

Evgenii Kucheruk

**CODE COVERAGE EFFECTIVENESS IN CONTINUOUS INTEGRATION OF QT
FRAMEWORK AND PRODUCTS**

**CODE COVERAGE EFFECTIVENESS IN CONTINUOUS INTEGRATION OF QT
FRAMEWORK AND PRODUCTS**

Evgenii Kucheruk
Bachelor's Thesis
Spring 2021
Information Technology
Oulu University of Applied Sciences

ABSTRACT

Oulu University of Applied Sciences
Degree Programme in Information Technology

Author(s): Evgenii Kucheruk

Title of Bachelor's Thesis: Code Coverage Effectiveness in Continuous Integration of Qt Framework and Products.

Supervisor(s): Kari Jyrkkä

Term and year of completion: Spring 2021

Number of pages: 43

This Bachelor's thesis explores the effectiveness, importance, and implementation of the automated software checking using code coverage of Qt framework and products. The thesis is a continuation of company-oriented projects, including initial preparations such as getting code coverage results manually for the different framework versions (baseline) and fixing system environment problems for the operating systems.

The primary purpose of the thesis is to automate code coverage procedures. Thus, it will be easy for The Qt Company to track how patches and releases impact code coverage results in general. Knowing the code coverage results is essential for The Qt Company because higher coverage shows that the product sustainability improves and helps with marketing purposes.

As a result of company-oriented projects, it was clear that knowing trends of code coverage and measuring it manually helped teams inside the company keep an eye on the changes and impact before the actual release of the framework product happens.

The results of the thesis work provided evidence that increasing code coverage, in general, would prevent future bugs and improve the maturity of the product.

Keywords: Code Coverage, Continuous Integration, The Qt Framework, Quality Assurance.

ACKNOWLEDGEMENT

First and foremost, I would like to thank my supervisors from The Qt Company and OAMK: Asmo Saarela and Kari Jyrkkä, respectively: this work would not have been possible without your endless support. Their prodigious knowledge and ample experience have encouraged me in all the time of my research and daily life. My deepest gratitude extends to The Qt Company colleagues who helped me along the way, as well as the professors from OAMK.

Additionally, I would like to thank my dearest friends whom I met years before and during my studies. Thank you for supporting me. It is my pleasure to know each one of you.

To my family and the love of my life: thank you for your endorsement and having faith in me. There are no words that can describe my gratitude and love towards you.

CONTENTS

1	INTRODUCTION	7
2	CODE COVERAGE	9
3	CONTINUOUS INTEGRATION SYSTEM ARCHITECTURE	10
4	IMPLEMENTATION PLAN	12
5	WEEKLY REPORTS	17
5.1	Week 24	18
5.2	Week 25	19
5.3	Week 26	21
5.4	Week 27	24
5.5	Week 28	25
5.6	Week 29-31	27
5.7	Week 32	29
5.8	Week 33-34	30
5.9	Week 35-37	33
5.10	Week 38	35
5.11	Week 39-41	37
5.12	Week 42-44	38
6	CONCLUSION	40
	REFERENCES	41

GLOSSARY

LCOV	A graphical front-end coverage testing tool for gcov. It collects gcov data for multiple source files and creates HTML pages containing the source code annotated with coverage information. It also adds overview pages for easy navigation within the file structure. (3)
GCC	(GNU Compiler Collection) – is a compiler system produced by the GNU Project supporting various programming languages. GCC is a key component of the GNU toolchain and the standard compiler for most projects related to GNU and Linux. (4)
G++	(GNU C++ Compiler), The C++ compiler. (5)
GCOV	A test coverage program. It is used in concert with GCC to analyze programs to help create more efficient, faster running code. (6)
FrogLogic Coco	A cross-platform and multi-language code coverage tool. Automatic source code instrumentation is used to measure test coverage of statements, branches, and conditions. (7)
qmake	a utility that automates the generation of makefiles. Qmake generates a Makefile based on the information in a project file. (8)
CMake	An open-source, cross-platform family of tools designed to build, test, and package software. CMake is a build-system generator. (9)
CI	in this thesis, referred to as the "Continuous integration team" of The Qt Company.
Coin	in this thesis, referred to as the "Continuous integration system" of The Qt Company.

1 INTRODUCTION

The purpose of this diary-based thesis is to show what tasks and challenges are happening weekly inside The Qt Company. The main target is to implement an automated code coverage system. However, since the work is done inside the research and development area, it is not possible to choose only one topic or goal because priorities might change daily and other tasks might become more important at the given phase; thus, the diary-based version of the thesis was the only right choice due to its flexibility.

The importance of code coverage is hard to underestimate. Generally, code coverage is a measure of code that is executed with the tests. Generally, the benefits of using a code coverage tool to check your products are:

1. Find out which parts of the code are covered with tests.
2. Find the parts which are missed/not tested
3. High code coverage points to a well-written and testable code
4. High code coverage is crucial to investors (higher the coverage – more trust in the product)
5. High code coverage matters for some customers (especially if the product is used in the areas where safety matters the most). (1)

Keeping track of code coverage can be done manually, however, it takes human hours to build, run the tests, and collect the data. Thus, automation is not only crucial to free people from doing that but also to track all the changes with all the releases/patches, which manually would take an entire day of work to provide the report.

The Qt Company is a global software company based in Espoo, Finland, with a strong presence in more than 70 industries and is the leading independent technology behind millions of devices and applications. Qt is widely used by major global companies and developers around the globe, which is being achieved through its cross-platform software framework. Technologies of The Qt Company are being used by approximately one million developers worldwide with a single software code across all operating systems, platforms, screen types. From desktops and embedded systems to business-critical applications, in-vehicle systems, wearables, and mobile devices connected to the

Internet of Things. (2) For example, high code coverage is important in the automotive industry, which requires the highest safety.

2 CODE COVERAGE

Code coverage is a way of using analytics to get an idea of how well an application was tested (or measure a test suite quality). The purpose of software testing, in general, is to improve the quality, and code coverage is one of the tools that can lead towards better code. It is, undoubtedly, impossible to find all faults in an application and state that there are none. Usually, testing is treated as follows: on the one hand, there is a cost of future testing, writing test cases, and on the other hand are potential losses of faults in the future.

In the actual case scenario, based on the experience from the different research studies and work-related tasks, there are controversial statements on whether code coverage is beneficial or not. For example, according to the survey, once code coverage was implemented inside the IBM Company, it increases test suite quality (15). Another research article states that there is some evidence that code coverage works in practice and leads to a positive outcome in terms of quality; however, a different approach was used (using mutation testing method) (16). Last but not least, it was proven that there is a correlation between statement coverage and a number of bug fixes (17). Thus, for the given scenarios, code coverage was proven to be an effective analysis tool.

Nevertheless, it should also be mentioned that, for example, in L. Inozemtseva and R. Holmes's study, code coverage does not show a strong correlation with test suite effectiveness (18).

Some works claim that the answer to "is code coverage effective" is dependent on different factors and variables (e.g., size of the project) (19).

Summarizing the given statements, it is safe to say that code coverage effectiveness is based on various factors: programming language itself, tools that are used to measure code coverage, type of code coverage (MC/DC, Branch, Statement, etc.), size of the project and many more different things. For The Qt Company, code coverage was chosen as the analyzing tool in addition to the variety of different methods of testing.

3 CONTINUOUS INTEGRATION SYSTEM ARCHITECTURE

The company has implemented a continuous integration system that includes building the given module or entire Qt framework and running the tests. However, the implementation of code coverage measurements and collecting the data were not part of the automation system. Until now, it has been done manually.

To this day, the continuous integration system helps developers and testers schedule a build that will be done on a virtual machine (VM) without the need of a user to interact with that (except running with the pre-defined parameters). It helps to save time, avert the human factor that can cause errors, keep the runs on the pre-defined machines, configurations, and specifications.

Figure 1 below illustrates the workflow of the continuous integration system.

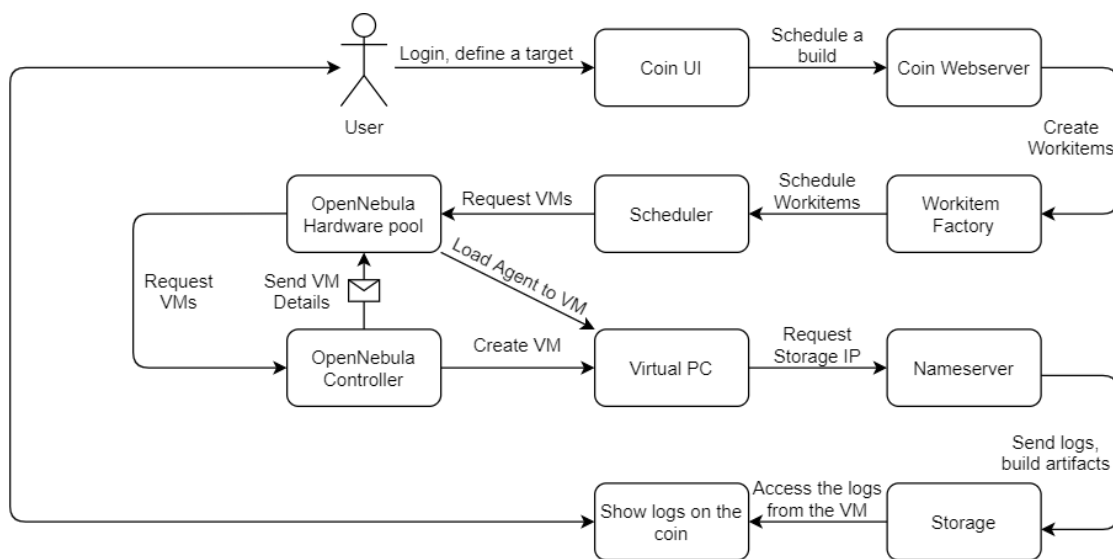


FIGURE 1. Example process flow.

The process to run is:

1. A user defines the needed parameters, versions, tested OS in the Continuous integration system (Coin).
2. A User schedules the build to start the integration process.

3. Webservice sends a message to a Workitem Factory, which parses the information and creates a matching set of workitems to perform all requested tasks, includes creating, provisioning and running test/build instructions on the VM.
4. A Workitem factory sends the workitems to the scheduler to be executed.
5. A Scheduler requests a suitable VM instance from Open Nebula Hardwarepool (ONHWP)
6. ONHWP forwards the request to Open Nebula over Open Nebula REST API to create a suitable VM.
7. Open Nebula Controller (ONC) creates a suitable VM.
8. When VM is created, its address is returned to ONHWP by the ONC.
9. Scheduler/ONHWP uploads Agent code to create the VM. An agent spawns up and registers back to ONHWP on the same connection.
10. An agent requests a Storage IP address from Nameserver.
11. An agent stores test/build logs to Storage and any defined build artifacts such as created executables, libraries, etc.
12. Log process is uploaded to the coin website in real-time, so a user can check it while the build is going or after the integration passed.

The continuous integration system helps the company to build and run the tests of the products. Since the process takes several hours to build the framework and run the tests, it is possible to set the needed parameters or configurations that need to be tested and check the result in the evening without any interference because each step is executed with the code. Moreover, it helps to avoid "human-related" factors as, for example, a person could forget to run a certain script. Besides, one person can start several builds with different frameworks or configurations at the same time and work with another project.

The continuous integration system is certainly helpful in many aspects. However, code coverage phase was never included in that.

4 IMPLEMENTATION PLAN

4.1 Project plan

As stated before in chapter 1, the main goal of the project is to implement a separate continuous integration system that will include the code coverage feature. The plan goes as follows:

1. Implementation of the personal continuous integration system (clone the existing one).
2. Add the feature that will enable the code coverage.
3. Code what the feature will do, the correct execution order.
4. Ensure that builds and tests are running well for every integration.
5. Create the script to archive the code coverage results and transfer them to the fileserver with the timestamp.

The company has a common continuous integration system in which developers can run the specific framework version with the given parameters on various platforms. However, in this project, it was needed to create a personal system to make changes to the automation. It is essential for several reasons:

1. If the changes are needed, the system must be restarted; thus, it may cause problems with the running builds.
2. To save time, builds are not running again for the same configurations but are copied from the artifacts from the previous successful runs if the SHA1s are comparable. It is not possible to use these because code coverage measurement files are not part of the repository.
3. The build system in general. For precision, code coverage measurements must be done at one place for the entire build system (from the parent folder), but it has been done separately for simplicity and time.

Thus, it is necessary to create an environment specifically for the code coverage to not interfere with the default procedures.

4.2 Manual process to reproduce

Before doing the measurement part itself, it is crucial to understand how the builds and tests are happening inside the continuous integration system. Thus, the logs of the successful runs would be helpful. This process can be divided as follows:

1. Preparation of the machine itself, which include:
 - 1.1. Clone the binaries of the Qt Framework from the git repository.
 - 1.2. Set up the environment (run the provisioning scripts which will install the needed dependencies)
2. Configure the Qt Framework with the needed parameters. This process will change the .pro files accordingly to the given configurations (for example: to enable code coverage, "-gcov" should be executed as one of the configuration parameters. This parameter applies "-gO -Wall -fprofile-arcs -ftest-coverage" to the CXX_FLAGS).
3. Build the Qt Framework.
4. Tests execution.
5. Start the code coverage execution process, which will include the following steps:
 - 5.1. Run the LCOV from the parent folder to collect the data into one file.
 - 5.2. "Clear" the file that was produced from the previous step to exclude the unwanted folders, files, and unnecessary information.
 - 5.3. Run the "genhtml" feature to generate the HTML report (One of the way to convert the data to the human-readable format)

The explanation of each step:

1. Preparation of the machine. In this part has to be decided which operating system or framework needs to be tested. For simplicity and generalization, it was planned to use Ubuntu 18.04, which goes perfectly with Qt 5.12.8. This phase requires cloning the repository from git and installing LCOV (sudo apt-get install lcov).
 - 1.1. Download the Qt Framework (in this example, it will be downloaded from the git)

- 1.2. System preparation. Each Qt version (if downloaded from git) with the provisioning scripts done by the CI team. The scripts contain all the necessary software and dependencies to build Qt. Every script has a number, which shows the order in which they should be executed. If the purpose of the given run is to check the code coverage, it is essential to check that GCC, G++, and GCOV are using the same version. Otherwise, it will not work due to different programming syntax. Figure 2 shows the provisioning scripts for Qt 5.12.8.

```
01-install_telegraf.sh          09-install-openssl.sh
01-systemsetup.sh              22-mqtt_broker.sh
02-apt.sh                       30-fbx.sh
02-disable-notifications.sh     35-install-breakpad.sh
02-disable-ntp.sh              40-android_linux.sh
02-git_lfs.sh                  40-cmake.sh
02-remove-apport.sh            50-openssl_for_android_linux.sh
02-remove-update_notifier.sh   60-qnx660.sh
03-gcc.sh                      70-qnx700.sh
03-qemu.sh                     80-docker.sh
04-libclang.sh                 90-squish.sh
04-yocto.sh                    91-squish-coco.sh
04-yocto_ssh_configurations.sh 99-version.sh
```

FIGURE 2. Provisioning scripts for the Qt 5.12.8 (Ubuntu 18.04)

2. Since Qt has dozens of configuration parameters, for precision and repeatability, figure 3 shows the configuration parameters were used for each run:

```
qt@evgenii-ubuntu1804:~/qt5full/qt5$ ./configure -gcov -no-use-gold-linker -R .
-openssl -opensource -confirm-license -verbose -developer-build -nomake examples
-qtnamespace TestNamespace
```

FIGURE 3. Configuration parameters for the Qt 5.12.8

3. The building is done by running the "make" command after the configuration. At this phase, all the source code, as well as tests, are being built. Build can be done with the power of all existing cores by providing -jX as the argument, in which X = number of cores. This will speed building time significantly.
4. The test phase implies running the autotests. Figure 4 shows the needed command to start the execution of the tests phase.

```
qt@evgenii-ubuntu1804:~/qt5full/qt5$ make check -j1 -i >testslog.info 2>&1
```

FIGURE 4. Starting the tests for the Qt 5.15.8

"make check" runs the tests, "-j1" sets to run the test using a single core (it is needed because some tests could not be executed while running them in parallel, which will increase the amount of failed or flaky tests up to 10 times). "-i" stands for "ignore errors", thus, tests will continue to run even after failed execution. Lastly, ">testslog.info 2>&1" is needed to write logs into the "testslog.info" file. This will include passed, failed, skipped, and blacklisted tests.

5. Now comes the coverage part. The coverage data files are created at the build (files with the .gcno postfix) and test (files with the .gcda) phases. The first one contains information to reconstruct the basic block graphs and assign source line numbers to blocks. The second file contains arc transition counts, value profile counts, and summary information. (10)

5.1. The next step is to combine the profiling files (.gcno and .gcda) into one (because each object file creates .gcno and .gcda files correspondingly). Figure 5 shows how to do so. The result of this program is the "reportdata.info" file which holds still raw but combined data.

```
qt@evgenii-ubuntu1804:~/qt5full/qt5$ lcov -c -d . -o reportdata.info
```

FIGURE 5. Combining the profiling files under the qt5 (parent) directory into the "reportdata.info" file.

5.2. The next step is to clear all unnecessary and irrelevant files from the report because raw data from the previous step also contains code coverage of the system files, tests itself, .moc files, etc. To filter the file, the command in Figure 6 needs to be executed:

```
qt@evgenii-ubuntu1804:~/qt5full/qt5$ lcov --remove reportdata.info "*.moc" "*.rcc"
"*/.moc/*" "*/.rcc*" "*.g" "*/3rdparty/*" "*/usr/lib/*" "*/usr/include*" "*/tests
/*" "*/generated/*" "*.y" "*.gperf" "*.ypp" -o Qt5_Root.info
```

FIGURE 6. Clearing the .info file from the irrelevant data.

5.3. Finally, it needs to be converted from the raw data to the human-readable (HTML in this example) format. Figure 7 shows the command that needs to be executed to create the HTML report.

```
qt@evgenii-ubuntu1804:~/qt5full/qt5$ genhtml Qt5_Root.info -o Qt_report
```

FIGURE 7. Generation of the HTML report

5.4. After the execution, the sample result should look like it is shown in Figure 8.

LCOV - code coverage report

Current view: top_level - animation		Hit	Total	Coverage
Test: Qt_corelib_self.info		Lines: 1250	1377	90.8 %
Date: 2020-02-10 12:10:00		Functions: 267	298	89.6 %

Filename	Line Coverage	Functions
abstractanimation.cpp	91.5 % 493 / 538	88.8 % 79 / 88
abstractanimation.h	100.0 % 2 / 2	100.0 % 4 / 4
abstractanimation_p.h	93.8 % 15 / 16	83.3 % 15 / 18

FIGURE 8. Example extract from the LCOV code coverage report.

The LCOV tool is helpful because it shows the general information about the code coverage and the exact places that were not executed by the tests, and how many times specific parts of code were accessed, as shown in Figure 9. The red color states that the code was not executed, while the blue background shows that tests could reach that part of the code and how many times that part was executed.

```

:
:
: 0 : QNodeId EnvironmentLight::shaderData() const
:   : {
:   :   return m_shaderDataId;
:   : }
:
:
: 8 : void EnvironmentLight::initializeFromPeer(const QNodeCreatedChangeBasePtr &change)
:   : {
:   :   const auto typedChange = qSharedPointerCast<Qt3DCore::QNodeCreatedChange<QEnvironmentLightData>>(change);
:   :   const auto &data = typedChange->data;
:   :   m_shaderDataId = data.shaderDataId;
:   : }

```

FIGURE 9. Example detailed code coverage report of the random file.

4.3 Tool choice for the project

In this thesis, the tools that are used to collect code coverage data are gcov and FrogLogic CoCo. Ideally, the second tool should have been used for the entire project. However, due to incorrect output within the massive project (whole Qt 5 Framework), it was decided to use gcov for its simplicity (since the provided information was enough and it was used already in the previous projects). As for the Qt SafeRenderer and Qt Design Studio, since the project is smaller, it was better to use the FrogLogic CoCo tool. This tool is better because it provides an expanded version of the report. It is also possible to generate different kinds of code coverage reports that the default gcov tool cannot produce (14).

In addition to that, FrogLogic CoCo tool uses "source code coverage", while GCOV works on the GCC level (binaries).

5 WEEKLY REPORTS

After the preparation of the plan and showing the manual execution, which is required to get code coverage data, the actual work can be started. Each week will contain a summary of the tasks that were done. Moreover, it will include findings, problems, and a way to fix them, preparations, and other things that might be useful. Since this is a diary-based thesis, some more planning for the different projects can be done in the upcoming weeks.

5.1 WEEK 24

Implementation of the continuous integration system has begun. At the initial planning, it was decided that the first target system would be Ubuntu 18.04 and the Qt 5.12.8 because it was done several times manually. It was planned that before reaching code coverage part, there is a need to check that the default process works as intended and without errors. The difference between a personal continuous integration system and a default one is that the first one requires running the entire module in one place. In contrast, the second one splits the build into the different modules.

Realization plans were as following:

1. Discuss with the CI team about creating a personal, tweakable coin instance. In the future, other teams might use this so it will not interfere with the streamlined process.
2. Prepare the plan for configuration, build, test, gather the data, filter the irrelevant information, and transfer it to the fileserver.
3. Compare the results with the manual runs, ensure that system works well, and data is reliable.
4. Create the script which will install the LCOV tool (not the part of the default provisioning scripts) and sets the versions of gcov, GCC, and G++ to match each other.

5.2 WEEK 25

The week began with the actual implementation of the personal continuous integration system. The machine and configurations with the personal credentials were created. However, it operates as the default one. Thus, tweaking of the system is required.

As stated before (Chapter 3), the first step is to run everything in the same (parent) folder. Since this personal instance is based on the working one, it splits modules automatically. In that case, a unique parameter had to be created with the platform instruction configuration. Figure 10 shows how the feature was stated

```
if Feature.CODECOVERAGE in platformInstructions.config.features:  
    self.buildCodeCoverage(platformInstructions)
```

FIGURE 10. Adding the code coverage feature to the personal coin instance.

Adding "CODECOVERAGE" as the feature, when executed from the personal Coin instance, runs not the default "platformInstruction" but a specially created one. This change will affect only build, test, and actual code coverage phases, while the configuration is held by the coin system itself since there is a proper way to put them beforehand.

Ideally, the process should be done as follows:

1. Run "make" to build the Qt Framework itself. There is no need to specify how many cores need to be used because the coin initially sets "build (ninja)" flags. There is no need to run the provisioning scripts because the coin will take the "tier 2" machine with the provisioning pre-installed for the specified version of the Qt Framework.
2. Run "make check" to run the tests. At that point, since it is done by the coin and logs are available online, there is no need to specify forwarding them to the file. Moreover, the "-i" argument is irrelevant because the system repeats tests up to 5 times if there was an error or a failure in the process. In addition to that, if the code coverage parameter itself causes some tests to fail, there is another option on the coin that will not stop the tests from running after an unsuccessful run.
3. Use the LCOV to provide code coverage information as well as filtering the data.

4. Generate the HTML report, transfer that to the server (archived, so it will save the space on the HDD. The name of the archive should be the timestamp from the run)
5. Provide the link to the report at the end of the log file.

5.3 WEEK 26

Meanwhile, by the request from the different teams, it was decided to switch the priorities to help to create the report for the customers with the different versions of the Qt Framework. In addition to that, it was requested to help with the code coverage of the Qt Design Studio using the FrogLogic code coverage tool. It is different because it provides more information to an end-user, reads not only C++ and C code but also the QML.

Creating reports for the customers using different versions is trivial; the only difference is to use a newer version of the compiler. The main problem was to update-alternatives, so GCC, G++, and GCOV use the same version. The gcov was never used by the Qt before, but it comes as part of the GNU Compiler Collection, thus, it was never matched to the latest version, which has to be done manually if the different versions of the compiler were installed previously. It is done with the command shown in figure 11:

```
qt@evgenii-ubuntu1804:~/qt5full/qt5$ sudo update-alternatives --install /usr/bin/gcov gcov /usr/bin/gcov-9 100
```

FIGURE 11. Changing the gcov version with the update-alternatives.

The last number is the priority or "importance", so the higher the value – the higher priority.

Talking about Qt Design studio, the process is different compared to the GCOV+LCOV coverage. The FrogLogic tool is a standalone program with the graphical user interface. While doing the company-oriented projects, it was the second goal to measure code coverage using the FrogLogic software. The problem was that it was comparable with most of the modules, but others were not providing the coverage files properly. Thus, it was decided to postpone the project because it requires a lot of manual tweaking.

As it is mentioned above, to enable gcov, it is needed to make changes inside the .pro files (changing CXX flags), while for the FrogLogic steps to enable the coverage are:

1. Install the FrogLogic tool itself. It is necessary to have the license server inside the environmental variables for verification.

2. Adding COVERAGESCANNER_ARGS and wrappers to the \$PATH (Ubuntu)
3. Clone repository, configure, then run qmake recursively from the qtbase/bin folder with the "CONFIG += testcococon" which will include testcococon.prf file. For different purposes, it might be needed to change the file accordingly.
4. Build and test as usual. This phase will generate csmes (information needed for the coverage measurements) and csexe (results of code execution) files (11).
5. After tests are finished, Figure 12 shows command needs to be executed to merge all the csmes files into one:

```
qt@ubuntu1804:~/qtDS$ cmmmerge -o auto_tests.csmes $(find -iname "*.csmes")
```

FIGURE 12. Merge coverage measurements files into one.

6. To combine the data with the results of the code execution, it is required to run the command, shown in Figure 13:

```
qt@ubuntu1804:~/qtDS$ cmcseximport -m auto_tests.csmes -e $(find -iname "*.csexe") -t $(find -iname "*.csexe")
```

FIGURE 13. Merge the coverage measurements with the results of code execution.

7. To open the merged file, the "coveragebrowser" command should be used. The sample result should look like Figure 14:

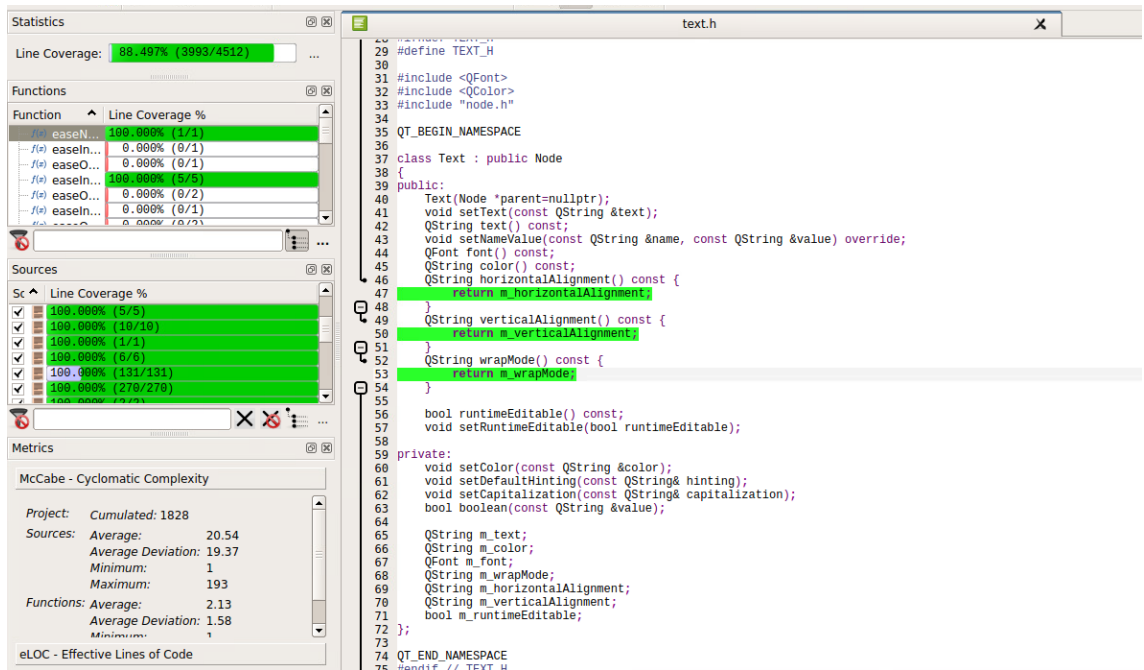


FIGURE 14. Example output of the coveragebrowser.

8. It is also possible to generate the HTML report using coveragebrowser GUI. The result will look as shown in figure 15. It is possible to generate different types of reports depends on the needs and what was requested. Usually, the most important one would be Line coverage since it covers most of the information that customers are looking for, while other reports are mostly for the developers to find out which part of the code is not working correctly.

Global Coverage

Level	Coverage
level 1	88.497% (3993/4512)

Metrics

McCabe - Cyclomatic Complexity

Project: Cumulated: 1828

Sources: Average: 20.54

FIGURE 15. An example of the HTML report using the FrogLogic Code Coverage tool.

5.4 WEEK 27

The necessary part for the Qt Design Studio code coverage was done; however, it might continue in the future.

From this week onward, it was requested to produce code coverage reports for the Qt Safe Renderer. This is one of the essential modules because it provides a solution for rendering the safety-critical information in functional safety applications based on Qt (12).

Thus, it is important that code coverage of the module would be as close to the perfect result as possible, which means it is vital to check it after every change is pushed to the repository. The prototype of the automated code coverage system was done before. However, it did not cover additional modifications to the project. Thus, it was requested to check code coverage for the entire module with the most recent update.

After the installation of the QSR project, it was noted that the results are not as great as the nightly runs were producing daily. Moreover, some tests were failing, which should not be the case for that particular project. After receiving help from the QSR team, two errors were spotted:

1. Default git version did not point to the needed version (fixed by changing to the dev branch)
2. The .prf file was pointing to the previous branch, which was incorrect.

After fixing them, it was possible to produce the following results, which are reliable:

qtsaferenderer_autotests

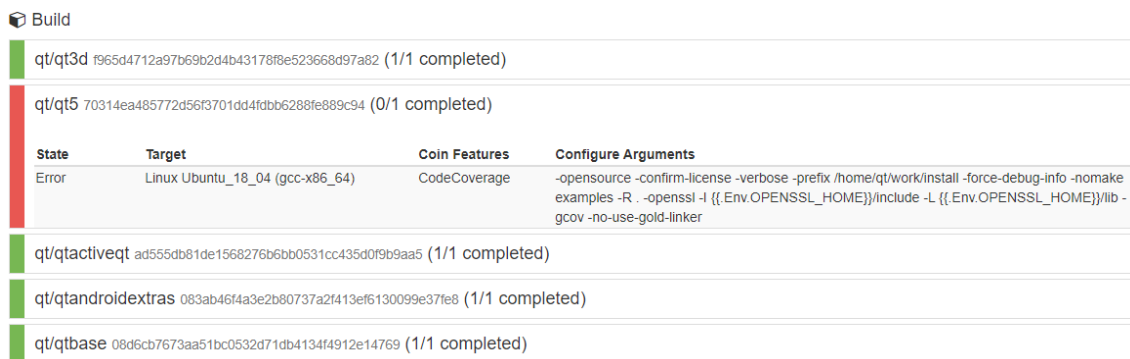
Sources

Source	Line Coverage % ▾
qsaferesource.cpp	100.000% (131/131)
text.h	100.000% (3/3)
statemanager.h	100.000% (23/23)
qsafewindow.h	100.000% (6/6)
qsafetextlayoutresourcecreator.cpp	100.000% (11/11)

FIGURE 16. An example of the QSR project code coverage report using the FrogLogic Code coverage tool.

5.5 WEEK 28

Some trial runs using the personal continuous integration system were done, but the results were far from success. By default, a machine splits the building modules and do them separately, while the code coverage case on personal coin build must be done from the one directory (top-level build). Figure 17 shows a modular split of the Qt5 integration.



Build

qt/qt3d f965d4712a97b69b2d4b43178f8e523668d97a82 (1/1 completed)

qt/qt5 70314ea485772d56f3701dd4fdbb6288fe889c94 (0/1 completed)

State	Target	Coin Features	Configure Arguments
Error	Linux Ubuntu_18_04 (gcc-x86_64)	CodeCoverage	-opensource -confirm-license -verbose -prefix /home/qt/work/install -force-debug-info -nomake examples -R. -openssl -I {{ Env.OPENSsl_HOME }}/include -L {{ Env.OPENSsl_HOME }}/lib -gcov -no-use-gold-linker

qt/qtactiveqt ad555db81de1568276b6bb0531cc435d0f9b9aa5 (1/1 completed)

qt/qtandroidextras 083ab46f4a3e2b80737a2f413ef6130099e37fe8 (1/1 completed)

qt/qtbase 08d6cb7673aa51bc0532d71db4134f4912e14769 (1/1 completed)

FIGURE 17. Coin example modular split.

At this point, it does not matter that different modules are being built properly because the "top-level" build happens inside the qt/qt5 run, which is failed. For the main qt/qt5 run, the feature "CodeCoverage" is enabled. The code that is triggered by this feature is shown in figure 18.

```
def buildCodeCoverage(self, platformInstructions: CommonInstructions) -> None:
    platformInstructions.cd('{{.SourceDir}}')
    platformInstructions.executeCommand(['make'], onIntegrationFailureMessage % "Could not execute build.", maxTime=60 * 60 * 24)
    platformInstructions.executeCommand(['make', 'check'], onIntegrationFailureMessage % "Could not execute make check.", maxTime=60 * 60 * 24)
    platformInstructions.executeCommand(['qsdctest'], onIntegrationFailureMessage % "Failure on purpose", maxTime=60 * 60 * 24])
```

FIGURE 18. Python code which triggers with the "CodeCoverage" feature

The continuous integration system does provisioning, preparation of the environment, and configuring by itself, so the steps which need to be coded are:

1. Enter the "Source directory".
2. Run the "make" command to build the entire Qt.
3. Run "make check" to execute the tests
4. Run "qsdctest" to fail on purpose because it is not the last step, and the machine needs to be accessed to check the logs/failures/change environment and restart the run, etc.

However, at this point, failure was not on the 4th step, which is "failure on purpose", but at the test phase. Flaky tests most likely caused these errors, but it is not certain now; further investigation needs to be done.

The current plan is to achieve a successful integration that includes the rest needed code coverage additions and modifications of the code.

5.6 WEEK 29-31

The given weeks did not move the progress any forward, but it was noticed that failure happens in different phases from time to time. The most problematic are the building failures since they are the ones that needed to be investigated further while test failures occur even on the default coin system.

The default coin system goes folder-by-folder and builds everything in parallel, then combines into the artifacts and runs the tests while the personal instance uses one machine to build the entire product in one place. That gave an idea to try to split the build phase by modules that did not show any success. Getting rid of the building errors was the priority task at that point. And the problem was caused by the "-gcov" feature that was enabling the code coverage. Figure 19 shows the errors produced in the sample build.

```
build.go:264: .obj/qstring_compat.o:qstring_compat.cpp:function _GLOBAL__sub_I_00100_0_qstring_compat.cpp: error: undefined reference to '__gcov_init'
build.go:264: .obj/qstring_compat.o:qstring_compat.cpp:function _GLOBAL__sub_D_00100_1_qstring_compat.cpp: error: undefined reference to '__gcov_exit'
build.go:264: .obj/qstring_compat.o(.data.rel+0x20): error: undefined reference to '__gcov_merge_add'
build.go:264: .obj/qmalloc.o:qmalloc.cpp:function _GLOBAL__sub_I_00100_0_qmalloc.cpp: error: undefined reference to '__gcov_init'
build.go:264: .obj/qmalloc.o:qmalloc.cpp:function _GLOBAL__sub_D_00100_1_qmalloc.cpp: error: undefined reference to '__gcov_exit'
build.go:264: .obj/qmalloc.o(.data.rel+0x20): error: undefined reference to '__gcov_merge_add'
build.go:264: .obj/qrunnable.o:qrunnable.cpp:function _GLOBAL__sub_I_00100_0_qrunnable.cpp: error: undefined reference to '__gcov_init'
build.go:264: .obj/qrunnable.o:qrunnable.cpp:function _GLOBAL__sub_D_00100_1_qrunnable.cpp: error: undefined reference to '__gcov_exit'
build.go:264: .obj/qrunnable.o(.data.rel+0x20): error: undefined reference to '__gcov_merge_add'
build.go:264: .obj/qsemaphore.o:qsemaphore.cpp:function _GLOBAL__sub_I_00100_0_qsemaphore.cpp: error: undefined reference to '__gcov_init'
build.go:264: .obj/qsemaphore.o:qsemaphore.cpp:function _GLOBAL__sub_D_00100_1_qsemaphore.cpp: error: undefined reference to '__gcov_exit'
build.go:264: .obj/qsemaphore.o(.data.rel+0x20): error: undefined reference to '__gcov_merge_add'
build.go:264: collect2: error: ld returned 1 exit status
build.go:264: Makefile:1315: recipe for target '../lib/libQt5Core.so.5.12.8' failed
build.go:264: make[3]: *** [../lib/libQt5Core.so.5.12.8] Error 1
build.go:264: make[3]: Leaving directory '/home/qt/work/qt/qt5/qtbase/src/corelib'
build.go:264: Makefile:2639: recipe for target 'sub-corelib-check' failed
build.go:264: make[2]: *** [sub-corelib-check] Error 2
```

FIGURE 19. Sample build errors caused by the gcov.

The figure shows that the first error at the building phase happens because of the "undefined reference to '__gcov_exit'". However, that situation does not occur when building the Qt using the default CI system or personal virtual machine.

The next logical step was to run the configuration without using the "-gcov" feature and run the build. As was expected, no errors at the building phase occurred, but the failure happened at the testing phase. That could be either a wrongly configured system, flaky test, or building/configuration error. Thus, running the standard build would be the correct way to see what type of errors are occurring.

However, the project with the implementation of the automated code coverage instance was postponed because QSR-related tasks and the feature release of the Qt 6 has taken priority.

A different report was created for the QSR project, which has shown the accurate data which developers will use to make the product even better.

5.7 WEEK 32

The week began with the news about the upcoming Qt 6 release. Since it has become the "number one priority", the implementation of the automated code coverage continuous integration system was postponed. To begin with, it was decided to generate the report using the following configurations: Ubuntu 20.04, qmake, default configuration parameters, and LCOV to convert the data to the HTML format. It was also decided that Qt version 5.15.0 would be a baseline to compare the results between Qt 6 and Qt 5.15.0.

Even though starting from Qt 6.0, it uses a different building system (CMake), it was decided to make the first report using qmake to check the raw comparison. The CI team provided a new machine with Ubuntu 20.04, and the next step has been to run the provisioning scripts for that.

Firstly, it was decided to collect the 5.15.0 data. The reason to make it a baseline is to compare the following data with some reliable, stable source, and using the previous building system (qmake), so that was the target for the company to track changes comparing these two versions. Both Qt 5.15.0 and Qt 6.0 were using the same machine, the same configurations, environmental variables to exclude that differences were caused by the environment itself and not the code. Since the new device was used for the given task (Ubuntu 20.04), provisioning scripts had to be executed which changed the version of the GCC, G++, and GCOV to 9.3.0 as well as many different installations, but this one is the most important. After the provisioning is done, the default routine goes as follows:

1. Configuration of the Qt with the given parameters
2. Build the Qt.
3. Run the tests.
4. Collect code coverage results.

The first three steps went as they should but collecting code coverage results was not successful. The only things that were changed comparing to the previous code coverage runs are the Ubuntu version and the GCC version. Ubuntu version 20.04 was selected as the primary OS for that task; thus, downgrading the GCC version was the only logical way.

5.8 WEEK 33-34

As the continuation of the previous week, the first thing that was done is to downgrade the version of the GCC from 9.3.0 to 9.1.0, which was tested with LCOV before. Since changing the GCC does not change the default process of the building and testing (steps to reproduce), the results were ready, and data was collected for Qt 5.15.0. The same routine came to the Qt 6.0. However, the folder structure changed slightly, and not all the modules were ready for release yet. As a result, the data was collected for the essential modules only using qmake as the building tool. The comparison between the versions was reported to the development team inside The Qt Company.

One of the developers noted that qmake data is helpful; however, since the company plans to change the building system from qmake to CMake, the Qt 6.0 data should be collected using CMake.

For CMake, steps are a bit different compared to the standard build using qmake. Since it was the first attempt gathering the code coverage data, it was decided to try with a single module first (Qt base). A few tries went wrong for the following reasons:

1. The building process is different. At first, the configuration is done, as is shown in figure 20. Even though it works and provides the output, the configuration should be done with CMake file itself.



```
qt@linux:~$ ./configure -cmake -R . -developer-build -nomake examples
```

FIGURE 20. The wrong configuration process that was used for the first few attempts to build the Qt 6.0.

2. Enabling the code coverage by exporting the CXXFLAGS did not provide the needed effect; thus, it must be done manually (will be explained later)
3. Configuration (using "-cmake") should be done inside the qtbase folder itself (not inside the parent folder as it used to be for qmake)

After several unsuccessful attempts, the correct way was found, and the steps to reproduce are:

1. Start the configuration with CMake file itself, not the usual way. Figure 21 shows the command that needs to be executed. This will create the CMakeCache.txt file and will do the configuration according to the given arguments.

```
qt@linux:~$ cmake -GNinja -DFEATURE_developer_build=ON -DBUILD_EXAMPLES=OFF
```

FIGURE 21. The correct configuration process for the Qt 6.0 using CMake

2. Change the "CMakeCache.txt" file to include the code coverage flags. While running the actual build, firstly, this file is checked. If some changes occurred compared to the default file, it will re-run the configuration (and save the edited parameters) and then execute the building itself. At this stage, configuration flags should look as it is shown in figure 22.

```
81 //Default glags used by the CXX compiler during all build types.
82 //CMAKE_CXX_FLAGS:STRING=
83
84 //Flags used by the CXX compiler during all build types to enable code coverage.
85 CMAKE_CXX_FLAGS:STRING=-g -O0 -Wall -fprofile-arcs -fptest-coverage
86
87 //Default Flags used by the CXX compiler during DEBUG builds.
88 //CMAKE_CXX_FLAGS_DEBUG:STRING=-g
89
90 //Flags used by the CXX compiler during DEBUG builds to enable code coverage.
91 CMAKE_CXX_FLAGS_DEBUG:STRING=-g -Wall -fprofile-arcs -fptest-coverage
__
```

FIGURE 22. Changing the CXX flags to enable code coverage of the module.

3. Start the build by running the commands in figure 23.

```
qt@linux:~/qt6/qt5/qtbase$ cmake --build . --parallel
```

FIGURE 23. Starting the build using CMake.

4. Start the tests by running the "ninja test" command.
5. Collect the code coverage data using LCOV.

While building and running the tests, no errors occurred, but LCOV could not collect the data because CMake uses GCC 9.3.0 still even though the primary GCC version of the machine (which was set before) is 9.1.0. From this, to get the code coverage results GCC of the system should match the version that it was built with. However, upgrading the GCC back did not have any impact,

and it was decided to try to upgrade the version of the LCOV since the GCC 9.3.0 data is incomparable with the LCOV 1.14.

According to the official LCOV website (3), the latest LCOV version is 1.14. Another way to collect the data is by using the FrogLogic Code coverage tool.

The FrogLogic tool had some problems with gathering the data as well (it was spotted in the different projects earlier while collecting code coverage results for the entire Qt, not the separate modules like QSR). This means that LCOV was the only proper way to do so at the moment. Although the official version was still 1.14, the Github version for the LCOV 1.15 was released the same week 33 (13).

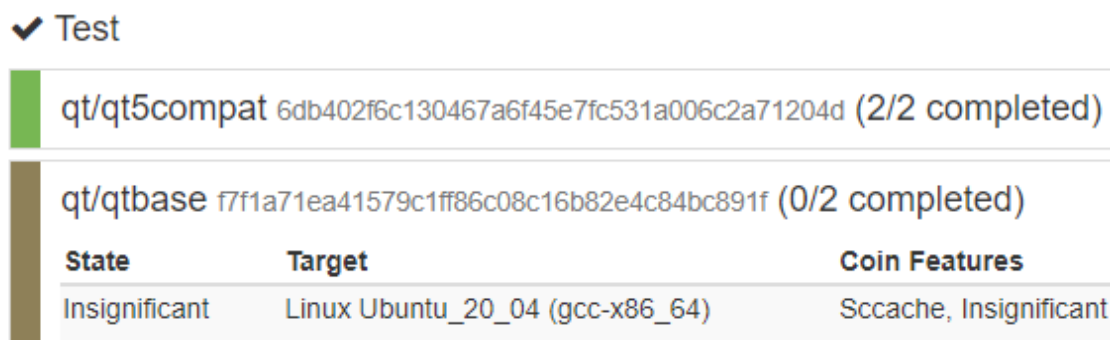
Returning the GCC to 9.3.0, upgrading the LCOV to v.1.15, and re-running the build did not produce any error at the start, so the last step is to run LCOV and check if any errors would appear at this stage. As it was expected, it helped, and the data for the Qt 6.0, only Qt Base module using CMake was collected, filtered, and shown to the developers.

5.9 WEEK 35-37

A new task was introduced in week 35. Because the Qt 6 release was coming and help with spotting building/testing errors was required. Before this stage, the default coin system had only 2 cases of the different runs: green – successful integration, red – failed integration. Since week 35, an additional parameter was added, which is an "insignificant" integration case using the brown color. This feature is helpful since it keeps the integration running even if failures appear during the building or testing phase. Thus, the process of finding bugs and failing tests would be more straightforward, and they could be reported and fixed as soon as possible. At the first stage, it was decided to track Ubuntu 20.04 and openSUSE with the dev. version of the Qt (latest available release, Qt 6 Alpha/Beta versions) and 5.15 since these two operating systems and framework versions are the most important ones. For each operating system, corresponding Jira tasks were created. Each of these tasks is connected to the subtasks (reported bugs) that need to be fixed.

Releasing the Qt 6 was the most important priority at that time, and the effort from most of the different teams was crucial for that project. For the insignificant case, the job was to report either bugs are appearing during the build phase (P0, blocker, needs to be fixed as soon as possible) or bugs during the tests (P1/P2). Figure 24 shows passed and insignificant coin runs.

✓ Test



State	Target	Coin Features
Insignificant	Linux Ubuntu_20_04 (gcc-x86_64)	Sccache, Insignificant

FIGURE 24. Passed and insignificant coin runs.

The insignificant feature was applied only to the Qt 5.15 and Qt 6.0 dev branches so it would not affect the previous Qt versions. If the insignificant feature was enabled, but no bugs or errors appeared during the build or tests, it will be shown as passed (green), but if the human interaction is required to report/check – brown.

As a result of the first three weeks with the "insignificant" task, five bugs were reported with the connection to the main task of the Ubuntu, two bugs for the QEMU, and much more had JIRA tickets already; some were fixed right away. Ideally, all the bugs for the given versions should be connected to the main task, which would make the tracking easier and keep the things in one place; however, the main goal is to fix them, which is ongoing. This "insignificant" task helps the company to make the product even better before the release. Continuation with the task is essential, and that someone keeps an eye on the insignificant runs would significantly improve the quality and the timing of the future release.

5.10 WEEK 38

As a logical continuation of the week 33-34 task to collect the information for the Qt6 using LCOV should be done not only for the QtBase module (even though it is the most important one) but for the entire framework. Since the version of the Qt6 is still in pre-alpha, it does not support top-level builds, and modules have to be built one-by-one starting from the QtBase. The way to build and enable collecting the LCOV data is slightly different from the previous qmake runs. After cloning and initialization of the repository, the steps are the same as it has been for the week 33-34 (until the test phase).

After the QtBase is ready, the next task is to do the same for all the different modules included/porting already to the Qt6 pre-alpha version. QtBase should be done first since all other modules are dependent on it. For example, to build the QtDeclarative, the following steps should be done:

1. Inside the folder, run the command from figure 25.

```
qt@linux:~/qt6/qt5/qtdeclarative$ ../qtbases/bin/qt-cmake-private
```

Figure 25. Command to configure the QtDeclarative module with the parameters used to configure the QtBase.

2. To enable code coverage, CXX flags need to be enabled inside the CMakeCache file (the steps were explained above).
3. Initiate the building process.

The only difference is in the first step. After building all the modules, the next step is to execute the test, which should be done inside each folder one-by-one with the command "ninja test" and collect code coverage information using LCOV, which is the same as it was for Ubuntu 18.04.

As a result, code coverage report for the Qt 6 using CMake was created. Since Qt 6 is different, some more modules or additions were added, some modules were deprecated/changed. To see

the actual situation better, it was more efficient to create an excel file to compare the corresponding folders or modules. The general information might not be useful since lots of changes happened.

For example, the general code coverage of the entire Qt 6.0 is higher than the Qt 5.15.0. However, it does not mean that all of the modules are like that because a different essential module coverage might be lower, so this module should be investigated more.

Thus, this information could be valuable for the developers to cover the weak spots of the product.

5.11 WEEK 39-41

Since the target of the company is to make the first release of the Qt 6 either the same or even better in terms of code coverage, the task is to check the future releases (alpha, beta, release candidates) and how does code coverage is changing with that. Meanwhile, the work with the insignificant coin runs and reporting the bugs was continuing in parallel. Ideally, each release should have improved coverage comparing to the previous one. At this point, three code coverage reports for the Qt 6 were collected during these three weeks, and the increasing trend was easily spotted.

The good thing about insignificant runs and code coverage as the pair is that fixing the insignificant failed tests improves the code coverage. On the other hand, a higher coverage of the module could lead to fewer failed tests/bugs in the future. Nevertheless, this could be misleading and cause the "survivorship bias" since some modules (for example, Qt Quick 3d) are tested not with the auto-tests but with the Lancelot Graphics Testing. Hence, even though some modules have lower code coverage percentages, it is still tested, but differently.

The given weeks mainly included generating code coverage reports which include the most recent patches/releases or reporting the insignificant build/test failures. In addition to the Ubuntu and SLES, Windows insignificant flag was included.

Even though nothing new happened during these weeks, the routine quality assurance work before the release is valuable and helpful.

5.12 WEEK 42-44

After releasing the first beta version of Qt 6, a significant bug appeared that caused several tests to fail and drastically decreased the code coverage percentages. The reported issue was a blocker both for the future release and code coverage results in general since the bug was affecting the most crucial essential module to have about a 5% decrease in the coverage. The development team is responsible for fixing that bug.

Returning to the QSR project, it is much easier to automatize the task since the job is done not by the coin job but with the cron task, which triggers the script. Cron is a time-based job scheduler in Unix-like computer operating systems that helps run tasks periodically at fixed times, dates, or intervals. For the QSR project, a server runs the script that creates a FrogLogic report for the main part of the code. Nevertheless, the entire code coverage report should track the changes inside the entire structure of the code. The bash script that is triggered by the cronjob is defined as follows:

1. Include the codecoverage.pri file (.pri stands for project include file) to the .pro files. The first one (.pri) enables code coverage (by including the feature in the .pro file), the second one (.pro) is the building instructions (profile) of the module. Thus, while the build will be executed, the coverage will be included as well. This is done for one module inside the project and will be extended to all the .pro files inside the root folder of the Qt Safe Renderer.
2. Execute build and tests.
3. Collect the code coverage data from the parent folder.
4. Copy the results to the fileserver with the folder name given as the timestamp to distinguish the results.

Theoretically, this is how the implementation of the automatization should be done. Still, since the gathering of the results does not take as much time as the code coverage data for the entire main Qt framework, this could be postponed and done manually. For now, the main tasks are keeping an eye on the insignificant coin runs and creating the code coverage report for the newer Qt 6.0 version after the critical bug would be fixed. On the other hand, helping the QSR project team is also vital since, as was stated before, the project is highly code coverage dependent.

Besides, Qt 6.0 beta v.1 and v.2 were released at this time. Even though the previously mentioned bug has not been fixed yet, the second version of the code coverage data was collected since the difference between the alpha and beta releases is significant. Some modules were excluded from the main repository and no longer supported, and some structural changes happened. The collected information was shown to the developers to do future investigations.

Later in the process, the bug was fixed, but a valuable lesson was learned - a single bug can stop the release and drastically reduce code coverage values. After the fix, code coverage results not only returned to the previous level but with every beta release the numbers were higher.

6 CONCLUSION

The main topic of the thesis is code coverage effectiveness. Using the methods and the way of showing the code coverage results, it was proven that code coverage data is valuable for the company. For example, after gathering the code coverage data for the QSR team, developers could easily spot the parts that were not tested that well, and, in addition to that, pointed to a bug that was not found before. Different teams also started to use that and improve the maturity and stability of the code. The additional purpose of the thesis was to show how code coverage helps to find weak spots of the code or how bugs affect the framework in general. From the perspective of the company, the results of the project were considered successful and meaningful: it helped to increase the test coverage (by finding which parts of the code were not executed by the test cases), as well as finding new bugs or finding how bugs were associated with the code coverage data (week 42-44).

Even after the thesis, code coverage will take a considerable part of the quality assurance point inside the company to prove that the product is functional, well-tested, and checked. For the entire framework, gcov tool will be used in the close releases, but for the variability, different measurements, and deeper analysis, FrogLogic CoCo is the tool of choice. In the future, it is planned to use FrogLogic tool for the entire Qt Framework and automate it.

Even though automation was chosen as the second primary goal initially, since the work was done in the RnD area, it was hard to predict which tasks would take priority in a month or two. Thus, the content was changing. The diary-based thesis was proven as the best choice for that. In the future, automatization can be considered as the task to make an automated code coverage report not only for the QSR project but for the entire Qt Framework.

REFERENCES

1. "10 Reasons Why Code Coverage Matters". Date of retrieval 01.08.2020
<https://codeburst.io/10-reasons-why-code-coverage-matters-9a6272f224ae>
2. The Qt Company website. Date of retrieval 01.08.2020
<https://qt.io/company>
3. LCOV official website. Date of retrieval 19.11.2020
<http://ltp.sourceforge.net/coverage/lcov.php>
4. GCC Wikipedia page. Date of retrieval 10.12.2020
https://en.wikipedia.org/wiki/GNU_Compiler_Collection
5. Using the GNU Compiler Collection (GCC). Date of Retrieval 10.12.2020
https://gcc.gnu.org/onlinedocs/gcc-5.3.0/gcc/G_002b_002b-and-GCC.html
6. Introduction to gcov. Date of Retrieval 10.12.2020
<https://gcc.gnu.org/onlinedocs/gcc-5.3.0/gcc/Gcov-Intro.html>
7. FrogLogic website. Date of retrieval 12.12.2020
<https://www.froglogic.com/coco/>
8. qmake manual. Date of retrieval 12.12.2020
<https://doc.qt.io/qt-5/qmake-manual.html>
9. CMake website. Date of retrieval 12.12.2020
<https://cmake.org/>
10. Brief Description of gcov Data Files. Date of retrieval 07.06.2020
<https://gcc.gnu.org/onlinedocs/gcc/Gcov-Data-Files.html>
11. Squish Coco. Instrumentation. Date of retrieval 01.08.2020
<https://doc.froglogic.com/squish-coco/latest/tutorial.html#sec24>
12. Qt Safe Renderer Overview. Date of retrieval 01.08.2020
<https://doc.qt.io/QtSafeRenderer/qtsr-overview.html>
13. Linux-test-project/lcov github page. Date of retrieval 14.08.2020
<https://github.com/linux-test-project/lcov/releases/tag/v1.15>
14. Difference between Squish Coco and gcov/LCOV. Date of retrieval 16.02.2020
<https://www.froglogic.com/coco/faq/#Miscellaneous>
15. P. Piwowarski, M. Ohba and J. Caruso, "Coverage measurement experience during function test," Proceedings of 1993 15th International Conference on Software Engineering, 1993, pp. 287-301, doi: 10.1109/ICSE.1993.346035.

16. Code Coverage for Suite Evaluation by Developers. Date of retrieval 18.04.2021
<https://core.ac.uk/download/pdf/192426648.pdf>
17. I. Ahmed, R. Gopinath, C. Brindescu, A. Groce, and C. Jensen, "Can testedness be effectively measured?" in FSE, 2016, pp. 547–558. Date of retrieval 19.04.2021
18. L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in ICSE, 2014, pp. 435–445. Date of retrieval 16.04.2021
19. A. S. Namin and J. H. Andrews, "The influence of size and coverage on test suite effectiveness," in Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ser. ISSTA '09. New York, NY, USA: ACM, 2009, pp. 57–68. Date of retrieval 16.04.2021