit tests, whereas the Maven Failsafe Plugin to run integration tests. There is no need to add extra source and resource directories to the Maven build by using the Build Helper Maven Plugin since the default directory structure was applied in the project. The unit tests are differentiated from the integration tests by patterns that are included in the name of tests cases. By default the Maven Surefire Plugin will include the test cases matching one of the following patterns: <include>**/Test*.java</include>, <include>**/*Test.java</include>, <include>**/* Tests.java </include>, <include>**/*TestCase.java</include>, while the Maven Failsafe Plugin will include the test cases matching one of the following patterns: <include>**/IT*.java</include>, <include>**/*IT.java</include>,<include>**/*ITCase .java</include>.

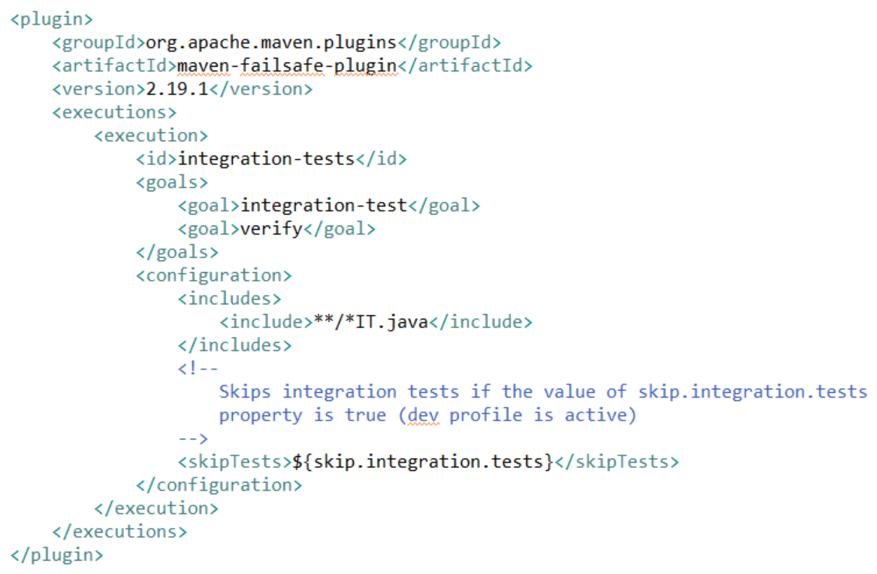
CI_tool_run_optimal_test_suite Jenkins job runs the test cases added to an optimal test suite by executing a Maven goal in Jenkins defined as follows: mvn clean test -P dev. In order to include only those test cases that have been selected by the Python script, the configuration section of the Maven Surefire Plugin in updated_pom.xml file is defined as shown in figure 8.4. The default namespace pattern for unit tests is deactivated by excluding the default patterns. The test cases submitted to execution are included by adding their exact names to updated _pom.xml file.



```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>2.19.1</version>
    <configuration>
        <!-- magic is going to happen here -->
        <excludes>
            <exclude>"**/*Test.java"</exclude>
            <exclude>"**/*Tests.java"</exclude>
            <exclude>"**/Test*.java"</exclude>
            <exclude>"**/Test*.java"</exclude>
        </excludes>
        <includes>
            <include>**/tcpClient/ClientManagerTest.java</include>
            <include>**/tcpClient/SendMessageTest.java</include>
            <include>**/tcpClient/TCPclientTest.java</include>
            <include>**/tcpServer/ComputeEngine_RunnableTest.java</include>
            <include>**/tcpClient/MessagesHandlerTest.java</include>
            <include>**/tcpClient/MessagesHandler_ServerMessage_ACKTest.java</include>
            <include>**/tcpClient/MessagesHandler_ServerMessage_Request_MeasurementDataTest.java</include>
            <include>**/tcpClient/MessagesHandler_ServerMessage_Request_MeasurementHistoryTest.java</include>
            <include>**/tcpClient/MessagesHandler_ServerMessage_SensorInfoQuerryTest.java</include>
            <include>**/tcpClient/MessagesHandler_ServerMessage_SensorInfoUpdateTest.java</include>
            <include>**/tcpClient/InitClientTest.java</include>
            <include>**/tcpClient/RunTest.java</include>
            <include>**/deliverables/UUT_TCPclientTest.java</include>
            <include>**/tcpServer/StartServerTest.java</include>
        </includes>
        <!--
            Skips unit tests if the value of skip.unit.tests
            property is true (integration-test profile is active)
        -->
        <skipTests>${skip.unit.tests}</skipTests>
    </configuration>
</plugin>
```

Fig. 8.4. Configuration of Maven Surefire Plugin in updated_pom.xml to run an optimal test suite.

CI_tool_run_integration_sanity_test Jenkins job runs the integration sanity check by executing a Maven goal in Jenkins defined as follows: mvn clean verify -P integration-test. There is only one integration test that is referred as ApplicationSanityCheckIT. Its name is consistent with the pattern defined in the Maven Failsafe Plugin, hence the job uses the default pom.xml file. The configuration section of the Failsafe Plugin is presented in figure 8.5.



```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-failsafe-plugin</artifactId>
    <version>2.19.1</version>
    <executions>
        <execution>
            <id>integration-tests</id>
            <goals>
                <goal>integration-test</goal>
                <goal>verify</goal>
            </goals>
            <configuration>
                <includes>
                    <include>**/*IT.java</include>
                </includes>
                <!--
                    Skips integration tests if the value of skip.integration.tests
                    property is true (dev profile is active)
                -->
                <skipTests>${skip.integration.tests}</skipTests>
            </configuration>
        </execution>
    </executions>
</plugin>
```

Fig. 8.5. Configuration of Maven Failsafe Plugin in pom.xml to run integration sanity check.

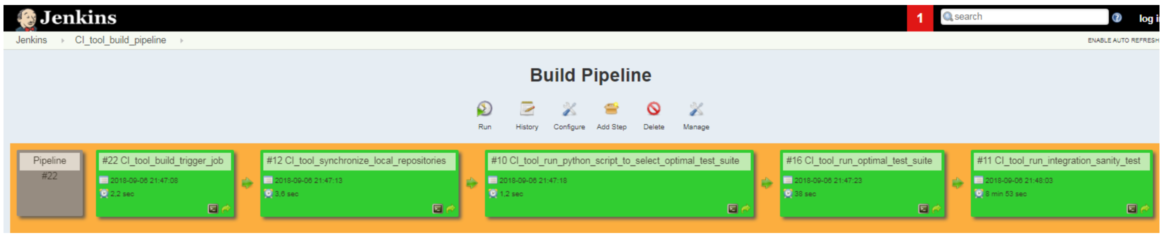
As defined in the Apache Maven Project documentation [13], the maven goal for unit tests execution is called “test”, whereas the maven goal for integration tests execution is called “verify”. The rationale for such approach could be found in the Maven default build lifecycle that comprises of the following steps: validate, compile, test, package, verify, install, deploy. These lifecycle phases are executed sequentially.

A unit test requires the code to be compiled, but not packaged, hence in figure 8.4. the packaging type parameter for the dev profile is set to pom. Test goal means to test the compiled source code using a suitable unit testing framework. The integration test phase needs to be preceded by packaging that takes the compiled code and packages it in its distributable format, e.g. JAR as shown in figure 8.3. Verify goal means to run any checks on results of integration tests to ensure quality criteria are met [13].

8.3. CI tool successful run for stable software

In this paragraph, a successful execution of CI_tool_build_pipeline is going to be illustrated. The commit that has triggered CI_tool_build_trigger_job can be uniquely identified by its commit ID: 9ab177a21a1acec9a08114cd410e0ba0aa536004. Its commit message is: “06-09-2018 CI tool official dry run”.

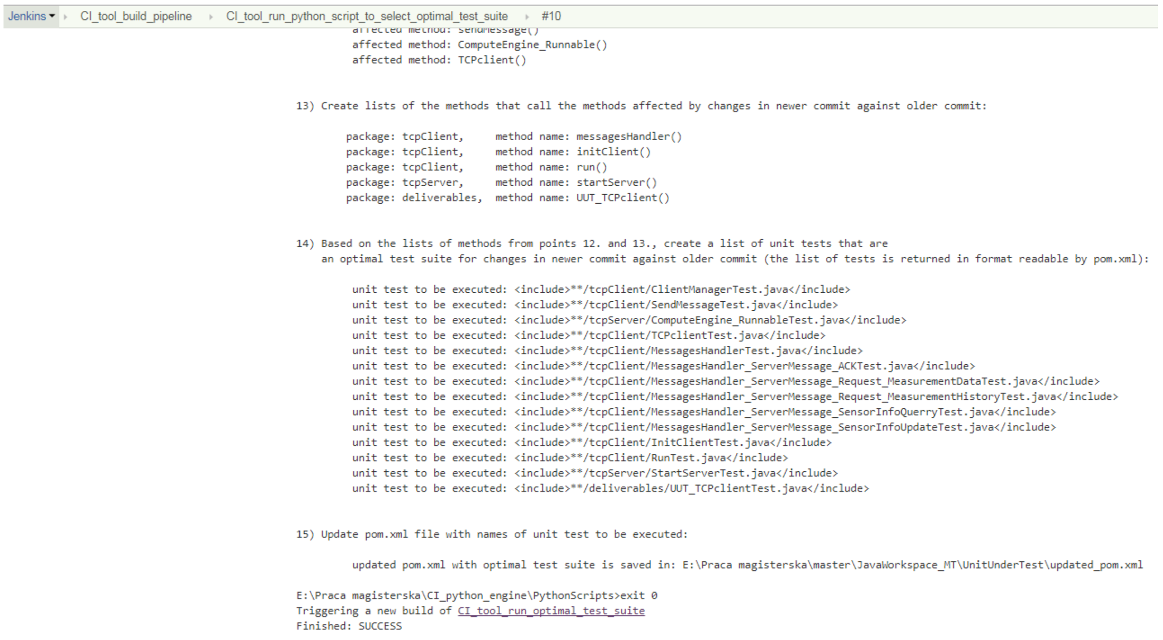
Basically, the changes made to the code by the commit under analysis have been already discussed in chapter 7. The changes are highlighted in figures 7.8., 7.9. and 7.10., while an optimal test suite for these changes is presented in figure 7.7. The updated pom.xml file that have been used in the CI_tool_run_optimal_test_suite Jenkins job for project build configuration is shown in figure 8.4.



```
Pipeline #22
#22 CI_tool_build_trigger_job                              2018-09-06 21:47:08   2,2 sec
#12 CI_tool_synchronize_local_repositories                 2018-09-06 21:47:13   3,6 sec
#10 CI_tool_run_python_script_to_select_optimal_test_suite 2018-09-06 21:47:18   1,2 sec
#16 CI_tool_run_optimal_test_suite                         2018-09-06 21:47:23   38 sec
#11 CI_tool_run_integration_sanity_test                    2018-09-06 21:48:03   8 min 53 sec
```

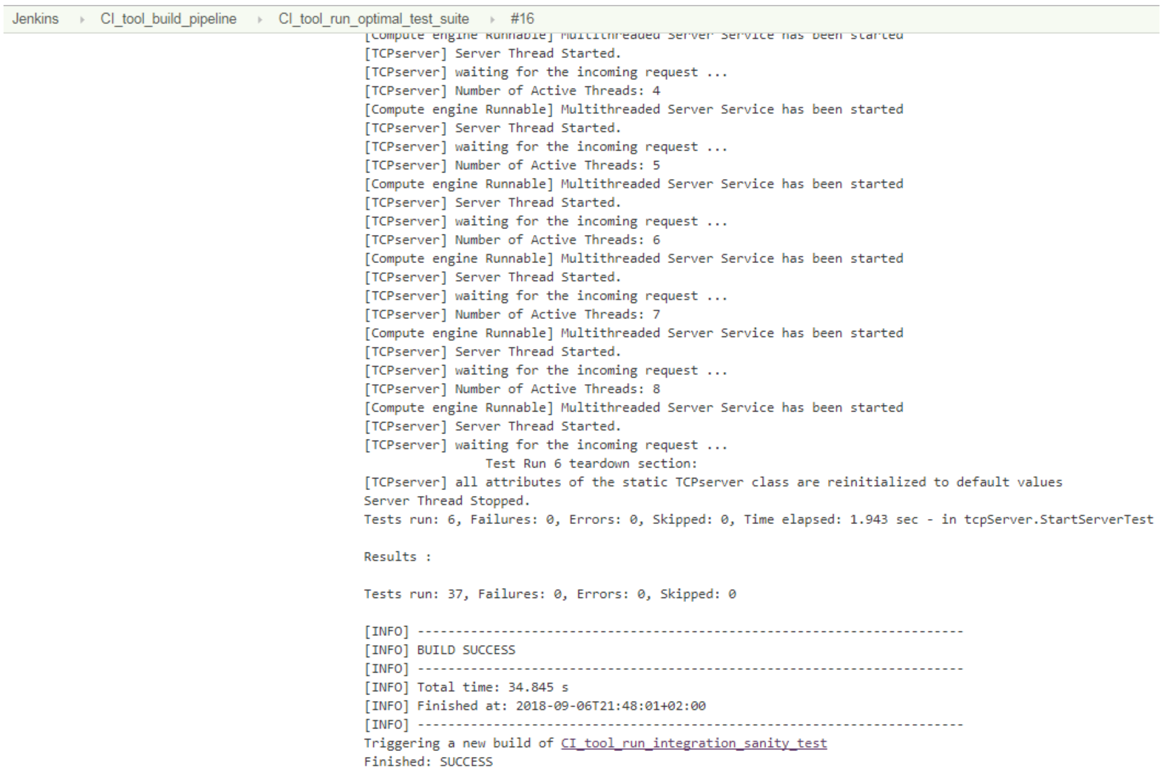
Fig. 8.6. Successful execution of a Jenkins pipeline: #22 CI_tool_build_pipeline.

In figure 8.6. there is GUI of the Jenkins build pipeline view. Jenkins web interface provides details on each of the jobs that have been executed during CI_tool_build_pipeline. The jobs can be uniquely identified by its number. In the below figures there are snippets from console output that contains the results of execution of the following Jenkins jobs: #10 CI_tool_run_Python_script_to_select_optimal_test_suite, #16 CI_tool_run_optimal_test_suite, #11 CI_tool_run_integration_sanity_test.



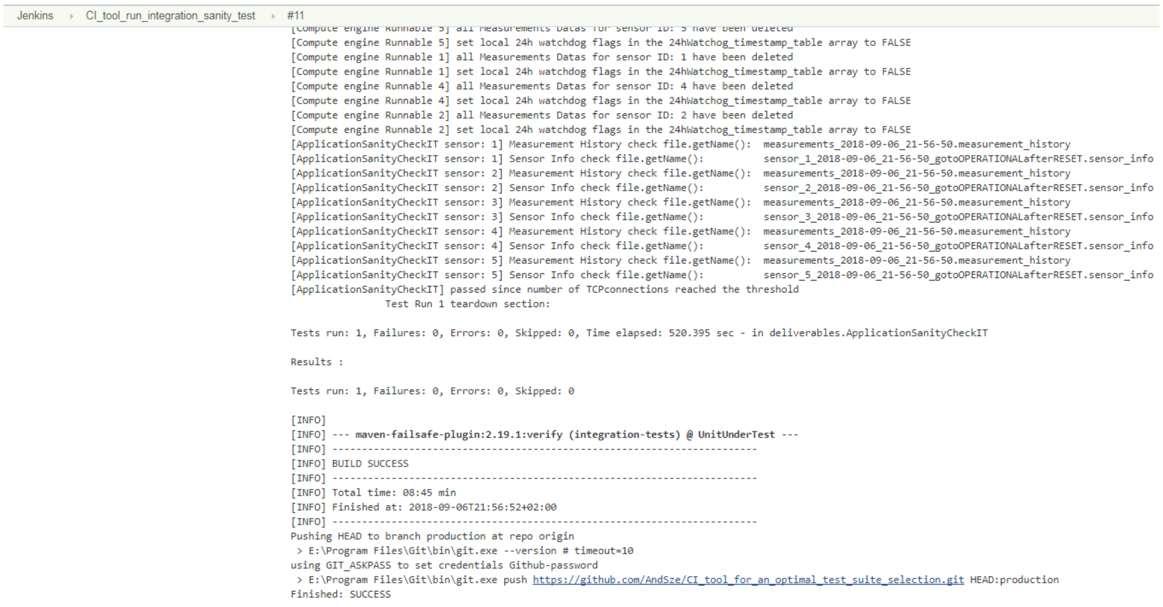
```
        affected method: sendMessage()
        affected method: ComputeEngine_Runnable()
        affected method: TCPclient()


13) Create lists of the methods that call the methods affected by changes in newer commit against older commit:

       package: tcpClient,     method name: messagesHandler()
       package: tcpClient,     method name: initClient()
       package: tcpClient,     method name: run()
       package: tcpServer,     method name: startServer()
       package: deliverables,  method name: UUT_TCPclient()


14) Based on the lists of methods from points 12. and 13., create a list of unit tests that are
    an optimal test suite for changes in newer commit against older commit (the list of tests is returned in format readable by pom.xml):

        unit test to be executed: <include>**/tcpClient/ClientManagerTest.java</include>
        unit test to be executed: <include>**/tcpClient/SendMessageTest.java</include>
        unit test to be executed: <include>**/tcpServer/ComputeEngine_RunnableTest.java</include>
        unit test to be executed: <include>**/tcpClient/TCPclientTest.java</include>
        unit test to be executed: <include>**/tcpClient/MessagesHandlerTest.java</include>
        unit test to be executed: <include>**/tcpClient/MessagesHandler_ServerMessage_ACKTest.java</include>
        unit test to be executed: <include>**/tcpClient/MessagesHandler_ServerMessage_Request_MeasurementDataTest.java</include>
        unit test to be executed: <include>**/tcpClient/MessagesHandler_ServerMessage_Request_MeasurementHistoryTest.java</include>
        unit test to be executed: <include>**/tcpClient/MessagesHandler_ServerMessage_SensorInfoQuerryTest.java</include>
        unit test to be executed: <include>**/tcpClient/MessagesHandler_ServerMessage_SensorInfoUpdateTest.java</include>
        unit test to be executed: <include>**/tcpClient/InitClientTest.java</include>
        unit test to be executed: <include>**/tcpClient/RunTest.java</include>
        unit test to be executed: <include>**/tcpServer/StartServerTest.java</include>
        unit test to be executed: <include>**/deliverables/UUT_TCPclientTest.java</include>


15) Update pom.xml file with names of unit test to be executed:

        updated pom.xml with optimal test suite is saved in: E:\Praca magisterska\master\JavaWorkspace_MT\UnitUnderTest\updated_pom.xml

E:\Praca magisterska\CI_python_engine\PythonScripts>exit 0
Triggering a new build of CI_tool_run_optimal_test_suite
Finished: SUCCESS
```

Fig. 8.7. Successful execution of Jenkins job: #10 CI_tool_run_Python_script_to_select_optimal_test_suite.



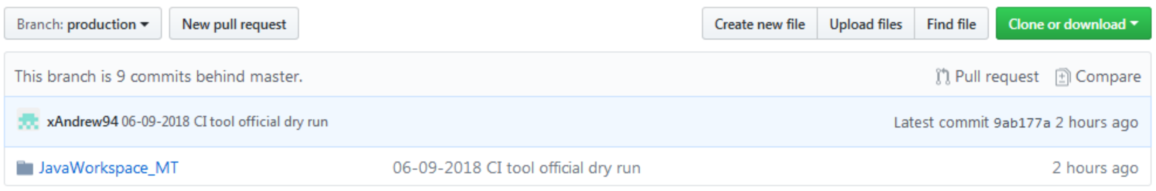
```
[Compute engine Runnable] Multithreaded Server Service has been started
[TCPserver] Server Thread Started.
[TCPserver] waiting for the incoming request ...
[TCPserver] Number of Active Threads: 4
[Compute engine Runnable] Multithreaded Server Service has been started
[TCPserver] Server Thread Started.
[TCPserver] waiting for the incoming request ...
[TCPserver] Number of Active Threads: 5
[Compute engine Runnable] Multithreaded Server Service has been started
[TCPserver] Server Thread Started.
[TCPserver] waiting for the incoming request ...
[TCPserver] Number of Active Threads: 6
[Compute engine Runnable] Multithreaded Server Service has been started
[TCPserver] Server Thread Started.
[TCPserver] waiting for the incoming request ...
[TCPserver] Number of Active Threads: 7
[Compute engine Runnable] Multithreaded Server Service has been started
[TCPserver] Server Thread Started.
[TCPserver] waiting for the incoming request ...
[TCPserver] Number of Active Threads: 8
[Compute engine Runnable] Multithreaded Server Service has been started
[TCPserver] Server Thread Started.
[TCPserver] waiting for the incoming request ...
                Test Run 6 teardown section:
[TCPserver] all attributes of the static TCPserver class are reinitialized to default values
Server Thread Stopped.
Tests run: 6, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 1.943 sec - in tcpServer.StartServerTest

Results :

Tests run: 37, Failures: 0, Errors: 0, Skipped: 0

[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 34.845 s
[INFO] Finished at: 2018-09-06T21:48:01+02:00
[INFO] ------------------------------------------------------------------------
Triggering a new build of CI_tool_run_integration_sanity_test
Finished: SUCCESS
```

Fig. 8.8. Successful execution of a Jenkins job: #16 CI_tool_run_optimal_test_suite.



```
[Compute engine Runnable 5] all Measurements Datas for sensor ID: 5 have been deleted
[Compute engine Runnable 5] set local 24h watchdog flags in the 24hWatchog_timestamp_table array to FALSE
[Compute engine Runnable 1] all Measurements Datas for sensor ID: 1 have been deleted
[Compute engine Runnable 1] set local 24h watchdog flags in the 24hWatchog_timestamp_table array to FALSE
[Compute engine Runnable 4] all Measurements Datas for sensor ID: 4 have been deleted
[Compute engine Runnable 4] set local 24h watchdog flags in the 24hWatchog_timestamp_table array to FALSE
[Compute engine Runnable 2] all Measurements Datas for sensor ID: 2 have been deleted
[Compute engine Runnable 2] set local 24h watchdog flags in the 24hWatchog_timestamp_table array to FALSE
[ApplicationSanityCheckIT sensor: 1] Measurement History check file.getName():  measurements_2018-09-06_21-56-50.measurement_history
[ApplicationSanityCheckIT sensor: 1] Sensor Info check file.getName():          sensor_1_2018-09-06_21-56-50_gotoOPERATIONALafterRESET.sensor_info
[ApplicationSanityCheckIT sensor: 2] Measurement History check file.getName():  measurements_2018-09-06_21-56-50.measurement_history
[ApplicationSanityCheckIT sensor: 2] Sensor Info check file.getName():          sensor_2_2018-09-06_21-56-50_gotoOPERATIONALafterRESET.sensor_info
[ApplicationSanityCheckIT sensor: 3] Measurement History check file.getName():  measurements_2018-09-06_21-56-50.measurement_history
[ApplicationSanityCheckIT sensor: 3] Sensor Info check file.getName():          sensor_3_2018-09-06_21-56-50_gotoOPERATIONALafterRESET.sensor_info
[ApplicationSanityCheckIT sensor: 4] Measurement History check file.getName():  measurements_2018-09-06_21-56-50.measurement_history
[ApplicationSanityCheckIT sensor: 4] Sensor Info check file.getName():          sensor_4_2018-09-06_21-56-50_gotoOPERATIONALafterRESET.sensor_info
[ApplicationSanityCheckIT sensor: 5] Measurement History check file.getName():  measurements_2018-09-06_21-56-50.measurement_history
[ApplicationSanityCheckIT sensor: 5] Sensor Info check file.getName():          sensor_5_2018-09-06_21-56-50_gotoOPERATIONALafterRESET.sensor_info
[ApplicationSanityCheckIT] passed since number of TCPconnections reached the threshold
                Test Run 1 teardown section:

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 520.395 sec - in deliverables.ApplicationSanityCheckIT

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

[INFO]
[INFO] --- maven-failsafe-plugin:2.19.1:verify (integration-tests) @ UnitUnderTest ---
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 08:45 min
[INFO] Finished at: 2018-09-06T21:56:52+02:00
[INFO] ------------------------------------------------------------------------
Pushing HEAD to branch production at repo origin
 > E:\Program Files\Git\bin\git.exe --version # timeout=10
using GIT_ASKPASS to set credentials Github-password
 > E:\Program Files\Git\bin\git.exe push https://github.com/AndSze/CI_tool_for_an_optimal_test_suite_selection.git HEAD:production
Finished: SUCCESS
```

Fig. 8.9. Successful execution of a Jenkins job: #11 CI_tool_run_integration_sanity_test.



This branch is 9 commits behind master.

xAndrew94 06-09-2018 CI tool official dry run — Latest commit 9ab177a 2 hours ago

JavaWorkspace_MT — 06-09-2018 CI tool official dry run — 2 hours ago

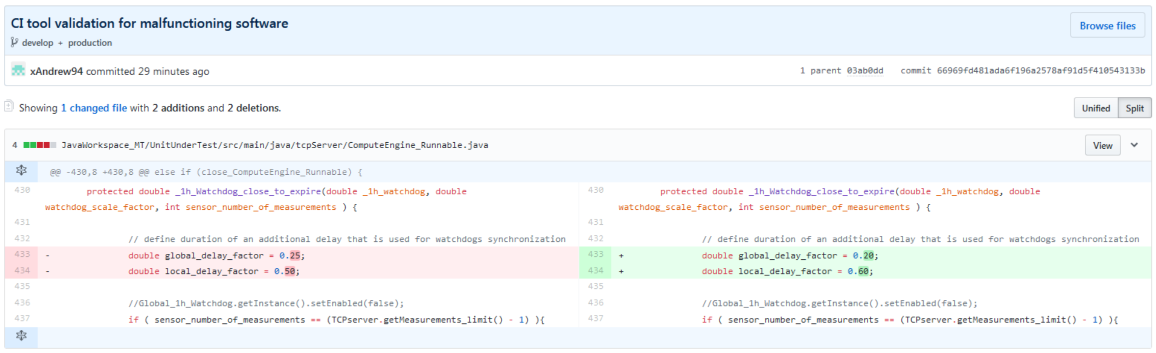
Fig. 8.10. Code merged to the production branch after successful execution of Jenkins pipeline: #22 CI_tool_build_pipeline.

It can be observed that in figure 8.7.. there are exactly the same results as in figure 7.7. Similarly, the results in figure 8.9. are identical to the ones in figure 6.7. The Python script has created a list of 14 test cases that verify the code that could have been affected by the changes. These 14 test cases have 37 test runs in overall. Figure 8.8. contains console output of the Jenkins Job that executed an optimal test suite comprising of the selected 37 test runs.

As mentioned in paragraph 8.1., there is a post build action defined in CI_tool_build_pipeline. This post build action would result in merging the code, which have been successfully built and tested by the process defined in Jenkins, to the production branch. Figure 8.10. shows the production branch on the github server. The last commit merged to the production branch is the commit containing the code changes that have triggered #22 CI_tool_build_trigger_job and resulted in successful execution of #22 CI_tool_build_pipeline.

8.4. CI tool unsuccessful run for malfunctioning software

The purpose of including an unsuccessful execution of the CI tool is to visualize that an optimal test suite is able to detect as many defects in the code as the entire test suite detects. In this paragraph, the commit to be analyzed has the following commit ID: 66969fd481ada6f196a2578af91d5f410543133b. Its commit message is: "CI tool validation for malfunctioning software". The changes that have introduced some bugs to the code are presented in figure 8.11.


CI tool validation for malfunctioning software
develop + production

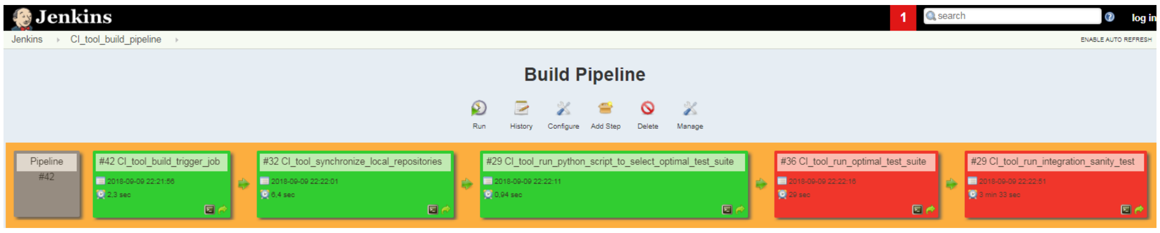
xAndrew94 committed 29 minutes ago   1 parent 03ab0dd   commit 66969fd481ada6f196a2578af91d5f410543133b

Showing 1 changed file with 2 additions and 2 deletions.

```
4  JavaWorkspace_MT/UnitUnderTest/src/main/java/tcpServer/ComputeEngine_Runnable.java
@@ -430,8 +430,8 @@ else if (close_ComputeEngine_Runnable) {
        protected double _1h_Watchdog_close_to_expire(double _1h_watchdog, double
watchdog_scale_factor, int sensor_number_of_measurements ) {

                // define duration of an additional delay that is used for watchdogs synchronization
-               double global_delay_factor = 0.25;
-               double local_delay_factor = 0.50;

                //Global_1h_Watchdog.getInstance().setEnabled(false);
                if ( sensor_number_of_measurements == (TCPserver.getMeasurements_limit() - 1) ){
```

```
        protected double _1h_Watchdog_close_to_expire(double _1h_watchdog, double
watchdog_scale_factor, int sensor_number_of_measurements ) {

                // define duration of an additional delay that is used for watchdogs synchronization
+               double global_delay_factor = 0.20;
+               double local_delay_factor = 0.60;

                //Global_1h_Watchdog.getInstance().setEnabled(false);
                if ( sensor_number_of_measurements == (TCPserver.getMeasurements_limit() - 1) ){
```

Fig. 8.11. Github side-by-side diff for the malfunctioning code in the commit under analysis.



Pipeline #42

#42 CI_tool_build_trigger_job
2018-09-09 22:21:56
2,3 sec

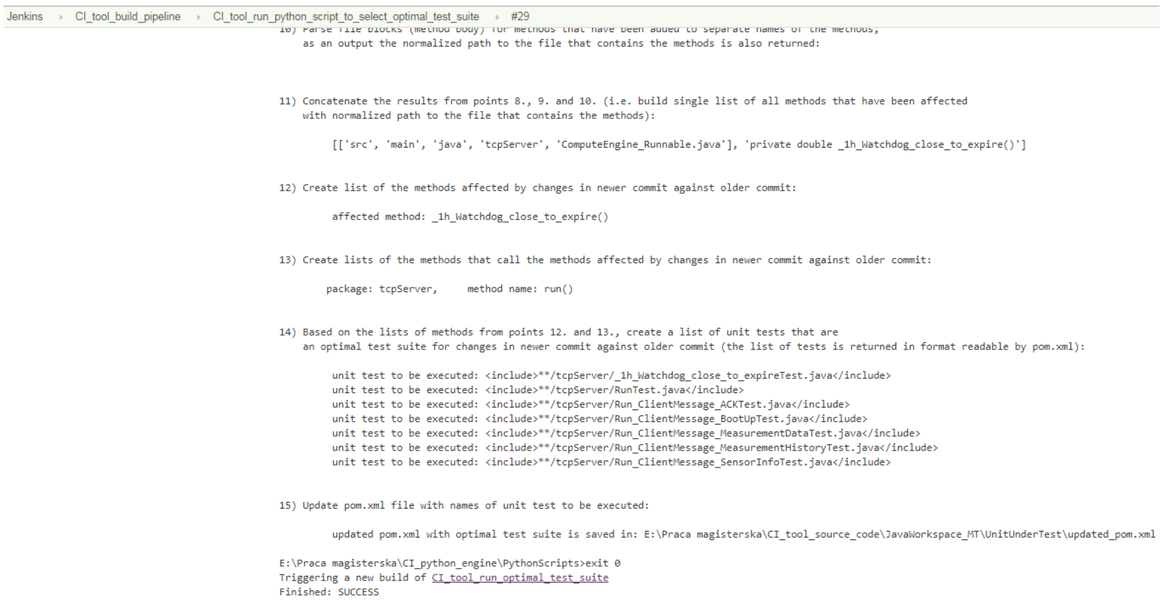
#32 CI_tool_synchronize_local_repositories
2018-09-09 22:22:01
6,4 sec

#29 CI_tool_run_python_script_to_select_optimal_test_suite
2018-09-09 22:22:11
0,94 sec

#36 CI_tool_run_optimal_test_suite
2018-09-09 22:22:16
29 sec

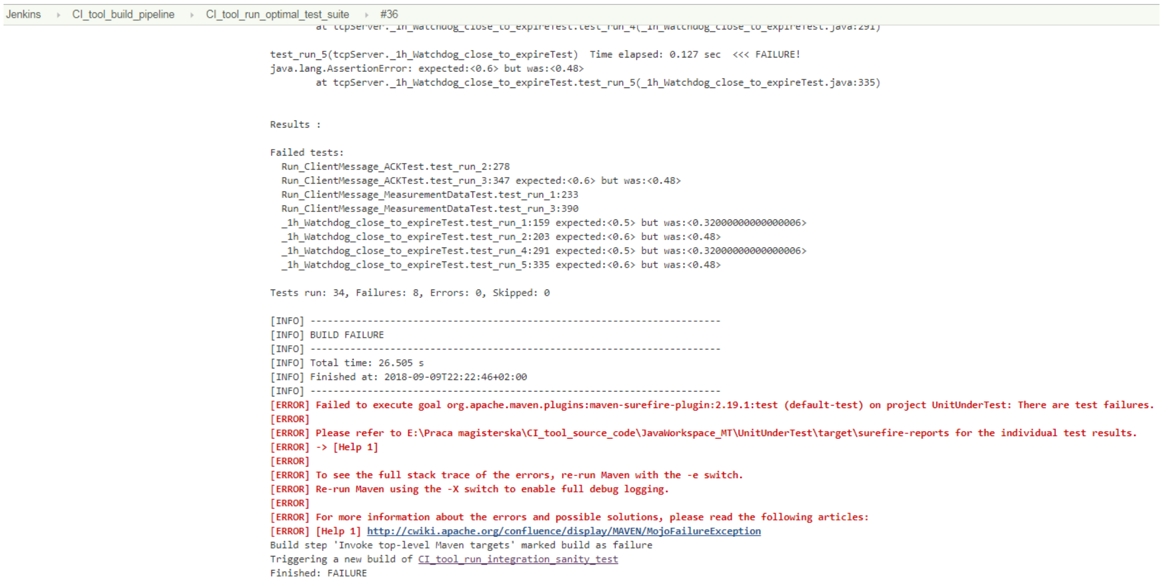
#29 CI_tool_run_integration_sanity_test
2018-09-09 22:22:51
3 min 33 sec

Fig. 8.12. Unsuccessful execution of a Jenkins pipeline: #42 CI_tool_build_pipeline.



```
Jenkins  ›  CI_tool_build_pipeline  ›  CI_tool_run_python_script_to_select_optimal_test_suite  ›  #29
10) Parse file blocks (method body) for methods that have been added to separate names of the methods,
    as an output the normalized path to the file that contains the methods is also returned:



11) Concatenate the results from points 8., 9. and 10. (i.e. build single list of all methods that have been affected
    with normalized path to the file that contains the methods):

        [['src', 'main', 'java', 'tcpServer', 'ComputeEngine_Runnable.java'], 'private double _1h_Watchdog_close_to_expire()']


12) Create list of the methods affected by changes in newer commit against older commit:

        affected method: _1h_Watchdog_close_to_expire()


13) Create lists of the methods that call the methods affected by changes in newer commit against older commit:

       package: tcpServer,     method name: run()


14) Based on the lists of methods from points 12. and 13., create a list of unit tests that are
    an optimal test suite for changes in newer commit against older commit (the list of tests is returned in format readable by pom.xml):

        unit test to be executed: <include>**/tcpServer/_1h_Watchdog_close_to_expireTest.java</include>
        unit test to be executed: <include>**/tcpServer/RunTest.java</include>
        unit test to be executed: <include>**/tcpServer/Run_ClientMessage_ACKTest.java</include>
        unit test to be executed: <include>**/tcpServer/Run_ClientMessage_BootUpTest.java</include>
        unit test to be executed: <include>**/tcpServer/Run_ClientMessage_MeasurementDataTest.java</include>
        unit test to be executed: <include>**/tcpServer/Run_ClientMessage_MeasurementHistoryTest.java</include>
        unit test to be executed: <include>**/tcpServer/Run_ClientMessage_SensorInfoTest.java</include>


15) Update pom.xml file with names of unit test to be executed:

        updated pom.xml with optimal test suite is saved in: E:\Praca magisterska\CI_tool_source_code\JavaWorkspace_MT\UnitUnderTest\updated_pom.xml

E:\Praca magisterska\CI_python_engine\PythonScripts>exit 0
Triggering a new build of CI_tool_run_optimal_test_suite
Finished: SUCCESS
```

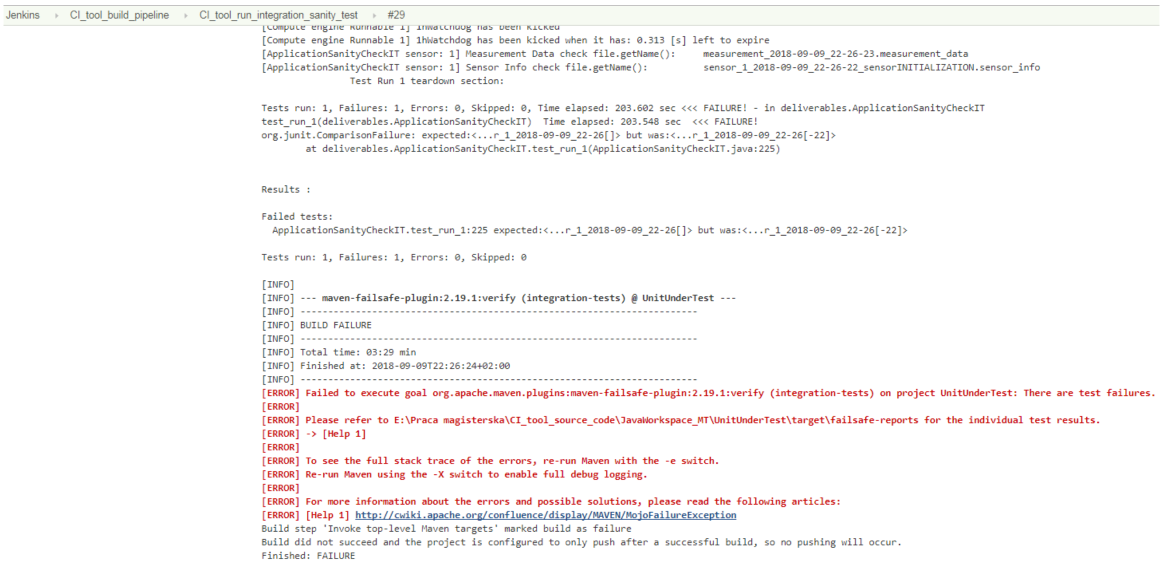
Fig. 8.13 An optimal test suite for malfunctioning software selected in a Jenkins job:
#29 CI_tool_run_Python_script_to_select_optimal_test_suite.



```
Jenkins  ›  CI_tool_build_pipeline  ›  CI_tool_run_optimal_test_suite  ›  #36
        at tcpServer._1h_Watchdog_close_to_expireTest.test_run_4(_1h_Watchdog_close_to_expireTest.java:291)

test_run_5(tcpServer._1h_Watchdog_close_to_expireTest)  Time elapsed: 0.127 sec  <<< FAILURE!
java.lang.AssertionError: expected:<0.6> but was:<0.48>
        at tcpServer._1h_Watchdog_close_to_expireTest.test_run_5(_1h_Watchdog_close_to_expireTest.java:335)


Results :

Failed tests:
  Run_ClientMessage_ACKTest.test_run_2:278
  Run_ClientMessage_ACKTest.test_run_3:347 expected:<0.6> but was:<0.48>
  Run_ClientMessage_MeasurementDataTest.test_run_1:233
  Run_ClientMessage_MeasurementDataTest.test_run_3:390
  _1h_Watchdog_close_to_expireTest.test_run_1:159 expected:<0.5> but was:<0.32000000000000006>
  _1h_Watchdog_close_to_expireTest.test_run_2:203 expected:<0.6> but was:<0.48>
  _1h_Watchdog_close_to_expireTest.test_run_4:291 expected:<0.5> but was:<0.32000000000000006>
  _1h_Watchdog_close_to_expireTest.test_run_5:335 expected:<0.6> but was:<0.48>

Tests run: 34, Failures: 8, Errors: 0, Skipped: 0

[INFO] ------------------------------------------------------------------------
[INFO] BUILD FAILURE
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 26.505 s
[INFO] Finished at: 2018-09-09T22:22:46+02:00
[INFO] ------------------------------------------------------------------------
[ERROR] Failed to execute goal org.apache.maven.plugins:maven-surefire-plugin:2.19.1:test (default-test) on project UnitUnderTest: There are test failures.
[ERROR]
[ERROR] Please refer to E:\Praca magisterska\CI_tool_source_code\JavaWorkspace_MT\UnitUnderTest\target\surefire-reports for the individual test results.
[ERROR] -> [Help 1]
[ERROR]
[ERROR] To see the full stack trace of the errors, re-run Maven with the -e switch.
[ERROR] Re-run Maven using the -X switch to enable full debug logging.
[ERROR]
[ERROR] For more information about the errors and possible solutions, please read the following articles:
[ERROR] [Help 1] http://cwiki.apache.org/confluence/display/MAVEN/MojoFailureException
Build step 'Invoke top-level Maven targets' marked build as failure
Triggering a new build of CI_tool_run_integration_sanity_test
Finished: FAILURE
```

Fig. 8.14. Unsuccessful execution of a Jenkins job for malfunctioning software:
#36 CI_tool_run_optimal_test_suite.

The Python script has created a list of 7 test cases that verify the code that could have been affected by the changes presented in figure 8.11. These 7 test cases have 34 test runs in overall. Figure 8.13. contains console output of the Jenkins Job that selected the test cases to be run. Figure 8.14. contains console output of the Jenkins Job that executes an optimal test suite comprising of the selected 34 test runs. 8 test runs out of 34 test runs submitted for execution have failed. It can be observed that 4 of the failed test runs directly verify the piece of code that has changed (_1h_Watchdog_close_to_expireTest). While the other 4 failed test runs are intended to verify different features of the application. However, the changes made to _1h_Watchdog_close_to_expire method caused some of these features to malfunction.



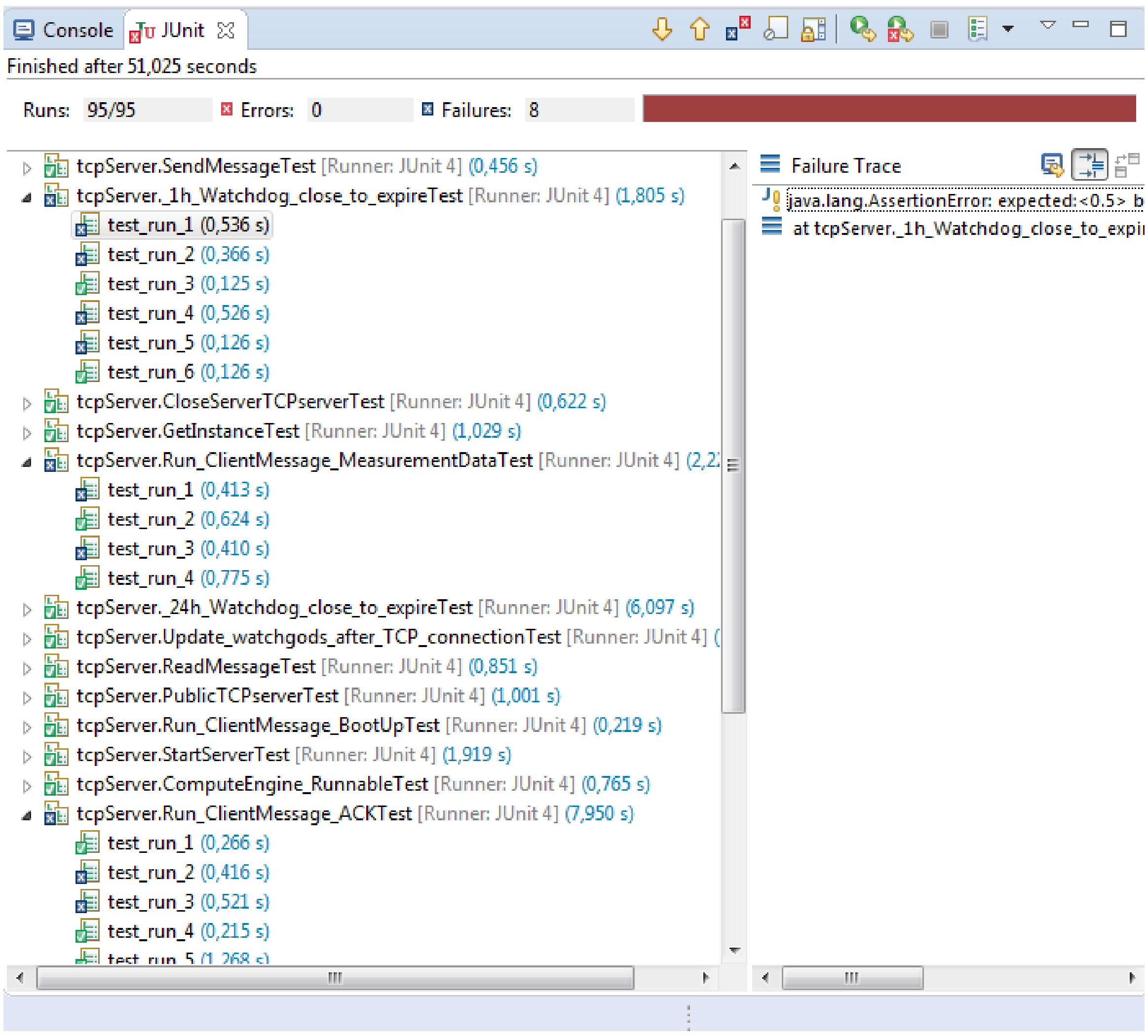
```
[Compute engine Runnable 1] 1hWatchdog has been kicked
[Compute engine Runnable 1] 1hWatchdog has been kicked when it has: 0.313 [s] left to expire
[ApplicationSanityCheckIT sensor: 1] Measurement Data check file.getName():     measurement_2018-09-09_22-26-23.measurement_data
[ApplicationSanityCheckIT sensor: 1] Sensor Info check file.getName():          sensor_1_2018-09-09_22-26-22_sensorINITIALIZATION.sensor_info
                Test Run 1 teardown section:

Tests run: 1, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 203.602 sec <<< FAILURE! - in deliverables.ApplicationSanityCheckIT
test_run_1(deliverables.ApplicationSanityCheckIT)  Time elapsed: 203.548 sec  <<< FAILURE!
org.junit.ComparisonFailure: expected:<...r_1_2018-09-09_22-26[]> but was:<...r_1_2018-09-09_22-26[-22]>
        at deliverables.ApplicationSanityCheckIT.test_run_1(ApplicationSanityCheckIT.java:225)


Results :

Failed tests:
  ApplicationSanityCheckIT.test_run_1:225 expected:<...r_1_2018-09-09_22-26[]> but was:<...r_1_2018-09-09_22-26[-22]>

Tests run: 1, Failures: 1, Errors: 0, Skipped: 0

[INFO]
[INFO] --- maven-failsafe-plugin:2.19.1:verify (integration-tests) @ UnitUnderTest ---
[INFO] ------------------------------------------------------------------------
[INFO] BUILD FAILURE
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 03:29 min
[INFO] Finished at: 2018-09-09T22:26:24+02:00
[INFO] ------------------------------------------------------------------------
[ERROR] Failed to execute goal org.apache.maven.plugins:maven-failsafe-plugin:2.19.1:verify (integration-tests) on project UnitUnderTest: There are test failures.
[ERROR]
[ERROR] Please refer to E:\Praca magisterska\CI_tool_source_code\JavaWorkspace_MT\UnitUnderTest\target\failsafe-reports for the individual test results.
[ERROR] -> [Help 1]
[ERROR]
[ERROR] To see the full stack trace of the errors, re-run Maven with the -e switch.
[ERROR] Re-run Maven using the -X switch to enable full debug logging.
[ERROR]
[ERROR] For more information about the errors and possible solutions, please read the following articles:
[ERROR] [Help 1] http://cwiki.apache.org/confluence/display/MAVEN/MojoFailureException
Build step 'Invoke top-level Maven targets' marked build as failure
Build did not succeed and the project is configured to only push after a successful build, so no pushing will occur.
Finished: FAILURE
```

Fig. 8.15. Unsuccessful execution of a Jenkins job for malfunctioning software:
#29 CI_tool_run_integration_sanity_test.

In figure 8.15. it is also shown that the application is not up and running due to the defects in its code. The application is quite robust, so it has been able to run for 3 minutes and 29 seconds, but finally one of the messages received by the TCP server was different than expected. In turn, this incorrect message was the cause for the unsuccessful executions of the integration sanity check ApplicationSanityCheckIT and then the Jenkins job CI_tool_run_integration_sanity_test, which calls this integration test.

It is high time to challenge the correctness of an optimal test suite that has been executed, and as a result, it has revealed 8 failures. There has been the entire test suite for tcpserver package executed in order to evaluate the selected test runs in terms of the possibility for finding every bug that has been introduced to the code by the changes in figure 8.11. The results of execution of those 95 test runs implemented for tcpserver package are presented in figure 8.16.

Results of the execution of the entire test suite create the opportunity for reviewing if an optimal test suite is fit for purpose by comparing the number of failures. The execution of 95 test runs revealed the same number of failures as the execution of a subset comprised of 34 test runs selected by the Python script. Hence, in conclusion, it can be said that the CI tool is able to find bugs in the code by executing a limited number of test cases, which have been previously added to a tailor-made test suite.



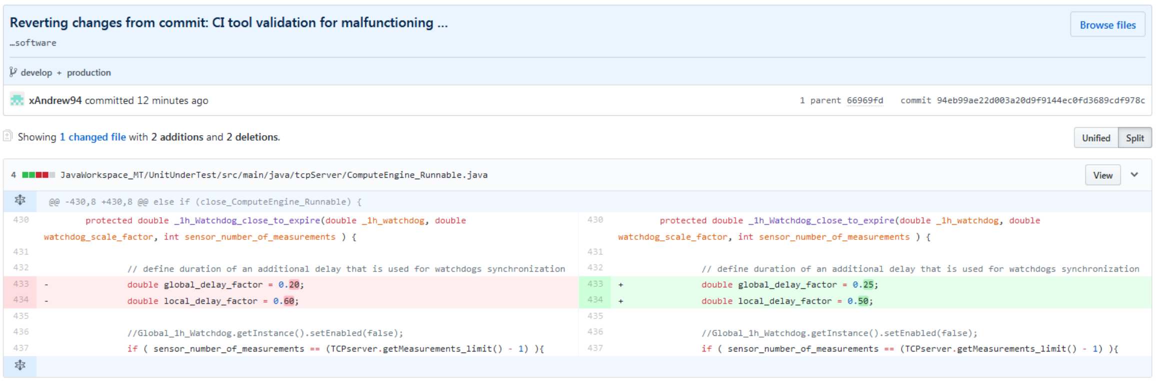
```
Finished after 51,025 seconds
Runs: 95/95    Errors: 0    Failures: 8

tcpServer.SendMessageTest [Runner: JUnit 4] (0,456 s)
tcpServer._1h_Watchdog_close_to_expireTest [Runner: JUnit 4] (1,805 s)
    test_run_1 (0,536 s)
    test_run_2 (0,366 s)
    test_run_3 (0,125 s)
    test_run_4 (0,526 s)
    test_run_5 (0,126 s)
    test_run_6 (0,126 s)
tcpServer.CloseServerTCPserverTest [Runner: JUnit 4] (0,622 s)
tcpServer.GetInstanceTest [Runner: JUnit 4] (1,029 s)
tcpServer.Run_ClientMessage_MeasurementDataTest [Runner: JUnit 4] (2,2
    test_run_1 (0,413 s)
    test_run_2 (0,624 s)
    test_run_3 (0,410 s)
    test_run_4 (0,775 s)
tcpServer._24h_Watchdog_close_to_expireTest [Runner: JUnit 4] (6,097 s)
tcpServer.Update_watchgods_after_TCP_connectionTest [Runner: JUnit 4] (
tcpServer.ReadMessageTest [Runner: JUnit 4] (0,851 s)
tcpServer.PublicTCPserverTest [Runner: JUnit 4] (1,001 s)
tcpServer.Run_ClientMessage_BootUpTest [Runner: JUnit 4] (0,219 s)
tcpServer.StartServerTest [Runner: JUnit 4] (1,919 s)
tcpServer.ComputeEngine_RunnableTest [Runner: JUnit 4] (0,765 s)
tcpServer.Run_ClientMessage_ACKTest [Runner: JUnit 4] (7,950 s)
    test_run_1 (0,266 s)
    test_run_2 (0,416 s)
    test_run_3 (0,521 s)
    test_run_4 (0,215 s)
    test_run_5 (1,268 s)

Failure Trace
java.lang.AssertionError: expected:<0.5> b
at tcpServer._1h_Watchdog_close_to_expi
```

Fig. 8.16. Results of execution of the entire test suite for tcpserver package.

8.5. CI tool successful run for restoring software to stable version

The purpose of including a successful execution of the Jenkins build pipeline for restoring software to the stable version is to give evidence of the CI tool stability and provide proof that there is no room for randomness in the results. In this paragraph, the commit to be analyzed has the following commit ID: 94eb99ae22d003a20d9f9144-ec0fd3689cdf978c. Its commit message is: “Reverting changes from commit: CI tool validation for malfunctioning software”. The changes that have fixed the previously introduced bugs are presented in figure 8.17.

In figure 8.18. there are two runs of CI_tool_build_pipeline. Each run can be identified by its number, respectively #42 and #43. The former CI tool execution is carried out on the buggy software, whereas the latter execution is performed on the software version with bug fixes.


Reverting changes from commit: CI tool validation for malfunctioning …
…software

develop + production

xAndrew94 committed 12 minutes ago   1 parent 66969fd   commit 94eb99ae22d003a20d9f9144ec0fd3689cdf978c

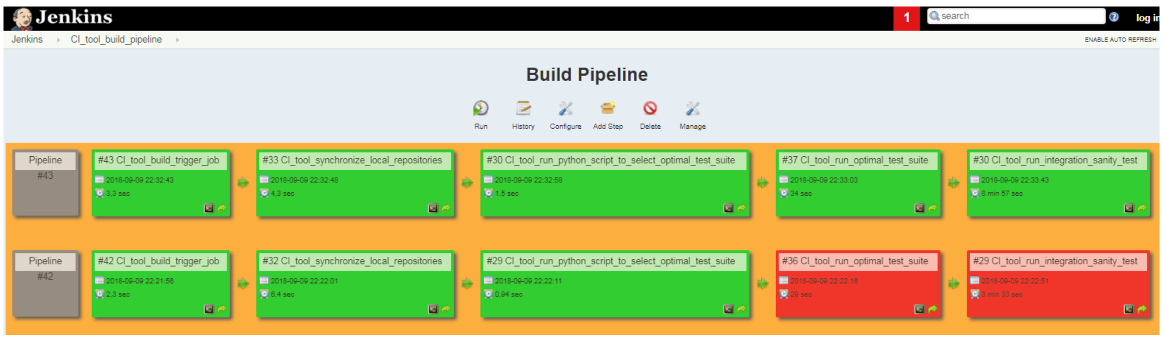
Showing 1 changed file with 2 additions and 2 deletions.

4 JavaWorkspace_MT/UnitUnderTest/src/main/java/tcpServer/ComputeEngine_Runnable.java

```
@@ -430,8 +430,8 @@ else if (close_ComputeEngine_Runnable) {
        protected double _1h_Watchdog_close_to_expire(double _1h_watchdog, double
watchdog_scale_factor, int sensor_number_of_measurements ) {

                // define duration of an additional delay that is used for watchdogs synchronization
-               double global_delay_factor = 0.20;
-               double local_delay_factor = 0.60;

                //Global_1h_Watchdog.getInstance().setEnabled(false);
                if ( sensor_number_of_measurements == (TCPserver.getMeasurements_limit() - 1) ){
```

```
        protected double _1h_Watchdog_close_to_expire(double _1h_watchdog, double
watchdog_scale_factor, int sensor_number_of_measurements ) {

                // define duration of an additional delay that is used for watchdogs synchronization
+               double global_delay_factor = 0.25;
+               double local_delay_factor = 0.50;

                //Global_1h_Watchdog.getInstance().setEnabled(false);
                if ( sensor_number_of_measurements == (TCPserver.getMeasurements_limit() - 1) ){
```

Fig. 8.17. Github side-by-side diff for the changes in the code that are supposed to fix the bugs.


Jenkins › CI_tool_build_pipeline ›

Build Pipeline

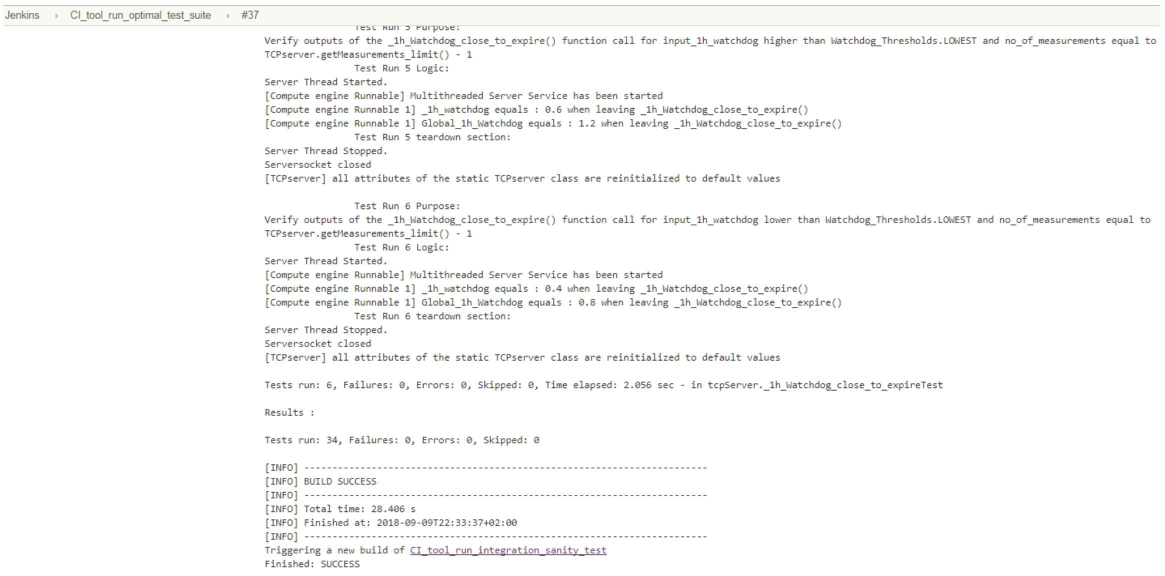
```
Pipeline #43 | #43 CI_tool_build_trigger_job 2018-09-09 22:32:43 3,3 sec | #33 CI_tool_synchronize_local_repositories 2018-09-09 22:32:48 4,3 sec | #30 CI_tool_run_python_script_to_select_optimal_test_suite 2018-09-09 22:32:58 1,5 sec | #37 CI_tool_run_optimal_test_suite 2018-09-09 22:33:03 34 sec | #30 CI_tool_run_integration_sanity_test 2018-09-09 22:33:43 8 min 57 sec
Pipeline #42 | #42 CI_tool_build_trigger_job 2018-09-09 22:21:56 2,3 sec | #32 CI_tool_synchronize_local_repositories 2018-09-09 22:22:01 6,4 sec | #29 CI_tool_run_python_script_to_select_optimal_test_suite 2018-09-09 22:22:11 0,94 sec | #36 CI_tool_run_optimal_test_suite 2018-09-09 22:22:16 29 sec | #29 CI_tool_run_integration_sanity_test 2018-09-09 22:22:51 3 min 33 sec
```

Fig. 8.18. Successful execution of a Jenkins pipeline: #43 CI_tool_build_pipeline preceded by an unsuccessful execution: #42 CI_tool_build_pipeline.

Since the changes made to the code revert the previous changes, the affected pieces of code are the same as in the commit that has introduced some bugs to the code. Therefore, Jenkins job CI_tool_run_Python_script_to_select_optimal_test_suite returns the identical list of 7 test cases that have been already described in the preceding paragraph. The results of the Python script for the corresponding Jenkins job #30 are exactly the same as the ones presented in figure 8.13.

Figure 8.19. contains console output of the Jenkins job that executes an optimal test suite comprising of the selected 34 test runs. 34 test runs out of 34 test runs submitted to execution have passed. In figure 8.20. it is shown that the application is up and running. This claim is supported by the fact that the integration sanity checks have passed. Not only are the results a sign of working software, successful execution of unit and integration tests also confirms that the entire test suite is stable since there is no random failure in test runs.

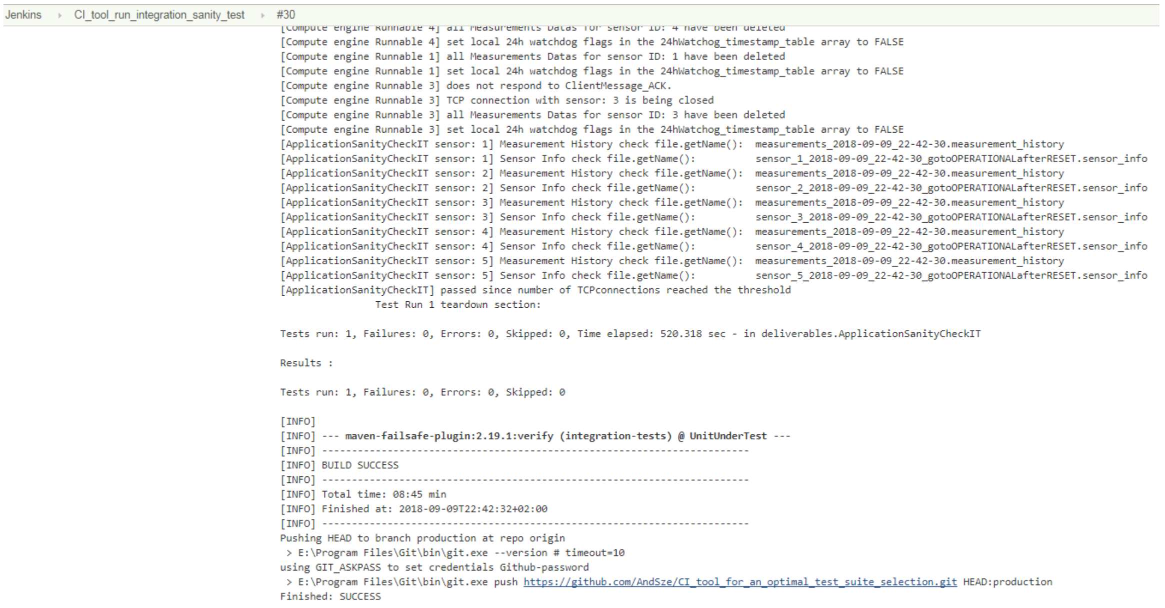
Based on the Jenkins post build action, a commit able to be built and tested successfully by CI_tool_build_pipeline is merged to the production branch. In figure 8.21. there is a screenshot of the production branch that contains the last commit merged to this branch. The commit ID and message that are shown in figure 8.22. identify the commit described in this paragraph.



Jenkins › CI_tool_run_optimal_test_suite › #37

```
                Test Run 5 Purpose:
Verify outputs of the _1h_Watchdog_close_to_expire() function call for input_1h_watchdog higher than Watchdog_Thresholds.LOWEST and no_of_measurements equal to
TCPserver.getMeasurements_limit() - 1
                Test Run 5 Logic:
Server Thread Started.
[Compute engine Runnable] Multithreaded Server Service has been started
[Compute engine Runnable 1] _1h_watchdog equals : 0.6 when leaving _1h_Watchdog_close_to_expire()
[Compute engine Runnable 1] Global_1h_Watchdog equals : 1.2 when leaving _1h_Watchdog_close_to_expire()
                Test Run 5 teardown section:
Server Thread Stopped.
Serversocket closed
[TCPserver] all attributes of the static TCPserver class are reinitialized to default values

                Test Run 6 Purpose:
Verify outputs of the _1h_Watchdog_close_to_expire() function call for input_1h_watchdog lower than Watchdog_Thresholds.LOWEST and no_of_measurements equal to
TCPserver.getMeasurements_limit() - 1
                Test Run 6 Logic:
Server Thread Started.
[Compute engine Runnable] Multithreaded Server Service has been started
[Compute engine Runnable 1] _1h_watchdog equals : 0.4 when leaving _1h_Watchdog_close_to_expire()
[Compute engine Runnable 1] Global_1h_Watchdog equals : 0.8 when leaving _1h_Watchdog_close_to_expire()
                Test Run 6 teardown section:
Server Thread Stopped.
Serversocket closed
[TCPserver] all attributes of the static TCPserver class are reinitialized to default values

Tests run: 6, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 2.056 sec - in tcpServer._1h_Watchdog_close_to_expireTest

Results :

Tests run: 34, Failures: 0, Errors: 0, Skipped: 0

[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 28.406 s
[INFO] Finished at: 2018-09-09T22:33:37+02:00
[INFO] ------------------------------------------------------------------------
Triggering a new build of CI_tool_run_integration_sanity_test
Finished: SUCCESS
```

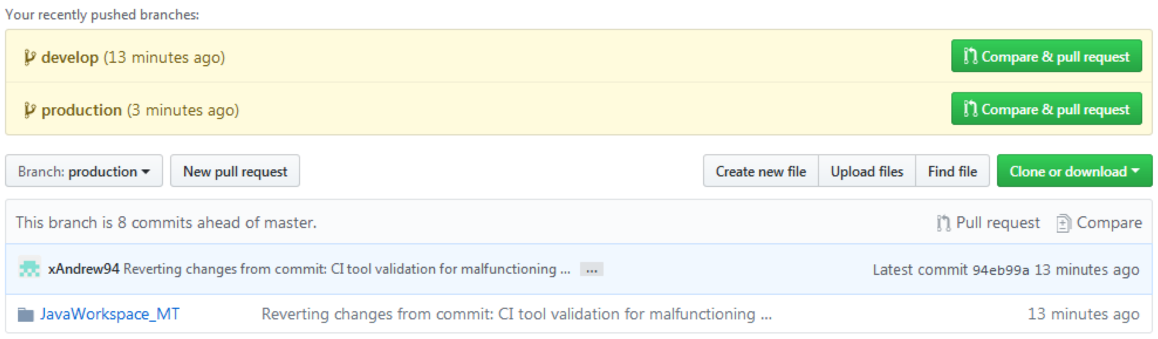
Fig. 8.19. Successful execution of a Jenkins job: #37 CI_tool_run_optimal_test_suite.



Jenkins › CI_tool_run_integration_sanity_test › #30

```
[Compute engine Runnable 4] all Measurements Datas for sensor ID: 4 have been deleted
[Compute engine Runnable 4] set local 24h watchdog flags in the 24hWatchog_timestamp_table array to FALSE
[Compute engine Runnable 1] all Measurements Datas for sensor ID: 1 have been deleted
[Compute engine Runnable 1] set local 24h watchdog flags in the 24hWatchog_timestamp_table array to FALSE
[Compute engine Runnable 3] does not respond to ClientMessage_ACK.
[Compute engine Runnable 3] TCP connection with sensor: 3 is being closed
[Compute engine Runnable 3] all Measurements Datas for sensor ID: 3 have been deleted
[Compute engine Runnable 3] set local 24h watchdog flags in the 24hWatchog_timestamp_table array to FALSE
[ApplicationSanityCheckIT sensor: 1] Measurement History check file.getName():  measurements_2018-09-09_22-42-30.measurement_history
[ApplicationSanityCheckIT sensor: 1] Sensor Info check file.getName():          sensor_1_2018-09-09_22-42-30_gotoOPERATIONALafterRESET.sensor_info
[ApplicationSanityCheckIT sensor: 2] Measurement History check file.getName():  measurements_2018-09-09_22-42-30.measurement_history
[ApplicationSanityCheckIT sensor: 2] Sensor Info check file.getName():          sensor_2_2018-09-09_22-42-30_gotoOPERATIONALafterRESET.sensor_info
[ApplicationSanityCheckIT sensor: 3] Measurement History check file.getName():  measurements_2018-09-09_22-42-30.measurement_history
[ApplicationSanityCheckIT sensor: 3] Sensor Info check file.getName():          sensor_3_2018-09-09_22-42-30_gotoOPERATIONALafterRESET.sensor_info
[ApplicationSanityCheckIT sensor: 4] Measurement History check file.getName():  measurements_2018-09-09_22-42-30.measurement_history
[ApplicationSanityCheckIT sensor: 4] Sensor Info check file.getName():          sensor_4_2018-09-09_22-42-30_gotoOPERATIONALafterRESET.sensor_info
[ApplicationSanityCheckIT sensor: 5] Measurement History check file.getName():  measurements_2018-09-09_22-42-30.measurement_history
[ApplicationSanityCheckIT sensor: 5] Sensor Info check file.getName():          sensor_5_2018-09-09_22-42-30_gotoOPERATIONALafterRESET.sensor_info
[ApplicationSanityCheckIT] passed since number of TCPconnections reached the threshold
                Test Run 1 teardown section:

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 520.318 sec - in deliverables.ApplicationSanityCheckIT

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

[INFO]
[INFO] --- maven-failsafe-plugin:2.19.1:verify (integration-tests) @ UnitUnderTest ---
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 08:45 min
[INFO] Finished at: 2018-09-09T22:42:32+02:00
[INFO] ------------------------------------------------------------------------
Pushing HEAD to branch production at repo origin
 > E:\Program Files\Git\bin\git.exe --version # timeout=10
using GIT_ASKPASS to set credentials Github-password
 > E:\Program Files\Git\bin\git.exe push https://github.com/AndSze/CI_tool_for_an_optimal_test_suite_selection.git HEAD:production
Finished: SUCCESS
```

Fig. 8.20. Successful execution of a Jenkins job: #30 CI_tool_run_integration_sanity_test.



Your recently pushed branches:

develop (13 minutes ago) — Compare & pull request

production (3 minutes ago) — Compare & pull request

Branch: production ▾ | New pull request | Create new file | Upload files | Find file | Clone or download ▾

This branch is 8 commits ahead of master. — Pull request | Compare

xAndrew94 Reverting changes from commit: CI tool validation for malfunctioning … … — Latest commit 94eb99a 13 minutes ago

JavaWorkspace_MT — Reverting changes from commit: CI tool validation for malfunctioning … — 13 minutes ago

Fig. 8.21. Code merged to the production branch after successful execution of Jenkins pipeline: #43 CI_tool_build_pipeline.

9. Conclusion

The Continuous Integration tool that supports the process by an optimal test suite selection fulfills the pre-defined high-level requirements. It does mean that the Jenkins build pipeline along with all project artifacts the CI tool comprises of presents the desired system behavior and may potentially improve the software development process.

The detailed description of the project artifacts is available in the previous sections of the thesis. The set of project artifacts includes the application, unit tests, integration unit tests, the sanity check, the optimal test suite selector and finally, the continuous integration build pipeline. Each of the aforementioned deliverables fulfills its pre-defined low-level requirements. Therefore, there can be little doubt that the solution proposed in the thesis is fit for purpose.

Supposedly, in order to make the best summary of the material covered by the scope of the thesis there should be a retrospective look at the general principles of the CI process made. These principles have been already described in paragraph 3.1. and can be defined as follows:

- Maintain a code repository.
- Automate the build,
- Make the build self-testing,
- Every commit should be built on an integration machine,
- Keep the build fast,
- Test in a clone of the production environment,
- Make it easy for anyone to get the latest executable version,
- Everyone can see the results of the latest build,
- Automate deployment.

The above list was mentioned again in this section for the purpose of illustrating that the CI tool successfully uses each of the principles. Furthermore, the proposed solution introduces additional features such as optimal test suite selection and the possibility to assess risks of an impact of the changes to software. The latter functionality may be extremely useful in critical chain project management (CCPM).

9.1. Possibilities for further improvement

The CI tool is able to expand its functionality in order to streamline the software development process further at the entire software life cycle level. In figure 9.1. there is a block diagram that illustrates how software life cycle work products are related to each other.

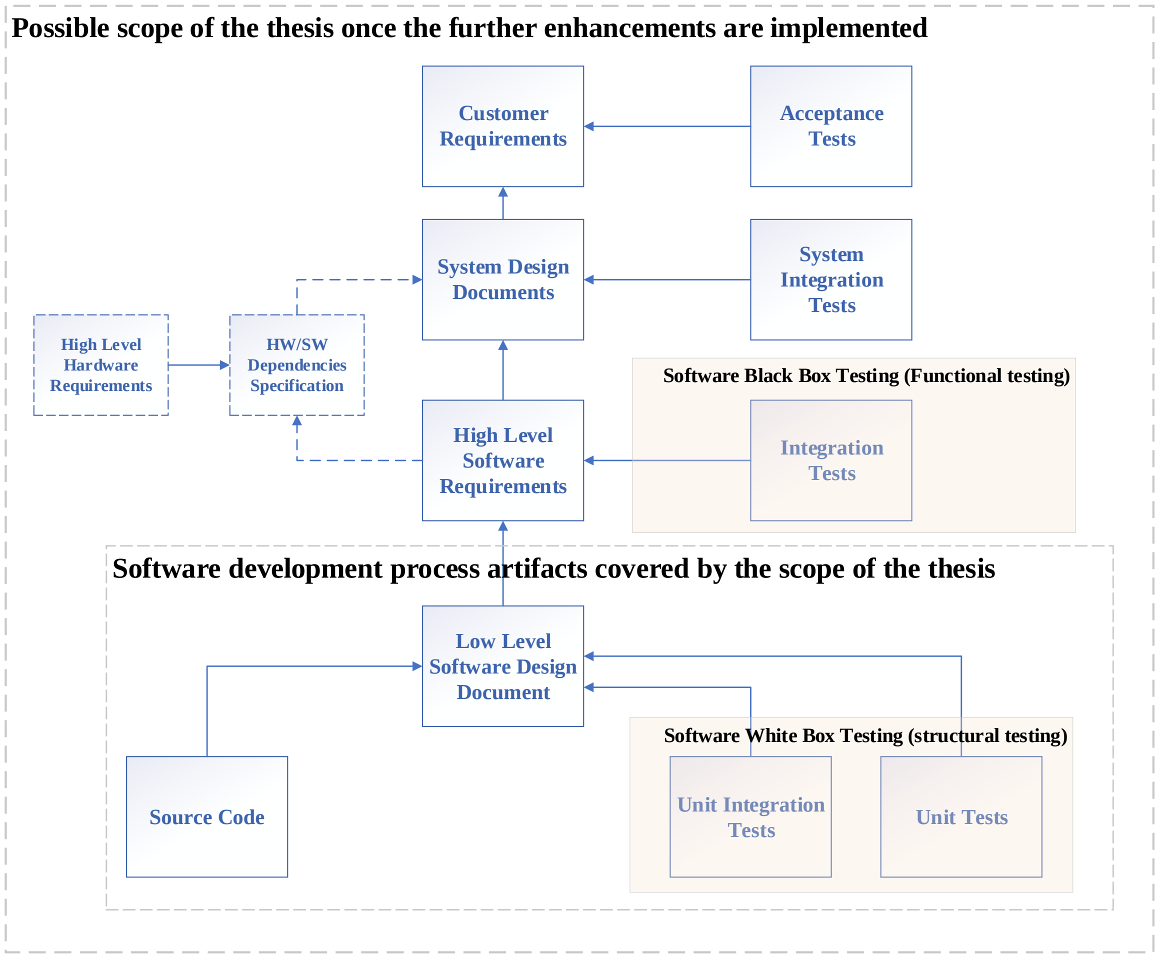

Possible scope of the thesis once the further enhancements are implemented
Customer Requirements
Acceptance Tests
System Design Documents
System Integration Tests
High Level Hardware Requirements
HW/SW Dependencies Specification
Software Black Box Testing (Functional testing)
High Level Software Requirements
Integration Tests
Software development process artifacts covered by the scope of the thesis
Low Level Software Design Document
Software White Box Testing (structural testing)
Source Code
Unit Integration Tests
Unit Tests

Fig. 9.1. The types of software life cycle work products with an indication where the scope of the thesis applies to the software development process.

Given the fact that the scope of the thesis covers the low-level software artifacts and each of the artifacts resulting from the high-level project activities is somehow related to the low-level part, it is easy to imagine the attractive features of the CI tool that can be developed in the future. There is no need to implement a separate logic for the various software artifacts as it has been done in the Python script for an optimal test suite selection since all artifacts are usually tracked in a requirement management tool. One of the most commonly used requirement management tool is Rational Dynamic Object Oriented Requirements System (DOORS). In DOORS, every software project artifact is defined in a separate module by a large set of objects. Each object contains detailed documentation of some small part of the module. In DOORS, a module is the synonym for a software project artifact. In figure 9.1. there are typical modules used for defining a software project in DOORS presented.

Basically, from the practical point of view, an object in DOORS is a requirement or a test that verifies some requirements. For example, requirements from the low-level modules are linked to the corresponding objects in high-level modules. Respectively, low level test cases (unit tests, unit integration tests) are linked to the low level software design document, which contains description of every function implemented in the code. In the thesis, the information about implementation of the functions was included in the functions headers. Figure 9.1. illustrates the links between all modules. Once the functionality able to link all deliverables of the software development process is ready, starting with even the smallest change to any piece of code, the proposed solution will visualize impact of this change to the high-level requirements. Similarly, there will be a possibility to visualize the impact in the opposite direction.

Bibliography:

[1] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mallor, Ken Shwaber, Jeff Sutherland: The Agile Manifesto. Snowbird, February 2001
[2] RBCS - Rex Black Consulting Services, Inc.: Software Testing Book, 2003-2011
[3] ISTQB Foundation Level 2018 in a Nutshell.
Available: https://www.istqb.org/certification-path-root/foundation-level-2018.html (visited August 9th, 2018)
[4] Exploring Jenkins Pipelines: a simple delivery flow.
Available: https://bulldogjob.pl/articles/726-exploring-jenkins-pipelines-a-simple-delivery-flow (visited August 10th, 2018)
[5] TechBeacon: 10 companies killing it at DevOps.
Available: https://techbeacon.com/10-companies-killing-it-devops (visited August 10th, 2018)
[6] Andrew Siemer, graphic posted on LinkedIn profile.
[7] Continuous Integration: important principles and practices.
Available: https://www.thoughtworks.com/continuous-integration (visited August 25th, 2018)
[8] Java SE 8 Features and Enhancements. Oracle Corporation, JDK 8 Release Notes.
[9] Mockito - Tasty mocking framework for unit tests in Java.
Available: https://site.mockito.org/ (visited August 28th, 2018)
[10] Jenkins user documentation. Available: https://jenkins.io/ (visited August 28th, 2018)
[11] Apache Maven: Maven concepts.
Available: https://en.wikipedia.org/wiki/Apache_Maven (visited September 3rd, 2018)
[12] Petri Kainulainen: Integration Testing With Maven.
Available: https://www.petrikainulainen.net/programming/maven/integration-testing-with-maven/ (visited September 7th, 2018)
[13] Apache Maven Project: Introduction to the Build Lifecycle in Maven.
Available: https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html (visited September 7th, 2018)

List of tables:

List of figures:

Attachments:

[1] Source code of the application. Available:
https://github.com/AndSze/CI_tool_for_an_optimal_test_suite_selection/tree/master/JavaWorkspace_MT/UnitUnderTest/src/main/java

[2] Source code of the tests. Available:
https://github.com/AndSze/CI_tool_for_an_optimal_test_suite_selection/tree/master/JavaWorkspace_MT/UnitUnderTest/src/test/java

[3] The POM files containing project build configuration. Available:
https://github.com/AndSze/CI_tool_for_an_optimal_test_suite_selection/tree/master/JavaWorkspace_MT/UnitUnderTest

[4] Python scripts for an optimal test suite selection. Available:
https://github.com/AndSze/CI_tool_for_an_optimal_test_suite_selection/tree/tests_selector

[5] Jenkins jobs that are used in the CI build pipeline. Available:
https://github.com/AndSze/CI_tool_for_an_optimal_test_suite_selection/tree/JenkinsJobs