BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

TOOL FOR TEST RUN OF ROBOTIC TESTING SYSTEM

NÁSTROJ PRO ZKUŠEBNÍ BĚH ROBOTICKÉHO TESTOVACÍHO SYSTÉMU

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR — **SAMUEL SLÁVKA**
AUTOR PRÁCE

SUPERVISOR — **PETR PERINGER, Dr. Ing.**
VEDOUCÍ PRÁCE

BRNO 2020

Brno University of Technology
Faculty of Information Technology

Department of Intelligent Systems (DITS) Academic year 2019/2020

Bachelor's Thesis Specification

Student: **Slávka Samuel**
Programme: Information Technology
Title: **Tool for Test Run of Robotic Testing System**
Category: Software analysis and testing

Assignment:

1. Analyze the robotic user interface testing system *RQA* (Robotic Quality Assurance) in YSoft company. Study the terminology and basic testing methods.
2. Design the system for test run of various scenarios of robotic testing system. The goal of designed system is checking the availability of all data *RQA* needs for performing the real tests. Design the suitable presentation methods of missing data and information about test run to the user.
3. Implement the system in .NET environment. Test the system functionality using well chosen examples.
4. Evaluate the project results and propose possible future improvements.

Recommended literature:

- Patton R.: "Software testing", 2nd edition, Sams Publishing, 2005.
- Other sources, according to recommendation of consultant (Pernikář Aleš, Ysoft).

Requirements for the first semester:

- First two points.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor: **Peringer Petr, Dr. Ing.**
Head of Department: Hanáček Petr, doc. Dr. Ing.
Beginning of work: November 1, 2019
Submission deadline: May 28, 2020
Approval date: May 18, 2020

Abstract

This thesis deals with the problem of creating tests in the Robotic Quality Assurance testing system. The goal is to create a tool for a test run of tests in the RQA system for the company Y Soft. The main focus was on the possibility of continuing after encountering an error and circumventing communication with the tested devices. When a missing element is encountered, the element is generated and marked as missing, allowing the test to continue, and the tester to correct the problem after test finishes. The mock access layer ensures the avoidance of communication with hardware. The resulting tool can run tests without the tested device and saves time spent on repeating tests.

Abstrakt

Táto práca sa zaoberá problémom vytvárania testov v testovacom systéme Robotic Quality Assurance. Cieľom je vytvoriť nástroj pre testový beh testov v systéme RQA pre spoločnosť Y Soft. Hlavným zameraním práce je umožnenie pokračovania testu po zistení chyby a vyhýbanie sa komunikácii s testovanými zariadeniami. Po výskyte chýbajucého prvku sa prvok vygeneruje, jeho výskyt sa zapíše a test pokračuje ďalej. Vytvorenie simulovanej prístupovej vrstvy sa používa na zabránenie komunikácie s hardvérom. Výsledný nástroj umžňuje spúšťať testy bez zariadenia a šetrí čas strávený opakovaným testovaním.

Keywords

dry run, mock test, unit test, castle windsor, test design, mock class dependency injection, robotic testing, ASP.NET Core

Kľúčové slová

dry run, mockový test, unit test, castle windsor, návrh testov, vkladanie závislosti, robotické testovanie, ASP.NET Core

Reference

SLÁVKA, Samuel. *Tool for Test Run of Robotic Testing System.* Brno, 2020. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Petr Peringer, Dr. Ing.

Rozšírený abstrakt

Táto práca sa zaoberá problémom vytvárania testov v testovacom systéme Robotic Quality Assurance (RQA). RQA je vytvorený firomu Y Soft. RQA slúži na testovanie zariadení s použitím dotykovéj obrazovky, hardverovými tlačítkami a aj prídavnými senzormi. RQA je reobotický testovací systém, ktorý ovláda zariadenie pomocou robotického manipulátora. Systém využíva kameru ako zdroj spätnej väzby. Spätná väzba je vo forme obrázku aktuálneho stavu na zariadení. Tento obrázok sa pomocou rozpoznávania obrazu priradí určitéj obrazovke uloženéj v databáze. Test prebieha postupnym vykonávaním príkazov na hardvéry a podla zmeny stavu zariadenia sa kontroluje ich úspech. Pri neúspešnom testovom príkaze sa test ukončí. RQA je určené na ovládanie robotického systému zo vzdialeného testovacieho systému.

Výsledkom práce je nadstavbou funkčného testovacieho systému. Cielom je spustiť testový beh testu, ktorý vykoná všetky testové akcie v aplikácii ale obíde funkcionalitu robota. Táto nadstavba sa musí vyhnuť komunikácií s robotom a kamerou a vytvoriť spätnú väzbu od kamery. Spätná väzba od kamery sa rieši simulovanovou prístupovou vrstvou ku image processing. Podla typu príkazu a predchodzích akcií sa vytvorý odhad aktualnej polohy. Podla aktualnej polohy sa overuje správnosť postupnosti testovývch príkazov. Vyhýbaniu komunikácie sa dosiahlo taktiež pomocou simulovanej prístupovej vrstvy. Simulujú sa len vrstvy na nižšioúrovňových službách, ktoré priamo ovládajú robota a senzory. Výsledný testový systém musí umožňovať aj normálne spúšťanie testov na hardvéri aj spúšťanie testov v testovom móde, tieto módy by si nemali prekážať a spusťenie módu závisí iba od testera. To sa dosiahlo simulaciou spodných vrstiev systému. Filtrovanie akcii sa použiva na zrýchlenie odpovedí testovému systému. Každý pokus o prístup do databázy sa musí zaznamenať, aby tester mohol skontrolovať výsledok testového behu testov. Po výskyte chýbajucého prvku v databáze sa prvok vygeneruje, jeho výskyt sa zapíše a test pokračuje ďalej. Chýbajucí prvok sa vytvorý pomocou získaných informácií počas behu testu. Toto vytvorenie umožní testovému behu testu pokračovať aj po najdení chyby a teda nájsť viacero chýb počas jedného behu. Výstupom testového behu testu je dokument so zoznamom všetkých prístupov do databázy a kontrol správanie prvkov, spolu s ich výsledkami. Implementácia je v jazyku C# s využitím frameworku ASP.NET Core. Framework umožňuje komunikáciu s testovým systémom.

Testovanie bolo prevedené s porovnávaní rýchlosti testového behu testov a normálneho. A aj s porovnávaním rýchlosti reálneho využitia aplikácie na tvorbu testov. Pri testovaní tvorby testov, sa merala rýchlosť najdenia viacerých chyb v jednom teste. Výsledný nástroj umžňuje spúšťať testy bez zariadenia a šetrí čas strávený opakovaným testovaním. Nástroj sa bude používať pri vytváraní testov, na zrýchlenie overovanie správnosti testu. Tester buďe používať oboje aj testový mód pre overenie informácií a normálny móe pre overenie behu ha hardvéri.

Tool for Test Run of Robotic Testing System

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Peringer Petr, Dr. Ing. The supplementary information was provided by Mr. Pernikář Aleš. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Samuel Slávka
May 28, 2020

Acknowledgements

I would like to express my gratitude to Petr Peringer, for his guidance and valuable advice. I also wish to thank Aleš Perrnikář for his support and patience.

Contents

Chapter 1

Introduction

Software development is not a precisely defined process, and there are as many correct approaches to solving any given problem as there are wrong ones. To distinguish whether the chosen approach is correct, developers test software. Software testing is an essential part of software development. The goal of testing is not only to check the required properties but also the behavior during unusual or even undesirable states. In more sophisticated systems, it is impossible to cover all possible inputs, outputs, and inner workings with tests due to the number of possibilities. Therefore during test development, it is essential to create tests for as many relevant scenarios as possible, which highlights the need for efficient test creation.

This thesis aims to extend the Robotic Quality Assurance (RQA) testing system developed by company Y Soft. Robotic quality assurance is a system for testing embedded devices. The system's main components are camera, robotic hand, database, and control application. Execution of a test currently requires a fully working and calibrated system, and the test creator has to wait for the hardware execution of test commands. Tests can fail for hardware error, which skews results and slows test creation even more. Alternatively, tests can fail for error in test design; checking test information before the test run would be beneficial for the test design process.

This thesis aims to simplify and accelerate the process of creating tests. The resulting application does not require a connection to the physical devices for running a test. If a connection is available, the application checks for calibration, and records the results. However, even when a connection is available, and devices are correctly calibrated, they are not used. Test creators can have faster feedback from their tests and be much more productive, by not depending on physical devices. Currently, there is no existing tool for a test run of tests in the RQA system.

The goal is to allow the test creator to create tests without the requirement of a physical device. The new RQA extension will allow the testers to execute tests without waiting for physical test execution and environment preparation.

In the following chapters, I describe software testing and its different types and uses. Afterward, I describe the Robotic Quality Assurance testing system and my design and implementation of its improvements. In the end, there are comparisons between the new and old system version.

Chapter 2

Principles of software testing

Software testing [7] is a process, or series of processes ensuring that the tested software behaves as it is supposed to and does not behave undesirably. All software starts as an idea, which gradually turns into a product specification. The product specification is formed to suit customers' wishes as best as possible, and it is an agreement between customer and software engineers. Depending on the purpose of the software, it can be dynamically changed during development or stay nearly identical to first drafts. The software specification creation process starts with the formation of tests, at the beginning in abstract form. The finished product is only accepted if it fulfills customers' wishes. Therefore developers try to create tests to monitor the wishes.

Testing has many more uses than just verifying product functionality. It is an integral part of the development process. The goal of software testing is finding bugs. A bug [7] is a problem that occurred during the creation of software. However, this is too general definition. A bug is what happens when software functionality differs from its specification but also when it differs from its purposes and intended usability. Finding bugs and correcting them is the central part of testing. Several studies [9] have shown that the origins of most bugs can be traced to software specifications. For this reason, finding bugs during the creation and earliest stages of development is essential. During specification design, a correction of a bug is just a matter of discussion with the customer. However, when a bug in specification gets implemented and released, not only can it be expensive to correct, it can, in some cases, become even dangerous.

Testing software completely is, in most cases, impossible. To create tests for all possible inputs, outputs, and inner workings of software, the number of needed test cases becomes too large to be usable, and resources required to maintain those tests rise. That is the reason for the importance of proficient test design and the reason for which random-input test creation [9] has the lowest efficiency. The two main approaches to test creation are Black-Box and White-Box.

Black-Box is an approach [7] that omits the inner workings of software and mainly focuses on inputs and outputs. This method considers the software as a black box. The box covers software, and testers can only see what comes into the box and what comes out. The goal is to find combinations of inputs and outputs which contradict its specification.

White-Box approach [7] focuses on the inner workings of the software. By examining the inner workings of code and trying to determine parts that are most prone to having bugs. The tester designs tests with precise paths trough the software by choosing inputs that should lead to these paths. White-box testing often utilized in unit tests 2.3. The testing method is utilized during integration or system tests, but on the higher testing

levels, the white box method is not sufficient. During software testing, a mix of White-Box and Black-Box is used most often. Each has a specific role, and neither is perfect. White-Box can omit missing or unimplemented requirements, and Black-Box can ignore internal errors.

The goal of testing is to ensure that the delivered software fulfills customer requirements as best as possible. Even though the requirements can not always be tested or delivered, testing enables testers to know the software limitations before the clients. This prior knowledge gives developers a chance to solve problems before they become hard to solve or even unsolvable.

2.1 Good test design

A good test [14] is a test that leads the tester to more knowledge about the tested system. It provides enough information, and its failure should be against users' requests. For the creation of a good test, the tester usually cannot just sit and write the first thing that comes to his mind. There are multiple approaches to test creation ranging from creating tests before the software development starts to skipping testing until the end of development. Tests are needed for reliable software and even more so good tests.

To design a good test, we first need a test case. A good test case [7] is a specific software scenario that has a high probability of having a bug and should cover other possible test cases. To create tests that cover the most considerable amount of test cases, we can use Equivalence classes. Equivalence class [7] is a set of test cases that should act similarly. When one test from the equivalence class fails, others should also fail, and the same applies to pass. It can be created by examining all influencing elements in the system and grouping them by similarities.

The problem is sometimes even when theoretically test cases should be in the same class, in real-world tests can return different results. This inconsistency is solved by Boundary-Value testing. Boundary-Value testing works under the assumption that most errors occur at the edges of Equivalence classes, and tests are created to cover those edge cases. Equivalence classes with Boundary-value testing still sometimes fail to cover some combinations of system behaviors. A tester should also use the knowledge of the system and create tests covering these combinations or just potential error cases. Which by itself does not ensure the software success. It makes the software more durable and makes the development process easier.

2.2 Software testing phases

To enable a structured approach to testing and keep a high quality of tests. Testing is divided into multiple phases [2]. Each of these phases has a different level of scope. They are ranging from testing small pieces of code to testing requirements of the customer. There are multiple phases during software creation, but not all of them concern developers. Testing is segmented to allow working on more complex systems whose development is also segmented. Each development stage requires tests specifically directed for the goal of the stage. Main tests concerning developers are:. Main tests concerning developers are:

- Acceptance tests are on the highest level, they check against customer requirements. Depending on the nature of the project, these tests do not always concern developers.

The goal of acceptance tests is to ensure the customer receives what was requested. The utilization of acceptance tests is the highest before the release of the product.

- System tests. These tests monitor the behavior of the system as a whole. Usually, they are not used for bug finding but non-functional requirements such as speed, reliability, or even coping with external interfaces.
- Integration tests test how well smaller parts of software work together. Testers use integration tests when development on a module finishes, and the tester needs to verify that the module can still cooperate with other modules.
- Unit tests are at the lowest level. Unit tests are typically automated tests verifying that smaller parts of the software(units) are working as intended in isolation. Typically testing is in smaller scope like methods or classes, but unit tests can also be used on testing bigger units like subprograms or a bit larger parts.
- Regression tests search for newly created bugs in software after a change. The old version of the software should have the same results of regression tests as the newer version. Regression testing starts after the software is finished and continues while the software is maintained.

2.3 Principles of unit testing

A unit test [8] is an automated code using smaller parts(units) of software and checking their behavior. It is entirely independent of outside variables and other tests. Its inputs and outputs are fixed predetermined values. Unit testing provides a quick way to verify changes in code.

A unit test should be a quick test of the functionality of a part of the software. This means the test needs to have a prepared environment for its task, and after the test finishes, this environment needs to be put into an initial state to prevent its results from interfering with other tests. Tests can be run in any order, can run in parallel, and all runs should be completely independent. Input and all influences should have fixed values and behavior to help with the consistency of results. The results should be easily readable. More emphasis in terms of the amount of information is in failing tests than passing ones because the tester is usually much more interested in those results.

During test design, the more time it takes to make tests, the less effort is put in making them. The benefits of unit tests diminish with too much time spent on creating and maintaining them. An emphasis on creating tests that do not require maintenance in later parts of development makes them also more usable and desirable. Tests should always be available to everyone, and able to return quickly test results so that developers are incentivized to use them. Separation of tests into different test suites for further speeds tests up. When working on a smaller part of the software, there is no need to run all unit tests after each change, for quicker feedback the developer can just run his specific suite and after more changes can run them all.

During development after a change, unit tests are used to check whether the software still works and provides the developer with faster feedback than regular tests. This feedback helps in developing new features or making changes in older code. Unit tests are more maintainable than integration or system tests because of their small scope and predefined inputs and external influences.

Unit test implementation often uses some framework that allows setting a system environment for each test or suite specifically. Frameworks offer a structured way of creating tests, and an easy way to run and repeat tests, and a way to examine results. Tests can be and often are implemented without frameworks; however, using frameworks is usually more efficient.

Unit tests don't test how well the units work together nor how well the software copes with outside interference. However, they are a quick way to check the functionality of relevant parts of software after a change.

I used unit tests mainly for testing low-level handling of test actions access layer to the database. Each central part of the test process is located in its test class with a specific setup. Unit tests are implemented in the Nunit framework. A test during the setup process creates an in-memory database and, in the first stages, filled up with appropriate values. In memory[6] database is a database system primarily stored and working in the main memory. After each test, the database and all values are scrapped.

2.4 Dependency injection

Dependency injection [13] is a technique that separates the client's behavior from the creation of its dependencies. It enables a higher degree of loose coupling [10] during software creation. Loose coupling defines the level of independence between modules. By making modules more independent, they can be easily modified and also are simpler to maintain.

It works by passing constructors as parameters to objects. Those objects are not bound to the implementation of their dependencies; neither are dependencies bound to the objects. They both use interfaces that define how and which services the objects can use. The prominent roles during dependency injection are service, client, interface, and injector.

A service is an object that is depended on by the client. It implements the interface and performs the client's requests. The client uses service by using defined ways in the interface. The client is independent of the service's implementation but depends on abstractions in the interface. The interface defines how the client and service can interact. It describes the form of their interaction, not the implementation. The injector constructs services and connects them with clients. Service is constructed with the first call of his dependant and provides the service with its requirements. Injector considers both clients and services as clients and services at the same time to allow more complex dependency chains.

The advantages of dependency injection are its flexibility, testability, and maintainability. It provides a flexible way to change services. As long as a service implements an interface, it is possible to inject it. Service injection makes unit testing easier. During unit testing, clients can use a mock version of services that respond in a predefined way. The resulting software is easier to extend and maintain. Classes have well-defined purposes and are separate from each other. This separation allows parallel development of software and easier changes to existing code. Dependency injection enables late binding [13], an ability to interchange services during runtime.

The main disadvantages of dependency injection are added overhead for injecting and class creation, increased complexity, and worse readability of code. It can lead to a dependency on the dependency injection framework. It is hard changing frameworks, and when this change takes place, a large part of code needs to be modified or entirely rewritten. Dependency injection moves complexity from classes to their linkages, and this makes it harder to test as errors happening in runtime.

There are multiple ways of implementing dependency injection. Constructor injection [13] is most frequently used design pattern. During constructor injection, when a class needs a dependency, it is passed as a parameter in its public constructor. The class stores dependencies in variables for further use.

- Property injection stores dependencies on public properties. Those properties are assigned outside of the constructor, which provides a bit more flexibility. Usually used when a class has local default value and dependency is optional.
- Method injection - Passes dependencies as arguments of methods on their calling. Usually used when dependencies change between calls of a method.
- An ambient context makes dependency available to every customer. Rarely used only to provide a single instance of strongly typed dependency.

Service lifetimes [11] defines the time that the object has allocated block of memory. Transient lifetime defines service that is created for each dependant separately. Transient lifetime services cannot store state, and each creation is a separate entity. In scoped lifetime service is instantiated once per user connection. Singleton lifetime service is created for the first time they are requested and is used by all users. Both scoped and singleton can store state and other information.

Dependency injection containers

A container is a library implementing dependency injection patterns. Containers no mandatory for dependency injection, but they shift the main development focus from implementation of the project environment to the project itself. In some cases writing containers from scratch can be beneficial for projects in decreasing overhead of loading classes, but more often than not, this increase is negligible compared to the effort needed for their creation.

Robotic Quality Assurance uses the Castle Windsor container. Castle Windsor is a part of open source .NET project Castle. It is one of the oldest and most popular [13] inversions of control providers. It provides a broad spectrum of API functionality. Castle Windsor uses more in-depth described in section 5.2

Chapter 3

Robotic Quality Assurance

Robotic quality assurance is a tool for developers and quality assurance engineers to perform repetitive and time-consuming tasks or tasks not performable by a person. RQA interacts with a device using a touch screen and hardware buttons. RQA can autonomously perform test scenarios on a device.

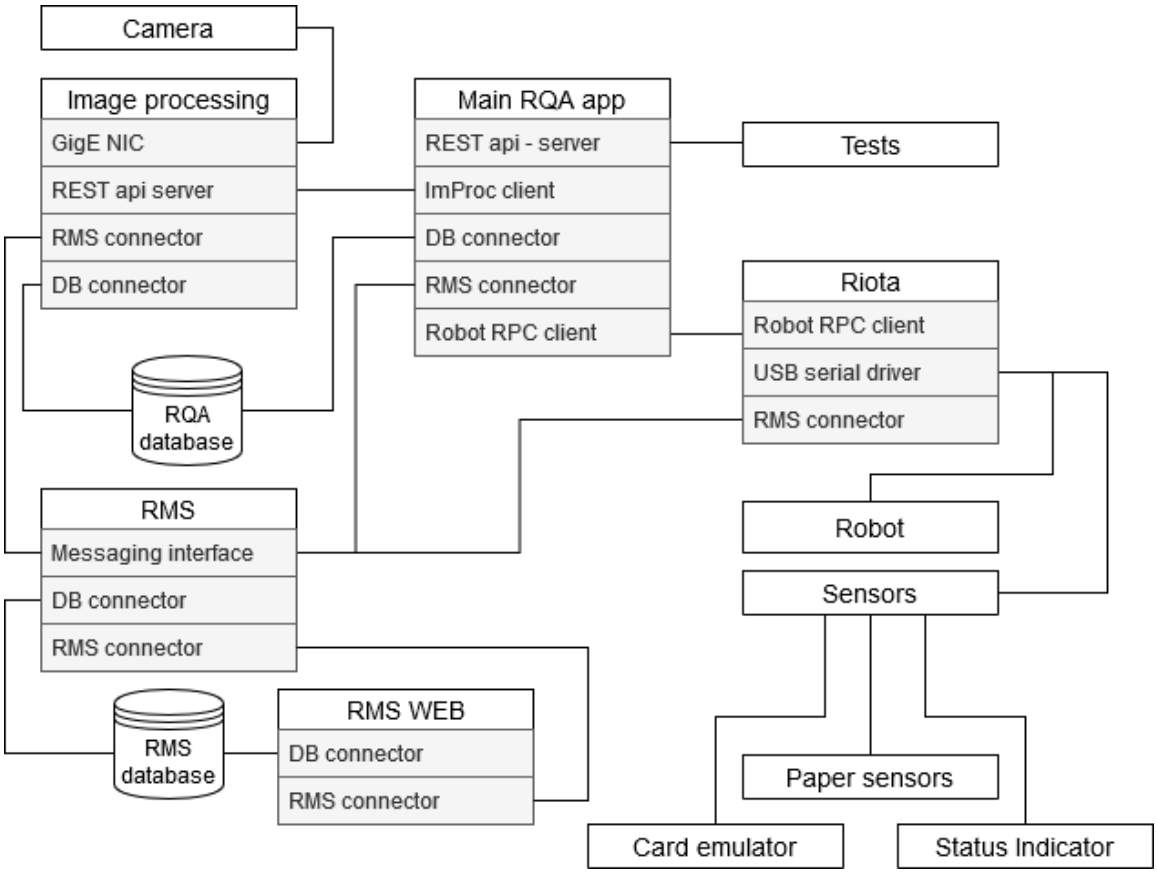

Figure 3.1: Robotic Quality Assurance architecture

Figure 3.1 shows the architecture of Robotic quality assurance. RQA consists of several parts. The central part is a robotic manipulator that interacts with the assigned device.

Another part of RQA is a camera with image processing software. The camera provides the only feedback for the system. The camera aims at the display and, as the feedback, provides an image of the currently displayed screen that is further handled by Robotic Quality Assurance. It is also able to detect the state and position of buttons and text on the screen. To be able to detect these, image processing needs to finish training by examining saved images of screens of the device successfully. The images are manually taken by tester before testing.

Tests are written with high-level keywords using Robot Framework. Robot Framework [3] is a Python-based test automation framework, it provides simple, easy to read keywords written in a tabular format. During test, the keywords send requests to the RQA app, which appropriately handles them.

RMS is a tool for monitoring testing system state and connecting devices to the RMS database. The RMS database is a `PostgreSQL` database, that stores all test and device information. It stores pictures of screens and definitions of device behavior.

RMS web application is a web-based tool that provides an interface for management and creation of information in the RMS database. It also has the functionality to control cameras and robots, and testers can view a live feed from the camera and control robot for simpler tasks. During test creation, all test information is configured through this application.

3.1 Tested device

Robotic quality assurance is designed for testing the functionality of devices. Those devices, in most cases, provide services for printing, scanning, copying. They can be controlled by hardware buttons or software on the touch screen. RQA needs some information about a device before testing, such as display dimensions, display type, and version of the software. The robot and camera are positioned in front of the device, allowing the robot to reach control elements and the camera to view the current state of the test and device. RQA also supports testing authentication through radio-frequency identification with employee card or username and login or with a PIN.

3.2 Robotic Quality Assurance peripherals

It is a robotic hand positioned in front of the device. It consists of five connected servomotors that end with a stylus. The manipulator is able to interact with capacitive and resistive displays but also with hardware buttons. It uses Riota as a connection to RQA app. RQA app utilizes gRPC for sending commands to the manipulator. GRPC is a remote procedure call system, allowing messages calling procedures on Riota. Riota monitors state of robot and sensors, and forwards communication between RQA app and the manipulator.

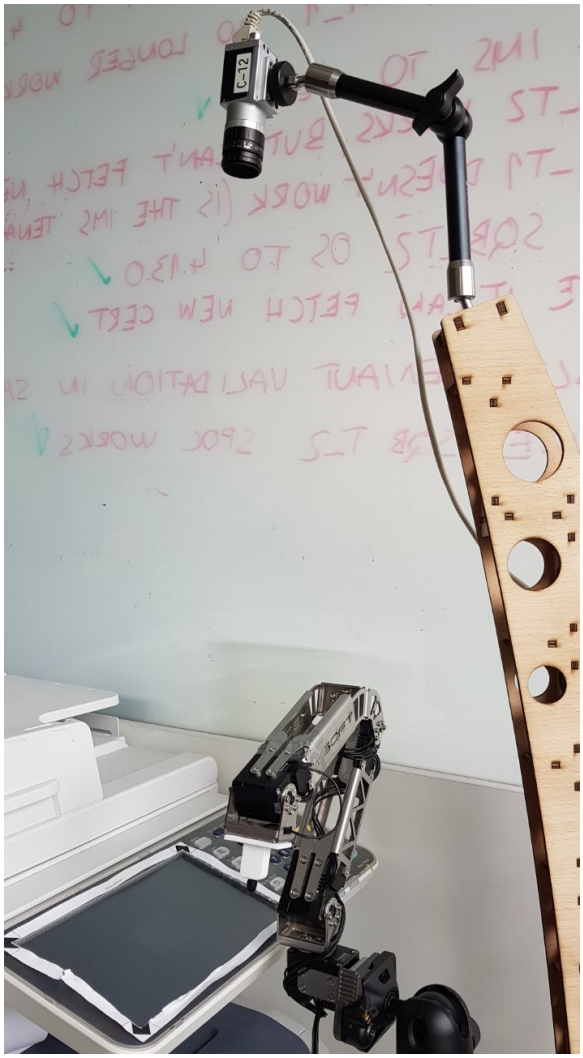

Figure 3.2: Robot with camera and tested device

The robot needs to be calibrated to the specific device position and display size before its use. While the manipulator can control both software and hardware buttons, they are defined differently. The hardware button position is physically measured on the device with coordinates starting in the bottom left corner on the display of the device. Software button position can be different on different screens, so the position is set in RMS web application by selecting places on appropriate screens and can also be found or identified by image processing.

Camera works as the only form of feedback of the device state. It is positioned above the robot and device with an unobstructed view of the screen. RQA asks after each physical action for the current screen, and the camera returns it to image processing to identify it. The camera needs to be calibrated for RQA to function correctly.

Card reader and emulator are enabling faster user authentication. If the device doesn't have a card reader, it can be externally connected. The card emulator is located on top of the reader, and emulates user swiping their card on the reader. By emulation, the user identification information is generated and received on the reader.

The paper sensor collects and provides information about printed papers. It can collect the number of printed papers, but it is also able to detect their size.

Image processing identifies pictures of the screen that the camera takes. Its main use-case is for identifying the current screen. But it can also identify active elements and position of buttons. It runs separately and is accessible via an HTTP request. Image processing needs to be trained on a suitable collection of screens to be able to recognize screens. This training can take a significant amount of time, depending on the device and test scope. Image processing provides the current state of test and result of actions.

3.3 Robotic Quality Assurance test structure

Tests are written with keywords using the Robot Framework. Robot Framework [3] is an open-source keyword-driven testing framework implemented in Python. It allows flexible and simple test creation and provides functionality to create new keywords. However, the RQA already contains a large amount of test keywords, so creation of new ones is uncommon.

Tests are organized in test suites. A test suite [1] is a collection of tests and other suites. Suites exist to enable tester testing smaller parts of functionality and have results more quickly. Suite scope can be specified for a single task, but also can be a large scope containing all tests for the device. Suites similarly to tests have unique names used for executing all tests in the suite.

If any keyword fails to execute the whole test stops execution and fails. Each test has its unique name, which works as an identifier used for executing tests and usually abstractly references tests function. Each keyword is written on a single line. Example of a keyword:

```
Set Dry Run
```

These keywords are sent to main RQA app in form of http messages. Example of message:

```
HTTP    GET /api/rqa/PropertiesLib/Set?DryRun=True HTTP/1.1
```

RQA app returns HTTP return value depending on the result of the task. It returns 200 on success and 500 on failure.

3.4 Test and robot configuration

Before running tests, RQA needs to be configured for specific scenarios. Robot, device, camera, and test configuration is stored in file `config.robot`. It configures IP addresses and authentication information of each service that tests use. It also contains additional variables describing test behavior and hardware properties.

During the mock run of the test, variable `DRYRUN` in `config.robot` has value set to `true`. This variable signals at the beginning of the test to signal RQA that it be a dry run test.

3.5 Test setup and teardown

Before each test, a suite setup runs. During this setup, environment variables are set and the device prepares for testing. After test finishes, the teardowns return system to

original state. Suite setup configures general variables linked to the device and initializes robot, camera, and remote services. During test setup, device and robot are preparing for testing, and variables for specific test suite are initialized, such as information about test and device, also temporary information for a specific test that like username and password. These variables are sent to the RQA application, which processes them and uses them while testing. After setup is complete, individual commands are received by RQA and appropriately handled. RQA can communicate with the robot and its peripherals, camera, image processing and respond to tester's device.

Since test setup communicates with tested device and robot, it would not be possible to pass this setup without a device or robot connection. dry run mode needs to have unique setup and teardown that omit this part of the setup and uses custom keyword to signal Robotic Quality Assurance that test runs in dry run mode.

3.6 Test structure

Tests use arguments for more flexibility in their use-cases. Their values are set differently in each test. They are set from test variables or test configuration before each test setup runs. After the setups, the first step is usually user authentication on the device. Tests can omit authentication, but it limits services provided by the device. There are three forms of authentication card, username/password, and PIN. Authentication information and type are set during test setup unless directly stated in-suite variables otherwise.

Test creation consists of writing test keywords in the test file with appropriate test setup and teardown. Image processing needs to be trained for detecting test properties such as screen images. After the image processing has successfully trained, the tester can run the test from his device. Tester's device sends commands in the form of HTTP queries, and RQA main app receives them. RQA handles commands and monitors the actual state of the device and test.

```
*** Settings ***
Suite Setup       Suite Setup - Regression tests
Suite Teardown    Suite Teardown - Regression tests
Test Setup        Test Setup - Print Copy Scan with robot
Test Teardown     Test Teardown - Print Copy Scan with robot

*** Test Cases ***
Print all with accounting
    Given User exists in SafeQ
    And User has jobs in secure queue
    And Paper sensor data are cleared
    When Robot authenticates with print all enabled
    And Robot logs out
    Then Papers are printed
```

Listing 1: Example of a test

The test in Listing 1 creates a user in the device environment, authenticates with the user credentials while printing jobs in his account. The test has configured setups and teardowns in the settings section.

3.7 System environment representation

The main elements used during testing are screens, buttons, regions, and flows. Screens control the actual state of the device. A screen is a picture of the device screen made by the same camera that is used for testing. Image processing is trained to detect and differentiate screens. A screen has an assigned template and terminal type. Templates are used for better consistency through tests and generally name what is on the screen. The terminal type is a property containing a version number of the software on the tested device. The device can have tests on different software since the software reinstalls before each test. Each screen contains a description of how the RQA handles it in the test. There are four handling types Continue, Wait, Throw, and Escape. Continue continues with test after screen appears. The wait handling is used on loading screens and tells RQA to wait for a fixed amount of time. Throw represents error screens, and after it appears, the test usually fails. Escape screens are most commonly used for popups and confirmations, RQA finds a button that leads away from this screen and presses it.

Figure 3.3 shows an example of a screen in the RMS web application. The screen has multiple buttons and active regions. The button is represented as a red rectangle.

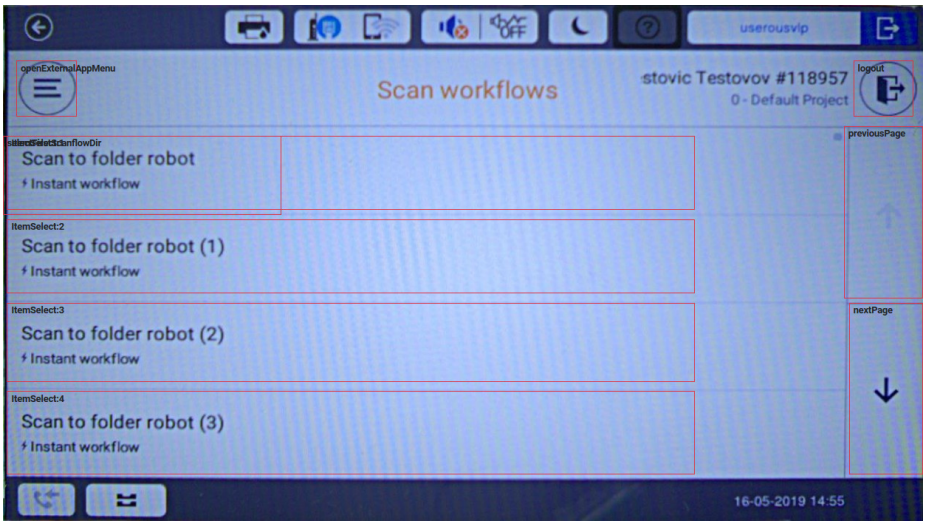

Figure 3.3: Screen in RMS web app

A button represents the physical or software button. A button stores all possible screens that it leads to, its name and screen that it is on. Buttons are defined by their abstractions. Button abstraction contains ID, Function, and optionally a comment. Abstractions are used as button references during testing. Each button stores its ID, its abstraction, device model, terminal type, and text. Text is the text that is on the button on the screen. It can be used for searching for the button with image processing. Button behavior is defined as a transition between screens. A button can have multiple behaviors. Button stores its three-dimensional coordinates with the origin being in the bottom right point of the screen. Button also stores its size. The difference between hardware and software buttons apart from the method of creation is just that hardware buttons do not contain terminal type nor text properties.

There are two types of regions. Feature region is a feature selected to help image processing to better differentiate between two screens. An active region is a place on the

screen that can be selected and used. Selecting an active region does not have to change the whole screen, but it changes a smaller part of it. The purpose of active regions can be paging, folder select, item select, or button toggle. Active regions are bound to buttons but usually represent only a part of the button that changes appearance with an action. They contain active and inactive color for differentiating button states. After an action, most often click on the button, image processing detects a change in the active region and confirms state change.

Flow is a sequence of actions. Usually, keywords represent flows. Each action is executed after the previous one successfully finished. Flows have a property marking, if they authenticate and what type of authentication they use. A flow that does not have an authentication set and crosses from an unauthenticated screen to authenticated fails. A fictive step is a step that represents predefined flow that can be used by other flows.

The flow consists of a sequence of buttons or actions. They do not have to be directly behind each other. For example, the first element is login, and next can be print even though the print button is not after logging in RQA can find the path and execute it. An essential part of the flow is flow type. Flow type represents whether flow authenticates and the method of authentication. Each flow step stores source and destination screens and value, signaling whether destinations screen needs a verification after the step. When verifying destination screen is enabled after the step, camera takes a picture and image processing compares the screen to the destination screen in the flow step. When the screens do not match flow and test stop with failure.

GoToScreen is a unique flow that generates dynamically. It calculates the path between screens and executes the transition. GoToScreen can calculate multiple layers in depth. Same as with flows, if any part of this transition fails, the whole GoToScreen fails. GoToScreen is not explicitly called in tests or flows by a keyword. It automatically runs when the next step starts on a different screen.

Figure A.1 shows flow representation in RMS web app. The flow starts on screen Copy, but when this flow is used, the current screen does not have to be copy, there needs to be an available path between the current screen and screen Copy.

Chapter 4

Design of dry run mode for RQA

This chapter describes the test development process for the RQA testing system and the design of test development using an extension of the system. Further, this chapter describes the design of the extension module and its testing.

4.1 Current state of test development

In the current state test development requires all hardware elements fully working, and the tester needs to run each test physically on hardware to check software errors. Test development is a process done in multiple steps. Starting with system setup and ending with executing successful test. Steps needed in test development are following:

1. The test development process begins with the hardware setup. The device needs to be connected and running. Robot and camera need to be positioned directly in front of the display on the device and calibrated. All peripherals need to be appropriately placed. The robot, camera, and peripherals need to be connected and functional. The computer that has those devices connected needs to run the RQA main app. There needs to be image processing running on a reachable device, and the database needs to be up and running. The tester needs to set up the environment. Test configuration needs to contain information about devices and tests.

2. The test development process begins with the creation of an abstract test. Usually just set of actions and expected results. After that, manual execution on the device checks its viability for RQA. While manually executing the test, the tester needs to use a camera and capture screen on each step of the test and save it to the database.

3. The tester needs to set all properties of screens based on the test scenario. Screens need to have set appropriate abstraction defining their functionality and their handling. Screens have property „authentication required“, that represents if authentication is needed to access the screen. The login screen does not need authentication, but menu or print needs.

   After the tester configurations all the screens, the tester adds buttons. Hardware button position is the direct measurement from the bottom left corner of the device display and added as coordinates. The tester adds software buttons by selecting a space on a screen in the RMS web application. With this, a button is created with a coordinated set automatically. Buttons need to have at least one destination screen.

Buttons' behavior can be overridden to lead to multiple screens that are used in flow creation.

Afterward, happens the flow creation. Flows are created in the flow editor in the RMS web application. Flows consist of one or more actions, usually button taps. The tester needs to set the software version and authentication method. Software versions on the same device often have different flows for various tasks.

Active regions allow the selection of multiple items or changing folders or pages when the screen does not significantly change. They symbolize regions that are changed after and action signaling change of state on a screen. Depending on the device and test active regions do not even have to be added.

4. After every screen is added, the tester runs image processing training. This may take some time depending on the similarity between the screens and the number of screens. After training, the tester can check the success rate, and if it is low, the tester can change lighting conditions or camera calibration.

5. The tester can finally run the test. If the test fails, from the test results, tester checks where the test failed and added missing information. If image processing training is needed, retrains, and reruns the test. If the test failed due to hardware, depending on the situation, the whole process would need to be repeated. When a test fails its execution stops, so even after fixing an error, it can still fail multiple times.

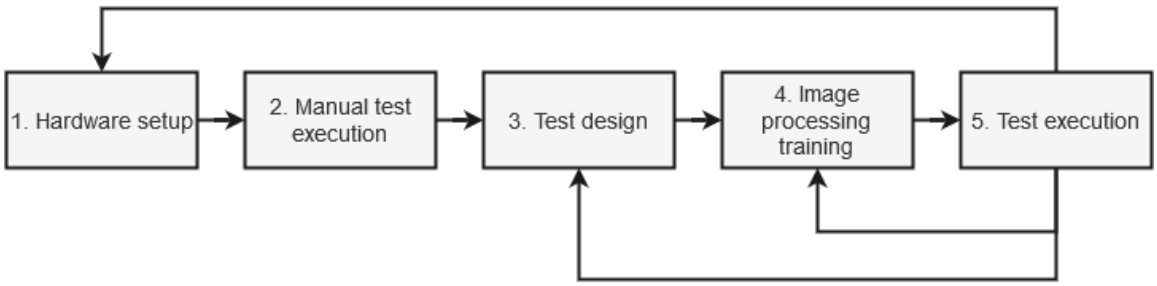

Figure 4.1: Test development

Figure 4.1 The current state of the test development process. Test development is a time demanding process, and with each error significantly lengthens. After each error, the tester has a lot of downtime, waiting for device or image processing to be ready for testing again.

4.2 Test development using dry run tests

The final program needs to make test creation quicker and a more user-friendly process. It is used when running tests physically on the device is either unavailable or too time demanding. The main improvement is disassociation from the hardware. Test creation is possible without actually running tests physically, and image processing training can be less frequent. Tests need to be run on hardware for final tests. However, their success rate is significantly improved. There is no difference in keywords in the two modes, RQA selects proper classes based on the test setup. Test development with dry run is also multiple step process.

Test development using dry run mode has following steps:

1. The process also starts with hardware preparation. However, only the camera and device suffice in minimal setup. The full hardware setup is needed when test creation is finished.
2. After creating an abstract test and manually testing the device, the tester adds screen pictures and screen information.
3. The tester proceeds with information input in the same way as in regular test creation. Adds screens, buttons, flow regions, and configures them.
4. After adding screens, buttons, flows, and active regions, tester runs the test in dry run mode. Dry run mode returns the XML sheet with a list of all actions executed during the test. Every action has its description and result. When a test fails, it stores reason for failure and if it is possible continues running test. The tester can correct multiple errors after a single test run and continue testing.
5. Tester trains the image processing.
6. Afterwards the tester runs the test in normal mode. Only possible errors left are with hardware or image processing.

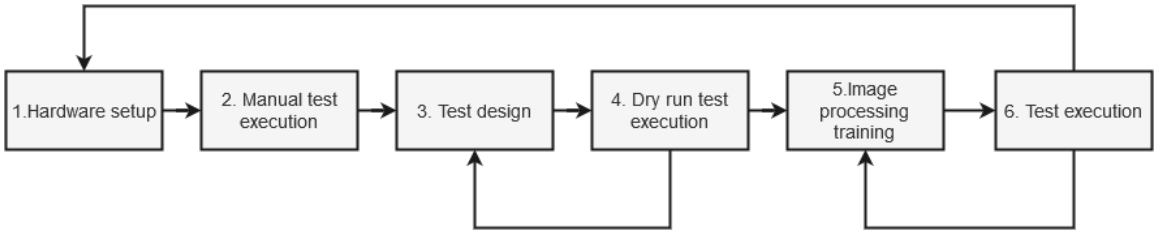

Figure 4.2: Test development with dry run

Figure 4.2 Test development process using the dry run of tests. This process is most useful in complex tests, where there are multiple errors during development, and when a larger amount of tests is developed. Only time dry run loses compared to regular run is when the tester finishes test on the first few tries, and only a small amount of regular test runs is needed. After passing the dry run, the tester also need to pass a regular test to confirm hardware functionality. After dry run tests passing, the tester needs to change setups and teardowns so that they do not ignore missing hardware devices. It is possible to use dry run with regular setup and teardown. However, dry run loses its advantage of not being bound to the device. The tester needs to be able to quickly and reliably change between both test modes.

4.3 Design of dry run extension

This extension adds the possibility of dry running tests, while not affecting the RQA during regular. The main feature of this extension is mocking hardware connections through dependency injection on some key services. Currently, the camera acts as feedback from the device. This is not possible during the dry run, so the dry run needs to collect the

current state from test keywords and test actions. The state is essential for determining the possibility of transitions from one state to another.

The dry run needs to be able to continue after encountering an error and needs to be faster than a test using hardware. Since all queries for hardware are immediately parsed and returned in the accessing layer, there is little time spent waiting.

4.4 Continuation after error during test

Dry run mode needs to be able to continue after it encounters an error. Continuation is dependant on a missing element. Dry run mode can create some elements like screens, but creating too many may lead to misleading results. In a simple scenario, where the robot presses a button, and the destination screen that it is supposed to lead to is missing, the logger saves information about the missing screen, and the access layer creates a new screen and sets it as the current screen. In more complex ones, when there is flow using dynamic path search, and multiple parts are missing, creating finishing state of the flow and reporting the missing step may lead to neglect of other steps in the flow. Failing a test in that scenario is better than continuing through those uncertainties.

The generation of new elements only happens when the next step has a distinct starting position, and RQA has enough information to create it. Since dry run mode image processing is not used, generating does not have to create fully-functional elements links to other steps allowing the test to continue.

When searching for the next element after an error, the test needs to take into consideration that the element does not have to be on the same screen. But searching all saved elements based on functionality does not work either, because when there are multiple elements with the same functionality test does not know which one to choose. To solve this test, chooses not only by functionality but also destination screen based on the next action.

4.5 Presentation of test errors

Every action is logged into the XML file. If there was an error or a warning, it is also printed to the command line. The XML file is converted to HTML for better readability. Every action is recorded, and in result, the tester can check if the test successfully executed given actions and if the test failed at which steps and the reason for the fail. For the creation of the XML file, I use XmlWriter, and translation to HTML was done with `XslCompiledTransform` using an XSL stylesheet.

Figure 4.3 shows the result of failed tests. Test failed to connect to the printing environment, failed to find robot calibration, and to find a button. The test continued after the failure of searching for a button.

4.6 Testing dry run

Testing dry run functionality was done on multiple different levels. Unit tests were created, which used an in-memory database with testing framework NUnit. Unit tests are mainly focused on mocked classes and their functionality. Unit tests utilize the in-memory database for creating an empty database with test-specific information in each test.

System tests are in from of simple test scenarios that are a shorter version of tests. Tests usually consisted of one or two simple actions to verify their functionality. For these

test Name	device Id	device Model Id	terminal Type	robot used
rnd2611.rnd.local	56	46	SQTASQ6T	False

action log	action context	aditional information	result
Searching for Devices with identifier 56			PASS
Searching for Devices with identifier 56			PASS
Searching for RobotPeripherals for device 56			FAIL
Searching for Devices with identifier 56			PASS
Searching for Screens with identifier 1			PASS
Searching for Flows with identifier AuthenticationCardOrUsernamePassword			PASS
Searching for Screens with identifier 1			PASS
Searching for Screens with identifier 394			PASS

Figure 4.3: Test result example

tests, I created a fictional device and robot which tests used as a reference to the database. I filled this device with screens, buttons, flows, and regions.

As acceptance tests, I used imitations of test creation scenarios. Testers were creating tests utilizing dry run mode and not utilizing it at all. Tests were done on real hardware with robots physically executing tasks. Test creation scenarios are mainly focused on the number of errors during test creation and were examining the time needed to execute each test run.

Chapter 5

Implementation of extended testing system

RQA application is written in C# language using ASP.NET Core framework 5.1. The application utilizes Castle Windsor container 5.2 for object management. Actions utilize services that are provided by the container. Communication with the testing environment is done with HTTP queries. After a query arrives, it calls an appropriate Controller, that controller starts an action and returns the action result to the testing environment with another HTTP query.

5.1 Filters and middleware ASP.NET Core

ASP.NET core [6] is an open-source web framework focused on the development and maintainability of web applications. It is a general purpose framework, on top of which we used ASP.NET Core MVC. ASP.NET Core MVC framework [4] provides a pattern-based approach to building applications. It utilizes the Model-View-Controller(MVC) design pattern [4]. MVC patters separates application into models, views and controllers. When using this pattern user action is handled by a controller. Controller then changes its with the action model and the model is displayed in view to the user. This exchange is in form of HTTP reqeuests and responses.

MVC middleware uses a pipeline for request handling. Middleware works on the same level as ASP.NET Core and all requests pass through it. Filters work on the MVC level and only some actions are filtered.

The testing system utilizes middleware for setting correct dependencies. Filters are used for responding to requests that do not need execution or can be skipped. Responses are either true or false.

Middleware in ASP.NET Core

Middleware are classes [6] handling HTTP requests and responses. There can be multiple layers of middleware in the application pipeline. Each layer can decide to run code before, after, or even if it wants to pass the request further in the pipeline.

Middleware is created in class `DryRunMiddleware`, it contains one method. The method `InvokeAsync` 5 is used for parsing incoming queries. During test setup, RQA app parses each query and searches for string `dryRun`. It allows to set appropriate variables to run the whole test in dry run mode. Test needs during test setup or suite setup send request

using the `Set Dry Run` keyword. After the initial test setup, it handles exceptions and is responsible for resetting and outputting test errors and logs. The difference between invoking dry run and regular is that during dry run an error does not end the test, for which method `InvokeAsync` is also responsible.

Filters in ASP.NET MVC

Filters [5] allow running code before, after, or during specified action. The main feature of filters is that they allow skipping the action by generating response and stopping execution. The filter is applied with attribute to action or controller.

Filters are used during processing incoming requests. Filters check attribute above action or controller and accordingly handles requests. Action filter processes action or controller if it has the `DryRunAttribute` attribute. Handling depends on the value of the attribute. In filtered actions, the output is either true or no output at all, and the action is skipped. Value true is returned in actions that always need to return true, such as status checking or connecting to devices. Filters are used only on basic actions that cannot run or are too slow.

The method in Listing 6 is called on every action in Controller, that has `[ServiceFilter(typeof(DryRunActionFilter))]` attribute. This method checks if the called method or its controller also have `DryRunAttribute`, if they contain the attribute, according to attribute value, the method is handled.

5.2 Dependency injection with Castle Windsor

Castle Windsor is an Inversion of Control container. Inversion of Control Container [12] is a container, that manages objects. It is aware of objects and their relationships and makes invocations on them. The container manages object creation, destruction, lifetime, configuration and dependencies.

Dependency injection is used for injecting mock classes. I made mock classes made for the lowest level parts of the system. RQA access layer, image processing, robot control were only services that were needed to be mocked. The dependant class has a constructor interface that is dependent on both mock and regular classes. When a test runs in a dry run mode injector chooses to resolve mock classes instead of normal ones.

```
Component
    .For<IRobotConnection>()
    .UsingFactoryMethod<IRobotConnection>((x) => {
        var prop = x.Resolve<TestProperties>();
        if (prop.GetDryRun()){
            return x.Resolve<RobotConnectionMock>();
        }
        return x.Resolve<RobotConnection>();
    }).LifestyleTransient()
```

Listing 2: Filter method

Listing 2 contains registration of a class. It decides which class should be injected into interface `IRobotConnection`. If `DryRun` property is set mock class will be injected.

5.3 Saving current state

The current screen is set at the start of the test to the default screen. After each access to the database searching for a button, the current screen is changed.

When trying to identify the active tab in normal mode, image processing is used. That is not possible without a camera and trained image processing. At the test beginning, the current tab and item are set to the first tab that is scanned by image processing. Since all tabs and items have the same functionality, it does not matter which tab is active. Afterward, each tab or item when detecting active tabs as compared to the first one and if it is the same image processing returns that it is active.

Interface `IDryRunError` is used for saving information about every action during the dry run. The interface has multiple implementations. Each of them is used for different types of errors. Each implementation stores a different amount of information but contains a method `Report()`, that condenses all this information to a single string that is saved in the final test result.

5.4 Database access layer

The original application was directly accessing the database. This however, prevents testing without database access and is harder to expand. I created an access layer to make the application more robust and flexible. In normal mode, the access layer usually directly accesses the database, but in dry run mode, it also collects and saves information about actions. It is the main source of test error information.

Both dry run and normal modes are represented within an interface `IRqaAccessLayer`. When constructing class depending on mode class `RqaAccessLayer` or class, `RqaAccessLayerMock` is injected.

```
Task<Screen> FindScreen(int screenId) ;
```

Listing 3: Method declaration in `IRqaAccessLayer` interface.

```
public async Task<Screen> FindScreen(int screenId) {
    var action = new MissingItemError(nameof(RqaContext.Screens),
                screenId.ToString());
    var screen = await _dbContext.Screens.
                SingleOrDefaultAsync( x => x.Id == screenId);
    if (screen == null)
        action.Result = false;
    _errorLogger.Add(action);
    return screen;
}
```

Listing 4: Method declaration in `IRqaAccessLayer` interface.

`RqaAccessLayerMock` has specific behavior in each method. Methods collect information, access database, return results, and sometimes create results to allow the test to continue running after an error. Information is collected by using class `MockErrorLogger`. The

method in listing 4 saves state in object `MissingItemError` that is passed into `errorLogger` with also result and additional information. Methods in regular classes usually directly accesses the database without any error or state collection.

5.5 Go to screen implementation

GoToScreen(GTS) is a unique method that dynamically searches for paths between screens. GTS relies on image processing to identify the current screen before it can search for a path. In dry run mode, GTS searches from the default screen of the device when the current screen is unknown. This is not always correct, but it allows the test to continue after an error. Both goto and the path are recorded in the log and visible to the tester. Goto initially does not allow searching from an unauthenticated screen to authenticated. However, it is necessary when creating a path in dry run mode.

During the execution of steps, each action changes state in `DryRunProperties`, which image processing mock class uses as a source of screen detection.

5.6 Image processing

Image processing is used for evaluating screens, and for dry run to work without screens. Image processing needs to be able to access test information. Class `ImageProc` is being avoided by injecting mock class `ImageProcMock`. This class returns active elements based on requests, the current screen only from dry run parameters. In most methods, not executing the task is sufficient, but in some image processing needs to verify that the item is saved in the database in those scenarios, class `MockErrorLogger` is used for log and error storage.

Chapter 6

Testing Robotic Quality Assurance in dry run mode

Testing with the robot was done with RQA running on Virtual machine with 8GB of Random-access memory and 2.1 GHz processor. Image processing was running on the same machine. The machine was in the same network as the tested device, its peripheries, and the RMS database. Testing dry run was done on a similarly equipped machine using a VPN connection to the network with the RMS database. Robots on all tested devices are the same, so are the cameras and the type of software installed. In the first test, tested devices are printers Epson WF-C20590, Konica Minolta C364, and HP E77650. I choose those devices to show the difference between manufacturers. For the second test, the tested device is Epson WF-C20590.

6.1 Testing dry run mode performance

	Dry run (mm:ss)	Epson (mm:ss)	Konica Minolta (mm:ss)	HP (mm:ss)
Authentication with correct Card	1:15	3:20	3:54	3:35
Secure print with accounting	2:50	3:25	3:20	5:43
Authenticate with username password	1:49	2:41	2:26	4:46

Table 6.1: Test run time

Table comparing test run time for different devices and dry run test. Dry run is the fastest in every test, however during test development after running test in dry run mode, test needs to be also run on hardware in regular mode. All values are averages of the last five runs, since the test run duration varies between runs, especially on hardware.

6.2 Testing development process with dry run mode

Table 6.2 compares to time spent developing the same test, depending on how many errors occurred during test development. Scenarios measure only the time needed to run the test and train image processing under the assumption that the time needed to input information, set up hardware, and setup software is the same.

	0 errors (mm:ss)	1 errors (mm:ss)	2 errors (mm:ss)	3 errors (mm:ss)	4 errors (mm:ss)
dry run	4:37	5:46	6:10	6:42	5:53
Regular no Image processing	3:23	7:50	11:30	15:31	18:12
Regular	3:23	9:31	15:55	22:17	27:32

Table 6.2: Test design time

During the dry run, development time amounts to the time needed for a dry run, the time needed for image processing, and the time needed for the regular run. Dry run returns all missing information from the test in a single run.

The regular test run has two test scenarios. In the scenario `Regular no Image processing`, image processing training is done only once, before the first test run, all errors are in test logic and element relationships. Test development time amounts to the time needed for repeated regular test runs, which replicates the test development process when the errors are in behaviors or properties. The number of test runs is one higher than the number of errors.

Scenario `Regular` has image processing training before the first run and after each error to emulate errors occurring during image capture or training errors. Test development time amounts to the time needed to run a regular, and after the test fails, train the image processing. The number of test runs is one higher than the number of errors.

6.3 Evaluation of test results

Testing has shown that the dry run test is faster than the regular test run. Dry run mainly gains time during test setup and image recognition. Dry run Does not wait for the device restart before nor after each test. Tests also shown that there is not significant difference between device manufacturers.

There are some scenarios where the regular run is more efficient than the dry run. When a test secedes on the first run of test development, the dry run is slower. Since the dry run does not fully cover all possible test errors, it needs after successfully finishing a regular run. With more complex tests, where test development errors are more common, the dry run is most useful for test developers.

Dry run efficiency heavily depends on how it is used and when. If the tester is creating a simple test that runs correctly after the first few iterations, most likely, the added need for changing test setup makes dry run less practical. But when creating a higher amount of tests in one session, the need to run tests on hardware and train image processing gradually favors using the dry run.

Chapter 7

Conclusion

The goal of this work was to improve the robotic testing system RQA, used by Y Soft. The created tool is a module of the RQA testing system that improves the test development process. The tool allows running the tests without using physical devices. During a test run, the test ignores hardware elements and can continue after encountering an error. During the implementation, dependency injection was used to inject mock classes, imitating communication with hardware devices. Action filters and an additional middleware layer were used for quicker responses to test queries.

The tool was tested by running the same tests with the dry run and without the dry run and comparing the time needed to finish the tests. Another testing scenario was testing the development process. The development process was evaluated by creating tests with a predefined number of test errors and comparing the time needed to fix those tests with the dry run and without the dry run.

The test results show that the dry run tests are faster and collect more information than regular run. However, the test speed is still too slow to be fully usable for test development. The need to run a regular test to confirm dry run test results coupled with a difference in test environment configuration during the dry run, adds more inconvenience to testers. The future development in the usability of RQA can be improvements in the speed of test execution and improving the testing environment for easier configuration changes. The next step is the inclusion of dry runs in the test development process at the company Y Soft and use it for designing tests.

Bibliography

[1] Bisht, S. *Robot Framework Test Automation*. 1st ed. Washington, DC, USA: Packt Publishing, July 2013. 30-34 p. ISBN 1783283033.

[2] Bourque, P., Fairley, R. E. and Society, I. C. *Guide to the Software Engineering Body of Knowledge*. 3rd ed. Washington, DC, USA: IEEE Computer Society Press, 2014. 1–12 p. ISBN 0769551661.

[3] Foundation, R. F. *ROBOT FRAMEWORK documentation* [online]. 2020 [cit. 2020-02-20]. Available at: https://robotframework.org/#documentation.

[4] Freeman, A. *Pro ASP.NET Core MVC*. 6th ed. Apress, 2016. 1–6 p. ISBN 9781484203972.

[5] Kirk Larkin, T. D. and Smith, S. *Filters in ASP.NET Core* [online]. April 2020 [cit. 2020-04-14]. Available at: https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/filters?view=aspnetcore-3.1.

[6] Lock, A. *ASP.NET Core in Action*. 1st ed. Manning Publications, july 2018. ISBN 1617294616.

[7] Myers, G. J. and Sandler, C. *The Art of Software Testing*. 3rd ed. Hoboken, NJ, USA: John Wiley & Sons, Inc., 2004. ISBN 0471469122.

[8] Osherove, R. *The Art of Unit Testing*. 2nd ed. Manning Publications, 2013. 1–20 p. ISBN 9781617290893.

[9] Patton, R. *Software Testing*. 2nd ed. USA: Sams Publishing, 2005. ISBN 0672327988.

[10] Pautasso, C. and Wilde, E. *Why is the web loosely coupled?: A multi-faceted metric for service design*. 1st ed. Madrid Spain: Association for Computing Machinery, April 2009, p. 911–920. WWW Conference, no. 8. ISBN 9781605584874.

[11] Pearl, S. *Object Lifetime in .NET Framework* [online]. 2015 [cit. 2020-01-07]. Available at: www.c-sharpcorner.com/UploadFile/b08196/object-lifetime-in-net-framework/.

[12] Rossi, J. *Castle Windsor Documentation* [online]. November 2019 [cit. 2020-03-02]. Available at: https://github.com/castleproject/Windsor/blob/master/docs/README.md.

[13] Seemann, M. *Dependency Injection in .NET*. 1st ed. Manning Publications, september 2011. 30–34 p. ISBN 9781935182504.

[14] Winteringham, M. What's the difference between a good test and a bad test? *Designing Tests* [online]. 2020 [cit. 2020-01-14]. Available at: www.ministryoftesting.com/dojo/lessons/designing-tests-what-s-the-difference-between-a-good-test-and-a-bad-test.

Appendix A

Appendices

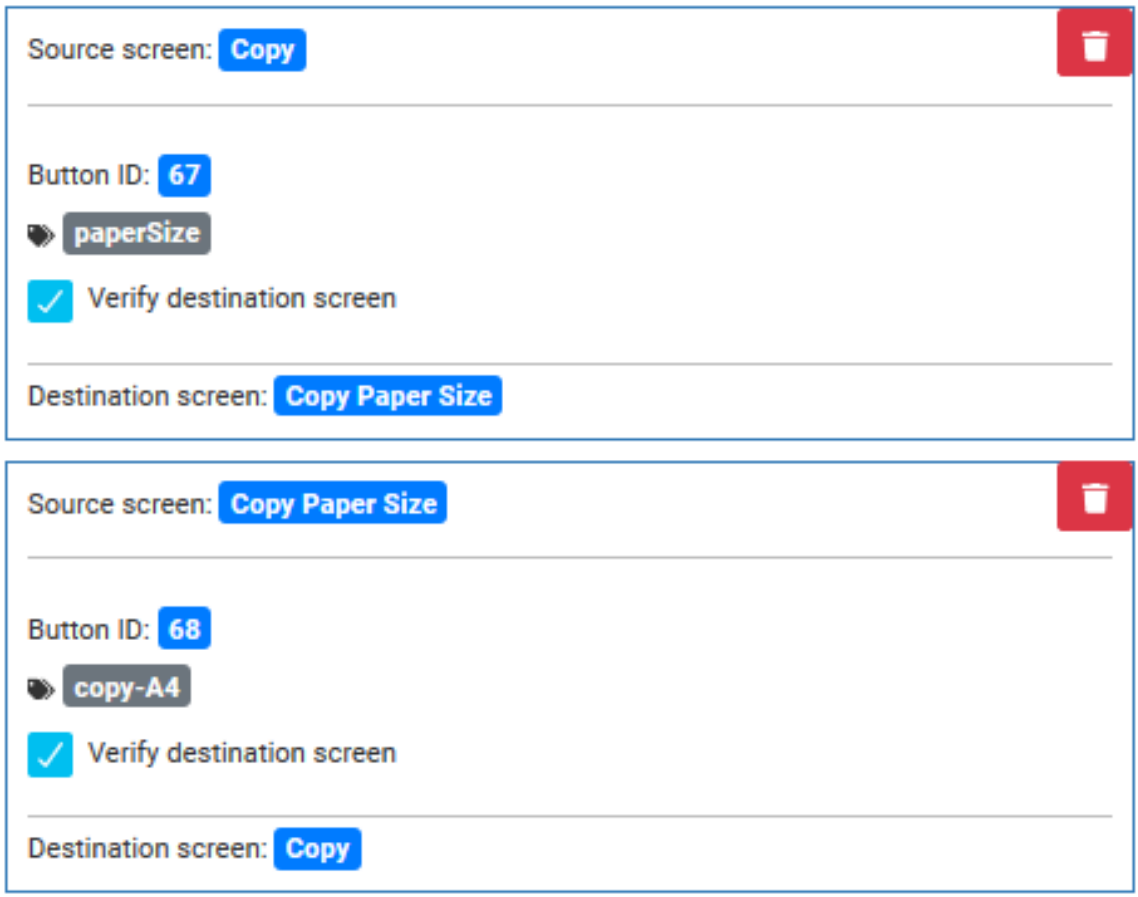

Figure A.1: Flow in RMS web app

```
public async Task InvokeAsync(HttpContext context, RequestDelegate next)
{
    if (context.Request.Query.TryGetValue("dryRun", out var value))
    {
    if (bool.TryParse(value, out var result))
        _testProperties.SetDryRun(result);
    }
    if (_testProperties.GetDryRun())
    {
        try
        {
            await next.Invoke(context);
        }
        catch (Exception e)
        {
            _testProperties.ClearTestErrors();
            throw (e);
        }
    }
    else
    {
        await next.Invoke(context);
    }
}
```

Listing 5: Middleware invocation method

```
public async Task OnActionExecutionAsync(
    Microsoft.AspNetCore.Mvc.Filters.ActionExecutingContext context,
    ActionExecutionDelegate next)
{
    string action = (string)context.RouteData.Values["action"];
    MethodInfo methodInfo = context.Controller.GetType().GetMethod(action);
    var attrs = methodInfo.GetCustomAttributes(typeof(DryRunAttribute), true);

    if (!string.IsNullOrEmpty(action) && attrs.Length > 0)
        CheckAttributes(attrs);
    else
        _action = FilterActions.Continue;

    switch (_action) {
        case FilterActions.ReturnTrue:
            context.Result = new AcceptedResult("", true);
        break;
        case FilterActions.Skip:
            context.Result = new AcceptedResult();
        break;
        default:
            await next();
        break;
    }
}
```

Listing 6: Action filter method