UNIVERSITY OF TURKU

Supporting web programming assignment assessment with test automation and RPA

Software Engineering

Master's Degree Programme in Information and Communication Technology

Department of Computing, Faculty of Technology

Master of Science in Technology Thesis

Author:

Tomi Salomaa

Supervisors:

MSc (Tech.) Sampsa Rauti (UTU)

MSc Jari-Matti Mäkelä (UTU)

October 2022

Master of Science in Technology Thesis
Department of Computing, Faculty of Technology
University of Turku

Subject: Software Engineering
Programme: Master's Degree Programme in Information and Communication Technology
Author: Tomi Salomaa
Title: Supporting web programming assignment assessment with test automation and RPA
Number of pages: 85 pages, 15 appendix pages
Date: October 2022

Automated software solutions to support and assist in assessment of student implemented applications are not a rarity, but often need to be custom engineered to fit a specific learning environment or a course. When such a system can be fielded in use properly, it has a tremendous potential to lighten the workload of course personnel by automating the repetitive manual tasks and testing student submissions against assignment requirements. Additionally, these support systems are often able to shorten the feedback loop which is seen to have a direct impact on student learning.

In this thesis test automation and robotic process automation are researched to discover how they can be used to support web programming assignment assessment. The background on software testing, automation and feedback related pedagogy are researched mainly by the methods of literature review and expert interview. A third methodology – design science – is then applied for the purpose of verifying and extending the learnt theory in an empirical manner. A research artifact is created in the form of a prototype capable of supporting in assessment tasks. Performance of the prototype is measured by recording set execution metrics while assessing anonymized case study student submissions from a web development course arranged by University of Turku: DTEK2040 Web and Mobile Programming.

Thesis concludes that to support assessment through test automation is to focus on unit and system level testing of functionalities while assuming the exact implementation at code level cannot be fully known. Suggestion is made that relying on assignment descriptions as basis for test design is not enough, but rather requirements engineering should be done together with course personnel to take advantage of their experience in what sort of errors are to be tolerated in student submissions. Thesis also concludes that automation can perform interaction with student submissions, file manipulation, record keeping and tracking tasks at a satisfactory level. The potential to shorten the feedback loop and summarizing quantitative feedback for the student is recognized, however, to build an automated system to identify, gather and summarize formative, pedagogically more valuable feedback was noted to be out of scope for this thesis and suggested as future work to possibly extend the prototype with.

Keywords: automation, testing, RPA, robot framework, web application, assessment

Diplomityö
Tietotekniikan laitos, Teknillinen tiedekunta
Turun yliopisto

Oppiaine: Ohjelmistotekniikka
Tutkinto-ohjelma: Tieto- ja viestintätekniikan tutkinto-ohjelma (DI)
Tekijä: Tomi Salomaa
Otsikko: Supporting web programming assignment assessment with test automation and RPA
Sivumäärä: 85 sivua, 15 liitesivua
Päivämäärä: Lokakuu 2022

Automatisoidut ohjelmistoratkaisut, jotka tukevat ja avustavat opiskelijoiden toteuttamien sovellusten arvioinnissa, eivät ole harvinaisia, mutta ne useimmiten joudutaan rakentamaan tiettyyn oppimisympäristöön tai opintosisältöön sopiviksi. Tällaiset järjestelmät omaavat kuitenkin valtavan potentiaalin keventää kurssihenkilöstön työtaakkaa automatisoimalla toistuvia manuaalisia työtehtäviä ja automaatiotestaamalla opiskelijoiden palauttamia tuotoksia asetettuja tehtävävaatimuksia vastaan. Järjestelmät johtavat varsin usein myös opiskelijan näkökulmasta nopeampaan palautesykliin, jolla kyetään todeta olevan suora vaikutus oppimiseen.

Tässä opinnäytetyössä tutkitaan testiautomaatiota sekä robottiprosessiautomaatiota pyrkimyksenä selvittää kuinka näitä teknologioita voitaisiin hyödyntää tukemaan web-ohjelmointitehtävien arviointia. Ohjelmistotestauksen, automaation ja palautteen pedagogiikan taustoja tutkitaan pääasiassa kirjallisuuskatsauksen ja asiantuntijahaastattelun menetelmin. Lisäksi sovelletaan kolmatta metodologiaa, suunnittelutiedettä, jonka tavoitteena on vahvistaa teoriaosuuden havaintoja sekä pyrkiä empiirisesti laajentamaan niitä. Suunnittelutieteen kautta tutkimusartifaktina syntyy prototyyppi, jonka suorituskykyä ja hyötyjä mitataan keräämällä dataa hyödyntäen aitoja, anonymisoituja opiskelijapalautuksia Turun yliopiston järjestämän DTEK2040: Web and Mobile Programming -kurssin tiimoilta.

Opinnäytetyön johtopäätöksenä on, että arvioinnin tukeminen testiautomaation avulla on keskittymistä yksikkö- ja järjestelmätason toiminnallisuuksien testaukseen. Testaukseen on liitettävä myös oletus, että arvioitavan kohteen tarkkaa toteutusta kooditasolla ei voida täysin tuntea. Tehtäväkuvausten käyttö testitapausten suunnittelun perustana todetaan riittämättömäksi, ja vaatimussuunnittelu ehdotetaan tehtävän yhdessä kurssin henkilökunnan kanssa, jotta heidän kokemuksiaan voidaan hyödyntää yleisimpien opiskelijapalutuksissa ilmenevien virhetapausten kartoittamiseksi sekä testitapausten tarkkuuden ja arvioinnin jyrkkyyden säätämiseksi. Prosessiautomaation osalta todetaan, että automaatio kykenee suorittamaan vuorovaikutusta opiskelijoiden palautusten, tiedostojen käsittelyä, kirjanpito- ja seurantatehtäviä tyydyttävällä tasolla. Mahdollisuus palautesilmukan lyhentämiseen ja summaavan palautteen yhteenvetoon opiskelijalle tunnustetaan myös empiirisesti. Laadullisen, pedagogisesti arvokkaamman palautteen kokoaminen ja jalostaminen todettiin tämän opinnäytetyön mittakaavassa liian suureksi projektiksi ja sen empiiristä toteutusta ehdotettiin yhtenä mahdollisena jatkotutkimusaiheena.

Asiasanat: automaatio, ohjelmistotestaus, automaatiotestaus, RPA, robot framework, verkkosovellus, arviointi

Table of contents

Appendix E: Example summary template

Appendix F: Common keywords

Appendix G: Custom library

List of figures

List of tables

List of codes

Abbreviations

API	Application programming interface
GUI	Graphical user interface
ISTQB	International Software Testing Qualifications Board
JSON	JavaScript Object Notation
RF	Robot Framework
RPA	Robotic process automation
SPA	Single page application
UI	User interface

1 Introduction

1.1 Background

It is perhaps fair to say that learning to program is challenging. While learning the related theory is one part of it, one could argue that hands-on practice by coding solutions to different kinds of problems is extremely important. Due to practice being invaluable for learning programming, programming courses are often designed to be very practical, containing multiple programming assignments to allow students to learn through repetition of the principles or concepts being taught.

The hands-on, learning-by-doing approach has been present in teaching of programming from the very beginning. From the very early on this task-and-assignment-based way of teaching programming has also sparked interest in being able to at least partly automate the work related to assessing student solutions. Descriptions of using automation to support grading of programming submissions are available from as early on as the year 1960 when Hollingsworth presented the automated grading system [17]. Hollingsworth used the automated grader during programming courses held at the Rensselaer polytechnic institute, a private research university in USA.

The motives for using automation were very similar to what can be found mentioned in reviews and surveys written on the subject matter today [1; 10; 21; 29; 33]: increasing class sizes, extensive workload related to assessing and time required to manually perform the assessment process. Additionally, the research results often list many more reasons to use automation over purely manual approaches such as the consistency and accuracy of assessment as well as removal of unintended biases, as noted by Romli & al. [33, pp. 1186], for example.

The research and overall interest for automated solutions in this field has been notable and as such a lot of advancements have also been made since the Hollingsworth's automated assignment grader. For example, in the year 2018 Keuning et al. were able to identify over a hundred [21, pp. 11-12] tools that can automatically assess and generate feedback for programming exercises. The same review also indicated that many of these tools are often custom built for a purpose, such as a specific programming course or a thesis, or aim to support the teaching and assessment of very basic principles of programming. From this perspective this thesis does not seem to be unique, however, the need for pursuing a customized solution seems reasonable and is hard to avoid as long as programming courses offer unique content that needs to be taken into account.

1.2 Problem statement and research questions

Automation is a possible avenue of approach to support an instructor in the assessment of student assignments. Implementing test automation and robotic process automation (RPA) together may be able to check for the basic functionalities of a student's proposed solution to an assignment, but to also collect and summarize individual and solution specific feedback. Aside from the hands-on assessment work, automation has the potential to help save workhours spent in the so-called business processes of the overall process of grading students on any course.

This thesis aims to develop a prototype system, while relying on open-source and free-to-use tools, that combines test automation and RPA for the purposes of supporting programming assignment assessment work. The concept seeks to integrate with already existing manual workflows and platforms for *DTEK2040 Web and Mobile Programming*, which is described as intermediate level studies and arranged by the university of Turku as part of the Bachelor of Science (technology) studies in information and communication technology.

The resulting prototype serves to present potential viability of automation in the areas of automation testing and RPA for programming courses while being specifically developed with DTEK2040 web application related exercises in mind. This study also researches aspects related to programming assignment feedback and as a part of the results presents ideas how these aspects could be later incorporated into an automated solution to possibly enhance student learning.

Thesis presents, and through the course of this study answers, the following research questions:

Q1: How to support the assessment of web application programming assignments with test automation?
Q1.1: Which testing levels should be focused on?
Q1.2: Which testing techniques are applicable for testing student solutions to assignments?
Q1.3: How to turn an assignment briefing into test cases?

Q2: How to support the assessment and feedback process of assignment assessing with RPA?
Q2.1: What manual work related to assessing and feedback is there to automate?
Q2.2: What kind of feedback should be gathered from the student solutions to assignments?

1.3 Scope and delimitations

Thesis studies software testing, test automation and RPA, which are observed within the context of applying the methods and technologies to support programming assignment assessment. To further

narrow the scope, programming assignments have been limited to web application programming assignments as described in the course contents of DTEK2040.

As this study proposes to construct a prototype to empirically verify the theoretical research efforts and their viability, but to also provide limited automation functionality for student submission assessment and scoring, the use of DTEK2040 as a case study target applies its own limitations to the scope for developing the prototype. The technological choices and assignment requirements that are in use within DTEK2040 must also be considered during development. While these hard delimitations most certainly guide the research, consideration of the generalizability of the final solution and results of this research are carried along throughout the study.

The prototype will use a tool called Robot Framework (RF). RF is a “generic open-source automation framework” [32] that originated at Nokia but has since become a framework maintained by a registered association called Robot Framework. Today the framework is widely used for testing and robotic automation in the software industry and beyond. [31] The choice of this framework has been affected by personal familiarity with the tool but also the extensive range of available libraries and programming language support of the framework which can be seen to help generalize the achieved solution in the future.

1.4 Research methods and sources

Three main research methods are used in conducting this study: literature review, expert interview and design science. The primary platforms used for searching academic articles were ACM Digital Library, Google Scholar and IEEE Xplore. Identical search queries were used in each one. The queries were constructed from a base query that was supplemented with a correlating topic specific support-query to find results. E.g., to find web application testing articles the base query would be extended with the “Testing” support query. The queries are presented in Table 1.

Table 1: Queries used for searching scientific reference material for the thesis.

General topic	Query string	Query type
Web application	(web OR website OR “web application” OR React OR Angular OR Vue OR JavaScript OR HTML OR DOM OR Python OR Java)	Base
Testing	AND (test OR testing OR "dynamic analysis" OR "static analysis")	Support
Automation	AND ((automation AND (test OR testing)) OR "robotic process automation" OR RPA)	Support
Assessment, feedback	AND (pedagogy OR learning OR course OR (programming AND assignment) OR assessing OR grading AND feedback)	Support

Additional inclusion criteria for academic sources were that the article should be peer-reviewed and published in the year 2005 or later. Other relevant book sources are extracted from the references found from academic articles. Aside from articles and books, web-based sources from well-known and respected authorities such as the ISTQB are included as their material is often referenced in the industry. Expert interviewing is used to supplement the findings from literature review, especially to gain the perspective relevant to the focus determined by DTEK2040.

The used methods of literature review and expert interview are also tied with design science. Design science is applied in the empirical part of this study. There the theory gathered prior is used to develop the design science artifact, a functional prototype, by following engineering design process. The process that is being followed is described by Eggert to contain four stages [11, pp. 6-8] as represented in Table 2.

Table 2: Eggert's four stages of engineering design process.

Stage	Activities	Goal
Formulation	Gather information such as requirements, performance targets, constraints and considerations.	To understand the problem and to start preparing a plan for its solution.
Generating	Synthesize or generate alternative designs to satisfy the gathered expectations.	To produce alternative design candidates for later analysis and evaluation.
Analysis	Predict the performance and/or behaviour of the design candidates.	To deduce whether a candidate design satisfies previously set constraints.
Evaluation	Compare design candidates based on their predicted performance by using criteria gathered in the formulation stage.	To decide the best design alternative to be implemented in practice.

While the described process is linear in the sense that it should be followed from formulation to evaluation in order, the generating and analysis stages together form a possible redesign iteration within the process. If a design candidate does not satisfy the constraints set during the formulation stage, it may be taken back into the generating stage for alterations and then reanalysed.

In this thesis the formulation stage is covered with Sections 2 and 3 which provide the necessary background information to understand the problem. These sections also partly give basis to design specifications that are based on theory gathered from literature and expert interview. The rest of the stages will be covered within the contents of Section 4 before finally presenting the results achieved with the decidedly best design implementation in Section 5.

1.5 Structure of the thesis

This thesis approaches developing the prototype to answer the research problem through four sections covering relevant theory and implementation. The first two sections following this introduction lay the foundation for testing, test automation and RPA. For these sections the most relevant sources come from literature and past research conducted on the subject. The latter two sections aim to build upon this theory and implement a combined solution as well as present the detailed results. For these sections literature and documentation sources are enhanced with the results of conducted expert interview.

The theory starts with Section 2. Here the required background information on testing of web applications will be provided. The aim is to cover the basis of and seek answers to sub-questions of Q1 through the following sub-sections:

2.1. describes the traditional objectives set for testing and how these could be considered when assessing solutions for programming exercises.

2.2. provides an answer for Q1.1. by considering the most usual way of dividing testing efforts based on their focus and highlighting the most meaningful levels of testing in the problem context.

2.3. provides an answer to Q1.2. by considering and comparing testing techniques, considering the special aspects of course exercises such as their potentially transforming nature.

2.4. provides an answer to Q1.3. by listing what should be considered when building a test case and how these notes should be incorporated with exercise assignments.

2.5. describes the most common challenges related to web application testing and how these should be considered when building the test cases or even the assignments.

The theory continues in Section 3. In this section the goal is to research and further define the differences between test automation and RPA. This section also researches the importance, nature of relevant feedback and the related principals to be considered when extracting feedback topics from student solutions to exercises. Section 3 seeks to answer the sub-questions of Q2:

3.1. describes the differences between test automation and RPA. Justifies their use for specific tasks and purposes in the assessment process and provides principles for implementing

automation. Partly answers Q2.1. by exploring what to consider when implementing RPA and what could be the potential targets in DTEK2040 assessment process.

3.2. describes the motives and requirements for using automation in web application testing. Completes the answer to Q2.1. by gathering principal ideas and focus areas to consider in terms of test automation and assessment for automation testing.

3.3. researches and describes the feedback that can be seen useful to advance the learning of programming. Considers what should and what should not be automated as well as answers Q2.2. by proposing a list of observations to be gathered and formulated into feedback from student solutions to exercises.

After these theory sections a model solution will be proposed. Section 4 describes this model with its open-source tools and libraries used as well as the overall functionality. The detailed implementation is then presented in Section 5 along with the results. Results are gathered from testing the thesis artifact with anonymized student solutions received as case study material from past instances of DTEK2040, and once analysed will be used to present the collected answers to research questions Q1 and Q2.

2 Testing web applications

2.1 Objectives of software testing

In the software industry, software testing is a quality assurance technique implemented throughout the software development lifecycle and applied widely to verify and validate different aspects of the end-product. This means that in the industry practices testing is often incorporated from the very early stages onwards to assure the quality by evaluating work-products such as requirements, specifications, program design as well as source code. [18; 23] As a process, testing can be said to contribute towards bettering reliability and the overall quality of the program under test by verifying and validating the various aspects of said product before it is brough available to end-users. According to sources such as Myers & al. the main goal of software testing as a process is to "find as many of the errors as possible" [23, pp. 6] to achieve the goal of increasing quality.

Apart from finding errors - and in this case finding errors from the perspective of assignment assessment taskers - testing is often credited with many other objectives as well. For example, ISTQB brings up aspects such as (a) building confidence to the quality of the product and the development work and (b) providing a safety net for the developers to do their work as being relevant goals for software testing. From assessment support perspective this could perhaps translate into boosting the student's morale by shortening the feedback loop through the help of automation testing. Ultimately the objectives set for software testing may vary depending on the overall context and details such as the test level that the testing efforts are being focused on. [19] This can also mean that by focusing to the pedagogical aspects extractable from testing results, software testing may very well lend itself to enhancing teaching efforts and student learning instead of only seeking errors in the system under test.

Though, it could be seen that quality as a central value remains even when applying software testing as a support for programming assignment assessment activities. The motivation behind testing for product quality can be seen to differ from an industrial context: testing activities are not there to necessarily assure an instructor or lecturer of the student's work quality but rather to support the pedagogical process. Thus, testing and the testing objectives need to be justified through pedagogical standpoints [18] and be able address the need for feedback of the learner who is also the developer of the work product under test.

Feedback itself can be formed through formative and summative assessment of assignment. From the formative point-of-view, quality as an objective means focusing on aspects that provide the learner opportunities to improve one's knowledge and skill that are relevant and within the scope of the studies the programming assignment relates to. On the other hand, testing objectives derived from the summative assessment point-of-view should provide the basis for making judgements about student achievements and progress, i.e. grading the student work product. [7]

Many studies propose and support the view that assessing for functionality is the most common approach when assessing programming assignments [1; 18; 29; 33]. Typically testing objectives for functional testing are gathered from basis consisting of given requirements and specifications such as business requirements, user stories, use cases or specific functional requirements [19]. The end-goal is to verify and validate that the system does what it is supposed to do. In the cases of typical programming assignments, functional requirements often boil down to the given assignment description and sub-task descriptions.

Software testing is understandably an integral part of web application development. Same reasons of bettering the overall quality of a software product apply to web applications as to any other software product. From the testing perspective web applications have a lot of common ground with traditional desktop applications when it comes to testing functionality, configuration and compatibility aspects of the product. They also present some unique issues that need to be taken into consideration. Some of the issues and considerations to note are underlined by Arom & Sinha in their review on techniques, tools and state of the art of web application testing. The aspects they raise in their research are: (1) performance requirements deriving from large user population, (2) state change related faults, (3) web browser related compatibility issues on top of operating system related compatibility ones, (4) multiple potential error occurrence points within a typical multi-tiered web application architecture and (5) the dynamic nature of software components being rendered at runtime based on user input as well as server response. [3]

Most of the theory and observations presented above about the objectives of software testing are also further supported by the results of the expert interview conducted with the personnel teaching and assessing student submissions for DTEK2040 [22]. From the results of this interview, one could analyse that at least the following notes are related to testing objectives:

- Finding errors from submissions is a core task; gained results are a basis for scoring and forming feedback.

- Assessing for functionality is the most common approach as the assignments often are very clearly formed to contain a set of specific functionality requirements; correct implementation reflects the student's understanding of the corresponding subject.
- Regarding course assignments dealing with React applications, state change related faults are among the somewhat recurring types of faults between different course iterations that should be looked for.

Regarding the assessment of non-functionalities, it was mentioned that if such aspects are considered then they should be derivable from the task assignment as clear requirements. Assessing the visual quality or structure of the code was mentioned to be often too difficult due to the subjective nature of such assessment. Answers provided also gave basis for understanding that many non-functional aspects were not necessary to be assessed given the scope and focus of DTEK2040. The reasoning behind very scoped assessing of functionalities was that the functionalities present in any given student submission is affected heavily whether an aspect was considered in the course material and the student should be able to present understanding of it. Nevertheless, the theory and observations backed up by the interviewees are interesting and should be considered when also thinking about which testing levels are relevant and should be focused on.

2.2 Testing levels

Software testing can be scoped to target certain abstractions levels. A common way of representing the testing levels and tying them to software development specifications is to present both in a so-called V-model where unit testing tests for unit specifications, integration testing tests for subsystem design, system testing tests for system specifications and acceptance testing tests for business needs and constraints. Names given to these testing levels may vary, with the most commonly variation being at the unit level. Sometimes this most atomic level of testing is referred to as module [4, pp. 104; 23, pp. 85] or component [21] level as well, but they all essentially have the same meaning.

2.2.1 Unit and integration testing

Unit testing focuses on software components that can be tested and verified separately in isolation from the rest of the software. Units can be thought to form the backbone of all functionalities of a software product and testing at unit level is considered important because defects at this level may be difficult to identify later when the whole software system is being considered. [4] Testing at this

level is a widely accepted practice in the industry and frameworks for testing are readily available almost regardless of the programming language or technology of choice [7].

In the most practical sense, unit testing is the process of testing entities of software such as functions. Main purpose is to catch local defects in that entity at the algorithm level. This also means testing activities themselves without exception need to rely on accessing the individual unit at a code level to be able to perform any sort of meaningful validation or verification. It is also why unit testing is often preferred to be performed by the developers themselves as they develop the units [7; 19] instead of an outsider tester.

Most typical defects detected at unit level are incorrectly implemented functionalities due to incorrectly coded logic or incorrect data flows. While the testing objectives for this level are almost exclusively related to functional testing, the main challenge at unit testing level lies with the test cases. Depending on the test coverage type and exhaustiveness, a single unit may require a plethora of written test cases since defects may need to be considered from the perspectives of execution paths that would not necessarily be obvious when considered from a business logic perspective of the system. [4; 23, pp. 85-111]

Unit level also poses certain challenges when observed within the context of this study: unit content variance. Basic programming assignments such as the ones requiring the student to write a method that takes certain parameters as input and then produces a required output are straightforward to test. Verifying that the method has been written as instructed should be easy in such cases but for more intermediate assignments the approach may not be as strictly set. As an extension, this also means that writing unit tests beforehand to use for supporting assessment activities is challenging because the student may quite freely approach the assignment when it comes to creating units. As it stands with software product development, the already mentioned habit of developers writing their unit tests perhaps springs from this. Often the one writing the unit logic is also the best person to match the unit test to that logic; writing these tests without knowing the exact behaviour on structure of the unit is quite a challenge.

In some programming courses relying on testing as a supportive element to assignment assessment the challenges of unit testing are partly taken into consideration by injecting hard requirements into the assignments. These hard requirements may include for example forcing the student to include a method named certain way into the program or, in a more web related context, giving forms and elements within the form certain identifiers by defining *id* attributes to be used. These requirements make the student implementation more testable as certain attribute values can be expected and used

with pre-created unit tests that are run against the submission.[1] The challenges related to testing individual algorithms at a code level are not entirely confined to unit level, but they do lessen as we proceed to higher-level testing.

From unit testing level the next step upwards is integration testing. Modern web applications are complex and, especially with service-oriented architecture solutions for example, integrate many components and data sources. This is also why integration testing is often another key testing level when web applications are being developed. The testing activities performed at this level can be done to verify that the interaction points of the web application work as intended and that the data flow between individual units and even from database interfaces is in a valid and required form. [34] Because the focus at this level is on the interactions and interfaces between separate units and as such it also often presents additional requirements for the test environment in the form of stubs and drivers.

While integration testing is a step up from the unit level, it still requires knowledge about the structure of the program and the interacting units to be carried out thoroughly. This is also why the usability of integration testing as a supportive test level for assessing support can be seen to suffer from many of the challenges that are also present at the unit testing level. However, integration testing does include certain traits that make it a little less inclined to requiring exact knowledge of separate units in terms of testing goals. For example, we may still write a test for an interface if we know two units should communicate by transferring data in JSON-format to test that format is being complied to, even though we would not know beforehand how the units themselves are going to behave internally. Though, even if integration testing may focus largely on testing interfaces, given the same approach of using hard requirements in assignments integration testing could be seen as even more of a valid focus when assessing certain web application programming assignments and parts of larger tasks, such as those concerning and dealing with APIs and database queries in general.

With DTEK2040 it is mentioned that the challenges brought up here regarding the use of unit and integration testing certainly exist and would most likely prove a challenge to automated testing in the current state of the course assignments. However, an additional note was also made that of course it is possible to modify the currently existing assignments and the tasks that are given so to

[1] Such techniques are assumable observable with *Web Software Development* course, for example. The course in question has at times very strict asserts in its automation tests but on the other hand allows for automating majority of the assessment work on part of web programming tasks. The course (https://wsd.cs.aalto.fi/) is arranged by Aalto university.

better be able to support the testing and automated assessment at these levels of testing. [22] This is an important note in the sense that testing at these levels does not necessarily need to be a one-way street. While the test object has the functional requirements that need to be fulfilled, it can also be built in such a way that it readily supports specific testing activities. An example of this with web applications would be the already mentioned requirement of using a certain id attribute with a specific element so that it is more predictably accessible and available for web automation test engines.

2.2.2 System and acceptance testing

It is worthwhile to note that system testing is a very large area of interest even from a study perspective [13] and it can be described or understood in many ways [23, pp. 119-131]. Though, a common notion is that system testing is the first level to consider also the non-functional requirements set for the system under test. Once the functionalities have been verified through unit and integration testing, system testing can be performed in the system environment with proper unit integrations and interactions. The focus is often on the end-to-end tasks and validation of business behaviours.

While test types are usually not too tightly boxed to certain levels of testing [19], system level can still be considered the level to perform testing activities related to stress testing, usability testing, security testing and configuration testing due to the nature of system testing [23]. Because system testing can also be mainly concerned with the non-functionalities of the program, the tests themselves also tend to follow a testing methodology that is concerned whether the observed outputs to specific inputs are equal to the expected outcome that is based on set requirements. This sort of approach makes many of the system tests less or entirely separated from knowledge of the inner workings of the code that will be executed when the actual test is performed.

Acceptance testing is a lot like system testing in this manner, however, the point-of-view for testing activities is traditionally from the end-user perspective and the goal to accept the system for production. In this sense the testing done at his level can be more confined compared to the system testing, focusing almost entirely on testing if the software meets the defined business requirements and workflows rather than individual functionalities or non-functional aspects of the system. Additionally to the business-focus the most notable difference to system testing is the absolute exclusion of the inner machinations of the program under test. [6]

From the perspective of this study, both the system test and acceptance test levels provide opportunity to be the focus levels for testing activities. Uncoupling test cases from the need to know specifically how the software under test has been built has potential to make the tests more reusable and does not unnecessarily limit the student from creating differing solutions that still meet the requirements from assignment point-of-view. System test and acceptance levels are also able to consider any non-functional requirements that might be part of a programming assignment, such as requirements for usability or security.

It is, however, important to note that for DTEK2040 assignments system testing is not considered to be able to fully cover the assessment and feedback requirements from pedagogical point-of-view [22]. It is mentioned that the coverage of testing should at least be extended to consider the course material thoroughly enough instead of just verifying functional outcomes. For example, the returned student assignment might seemingly be able to produce expected, correct behaviour when the system under test is system tested, this does not always mean that the implementation follows the provided study material. In such cases deeper scrutiny should be able to uncover that, for example, the application is creating unwanted side-effects or performing against the core principles of a SPA, which in turn could be considered as a defect from assessment perspective. The need for testing to cover the pedagogical aspects as well places interesting challenges in terms of deciding what kinds of testing methods and techniques to use to achieve meaningful support role for assessing student submissions.

2.3 Testing methods and techniques

2.3.1 Static testing

Software testing methods can be divided into two approaches: static and dynamic. Static testing is more generally also referred to as static analysis and it is traditionally performed manually by examining a work product. In quality engineering static testing allows for quality assurance to participate in the software development very early on as static analysis can be performed ideally as early as the first requirements are being formed for the software being developed. From testing and quality assurance perspective utilizing static testing is also extremely beneficial as often the defects found early in the software development life cycle are not only cheaper but also simpler to fix than the ones waiting to be discovered as failures during compilation or runtime of a program.

Static testing, also often referred to as static analysis, is also a well adopted approach when it comes to automated grading and assessment of programming exercise solutions [1; 14; 17]. The approach

and the related methods and techniques are also a growing research trend in the automation assessment field as observed by Paiva et al. in their review of automated assessment in computer science education [27]. One explanation offered by Paiva et al. in their review for the growing interest is that static analysis allows for a more human-like grading and feedback while also being more consistent in grading and feedback quality due to the automation. Other observations that support the use of static approach are perhaps more regarding its practicality: static analysis is often less demanding to perform overall as it does not require large-scale test suite scripting, setting up an environment and then executing the program to assess it. Due to this the approach is also said to provide some additional security aspects but also allows for assessing solutions that are only partly functional or unable to be executed successfully. [14]

Typical areas of an assignment solution to be evaluated and assessed through static means are for example (1) coding style, (2) programming errors, (3) software metrics, (4) design and (5) special features [1]. These are presented by Ala-Mutka in her survey of automated assessment approaches and their contents are partly represented in Table 3 to provide more insight into the individual areas.

Table 3: Areas of evaluation and assessment where static testing is often applied.

Static testing objectives	
Category	**Example objectives for analysis**
Coding style	Syntax Structural deficiencies Unused variables Language standards and best practices for readability Maintainability
Programming errors	Dead code Redundancy Logical errors Anti-patterns
Software metrics	Application size Lines of code Complexity
Design	Structural similarity Design patterns
Special features	Keywords Regular expressions Plagiarism

Gupta has also researched the use of static analysis for source code assessment purposes [14]. He observes that when static analysis is performed on source code, the process should start by generating an intermediate representation of the work product and then standardizing the representation to reduce diversity. The forms themselves could be characters and strings, abstract

syntax tree or graphs, for example. After this the static analysis can be carried out in so many techniques, but often in terms of testing the technique is to compare the work product to an example model and assess metrics such as similarity between them.

2.3.2 Dynamic testing

Static testing is often complimented with dynamic testing which involves testing the work product through executing the program to probe for failures and find defects. It is also worth mentioning that while one of the benefits of static testing was the fact that it allows for testing products that might not even be able to execute, in terms of programming assignments producing a solution that is able to be executed could be considered a desired minimum requirement, especially on an intermediate level course.

In Section 2.1. of this thesis it was also noted that when considering the objectives of software testing from the perspective of existing automated assessment tools, assessing assignments for functionality appears to be the most common approach. Testing for specific required functionalities is also often easier and more straight-forward to test by interacting with the program rather than attempting to analyse the flows in a static manner. Interacting also reveals the true behaviour of the program and in that sense dynamic testing is crucial for the non-functionalities, or the business logic, as well and therefore dynamic testing is an important and integral approach to consider when creating a solution for automatic assessment of programming assignments.

Dynamic testing is the approach that is usually conducted when test design and various testing techniques are being considered. Whereas static testing mostly takes the form of some sort of a review process with or without automated tools, dynamic testing could be said to culminate in the action of executing a designed test case. This is to be done by providing the system under test a specific input and observing if the output meets set expectations. [15; 23]

Testing and test design techniques within the dynamic testing approach are commonly divided into two categories. The division can be described to be based on whether knowledge of the internal structure of the system under test is required or not [15; 23; 33]: black box techniques that are designed based on system specifications and models, and white box techniques that are based on internal structure of the system and its components i.e., on the code. While this divide seems to be widely accepted in the industry, the techniques might not always turn out to be only black and white; often a test design technique might be a combination of the two and as such usually referred to as a grey box technique. Categorization into experience-based techniques [15, pp. 81] is also

sometimes used, however, in this study the theoretical categorization will only be done into black and white box techniques on the basis mentioned before while also accepting that techniques may be used jointly between categories to perform more thorough testing to achieve the desired testing objectives.

2.3.3 Black box techniques

Black box techniques are called as such because they are by nature data-driven and rely on input / output outcomes to produce test results [23, pp. 8-10]. The inner workings of the system are not visible or even of interest from a testing perspective and as such it is often imagined that the system itself is a metaphorical black box that only takes input and produces output without a view to what is happening precisely during this process. Sometimes these techniques are also referred to as specification-based techniques [15, pp. 82] or functional testing [25] techniques.

Black box testing is currently the more relied on category out of the two when it comes to existing automated assessment tools [33, pp. 1187]. A plethora of black box testing technique variations exist [33] but the most frequently agreed upon techniques of black box testing based on seminal works [15; 23] and software testing related studies [25] are: (1) equivalence partitioning, (2) boundary value analysis, (3) cause-effect techniques, (4) all pairs or pairwise testing and (5) error guessing.

The first technique, **equivalence partitioning**, aims to minimize the total number of test cases by partitioning the input domain into equivalence classes where representatives under a specific class can be expected to produce the same output when used as an input. Identifying these equivalences begins by identifying input conditions from the defined software specifications after which the conditions can be partitioned into groups. Myers proposes a few heuristics [23, pp. 51-52] represented in Table 4 [pp. 17] for identifying equivalence classes.

Table 4: Myers' heuristics for identifying equivalence classes.

	Number of equivalence classes to be identified	
Identified input condition	**Valid**	**Invalid**
A range of values	1	2
Specified number of values	1	2
A set of input values + a reason to expect each set being handled differently by the system	1 for each set	1 for each set
A "must-be" situation	1	1

To provide a simple, but concrete, example of applying the technique in web application context we can consider the following specification for a text input element: *The username input field must only allow for strings that consists of alphabet characters. The input string can only be longer than or equal to 5 characters, but no longer than 12 characters.* From this specification we would be able to deduce the following contents presented in Table 5.

Table 5: Valid and invalid inputs extracted from a specification by following the Myers' heuristics.

Input condition	Valid input	Invalid input	
Input string must consist of alphabet characters only	for each character in toLowerCase(input): character ∈ {a, … , z}	for any character in string.toLowerCase: character ∉ {a, … , z}	
Input string is between [5 - 12] characters in length.	4 < length(input) < 13	length(input) <= 4	length(input) >= 12

The example also demonstrates the technique can narrow down the amount testing required to verify functionality through assuming that every combination of alphabetical characters is valid if it fits the length requirement and is vice-versa invalid if even one of the characters is included is non-alphabet or does not fit the length requirement range.

Boundary value analysis as a technique builds on top of equivalence partitioning. As a technique it exploits the knowledge that the edges - or boundary values - of equivalence classes are usually where errors causing defects are more often discovered [15; 23]. The most notable difference to the previous technique is that not all elements within a class are equally representative of that said equivalence. Instead, the edge values of a class are taken as representative elements. To tie this with the example already used, with boundary value analysis we would only perform our test cases with input strings 5 and 12 characters in length instead of assuming that string of any length between 5 and 12 would do to test the functional validity.

This technique may also be applied in different variations. Both bottom and upper boundaries may be tested, and the boundaries may be tested for only valid or for only invalid values. The most thorough way of applying boundary value analysis is to perform a so called three-point analysis where tests are targeted and expected results validated for exact boundary value, value directly above and value directly below.

The technique understandably may create more tests for a given function that basic equivalence partitioning, but it is also considered to be more able in catching errors. It's also worth to note that boundary values are not always present and as such the use of boundary value analysis is not always an option even if equivalence class technique can be used: for example, any classification of non-ordinal objects are rarely potential targets for boundary value analysis.

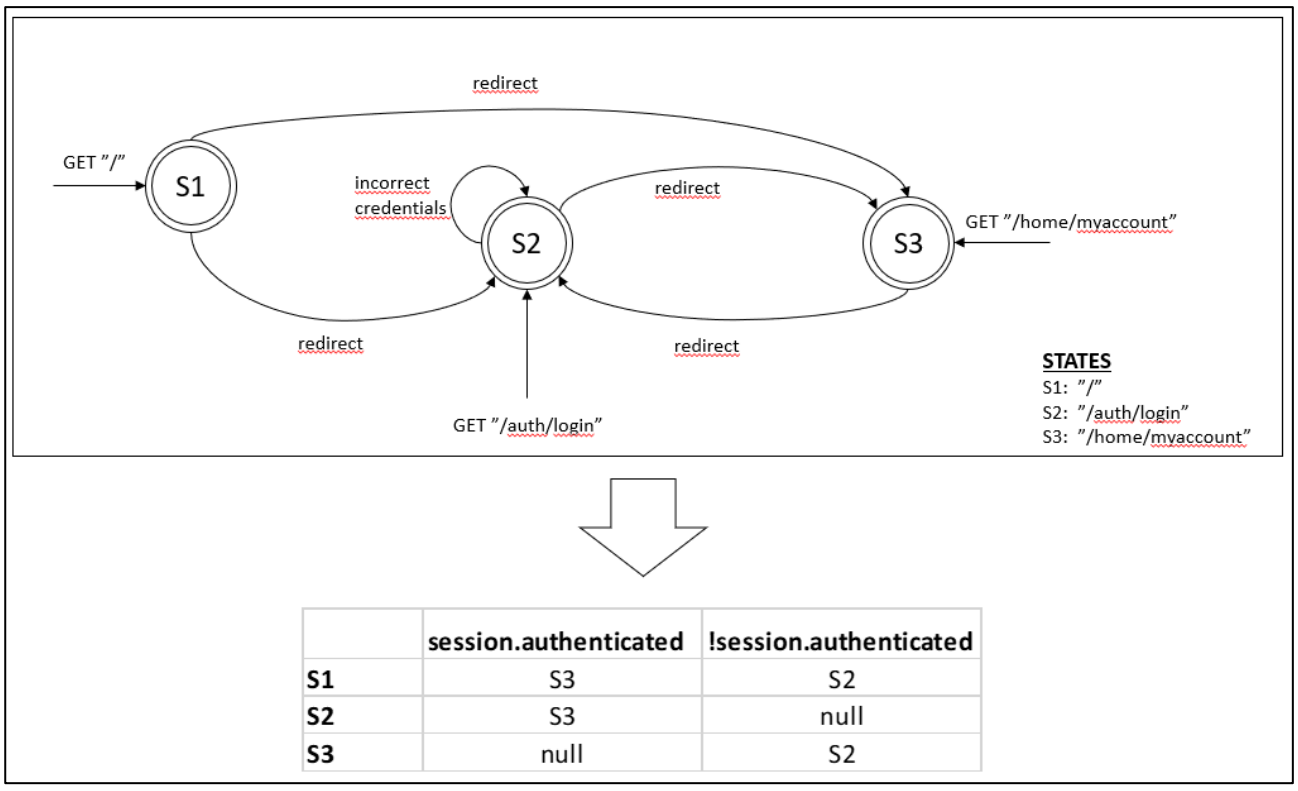
Equivalence partitioning and boundary value analysis are good techniques for limiting the amount of test cases in specific situations where the input data or output results can be expected to act in parts equally and as such can be also classified. However, this can also be undesirable; sometimes it might be needed to explore potential input combinations to search for errors. **Cause-effect techniques** in this study refer to a set of techniques that aim to accomplish the exploration of possible input combinations and the resulting state transitions and outputs.

Combinations can be explored meticulously through graphing techniques such as cause-effect graphing [23, pp. 61-80] but in practice - especially if the system under test is relatively simple - the mapping is done by collecting a set of conditions and expected outcomes into a decision table such as the one in table below. Table 6 [pp. 19] represents a minimalistic example of a decision table created based on a login page of a web application.

Table 6: A decision table based on a typical login page elements.

Condition	Rule 1	Rule 2	Rule 3	Rule 4
Username correct (True / False)	F	F	T	T
Password correct (True / False)	F	T	T	F
Expected output				
Redirect to "/home/myaccount"	F	F	T	F

State transition testing and use case testing [15, pp. 91-96] are also considered to belong into the category of cause-effect techniques and will be treated as such in this study. Transitions and use cases are useful from the perspective of web application testing in a sense that page transitions and redirects offer natural basis for designing state transition diagrams. The state transition graphs are then converted into state tables, as in the example shown in Figure 1, which are then used to produce test cases.



	session.authenticated	!session.authenticated
S1	S3	S2
S2	S3	null
S3	null	S2

Figure 1: An example state transition diagram of a web site.

Cause-effect techniques are advantageous in a sense that they can focus, visualize, and make clear of the expected system behaviour at a very high level which provides useful basis for testing what most likely matters to the end-user. While these techniques may be useful for a system with limited

states and complexity, such as the authentication-based state transfer illustration above, using them may quickly become tedious for illustrating and mapping larger systems unless the process can be aided with automated tools.

Software testing is not all graphs and tables, however. Often experience and intuition of the tester plays a great role in hunting for errors and that is also why it is not too rare to find certain techniques labelled under the label of "experience-based", as mentioned at the end of Section 2.3.2. One such black box technique is **error guessing**, which foregoes the afore mentioned other techniques to simply create test cases for errors that are deemed probable in the given context based on experience and intuition [15, pp. 118-119; 23, pp. 80-81] of either the tester or the collective consisting of testers and other stakeholders.

While such a technique may feel unreliable it is also important to realise that none of the techniques are mutually exclusive but rather complementary of one another when applicable. Error guessing can be especially potent technique for creating test cases from the perspective of programming assignment assessment since the lecturers and assistants involved with any given course may have experience regarding hundreds if not thousands of assessed assignments throughout the years the programming course has been taught.

2.3.4 White box techniques

Dynamic white box techniques are concerned with logic coverage: the basis for these techniques comes from the internal structure and the paths, statements, decisions, or conditions that present themselves within the source code. According to a recent review [27], white box testing has been used for marking an assignment solution source code on runtime, but it appears not to be fielded in any serious way to test the functionality of student assignments.

The nature of dynamic white box testing is most likely the reason it is also not very widely used by automated programming assignment tools. To make the techniques useful and create a set of predetermined set of tests to assess the internal structure of a student solution, one would have to know how the solution will be coded by the learner. However, dynamic white box techniques may have some limited use if hard requirements for certain elements exist in the programming assignments: testing-wise it is then possible to expect certain methods, variables, or elements to exist in the code structures and test cases for functionalities taking advantage of these hard-requirements can be created. In general, white box techniques can be divided into statement, decision, and condition coverage techniques [15, pp. 97-116; 23, pp. 42-49]. Flow charts and

control graphs formed from the code structure often prove themselves as helpful mediums to create test cases and assess required coverage.

Approaching the coverage through statements is called **statement testing**. Statement testing aims for a full coverage of every executable statement within the code that is under test, which makes it somewhat usable in verifying that the code can execute as it should but is otherwise proposed to be not very meaningful as a lone testing technique. Therefore, statement coverage is often overtaken with the more useful technique of decision coverage.

Decision testing technique in many cases fulfils statement coverage as an in-built feature. Decision testing aims to hit every possible path or branch within the code logic at least once which means that 100% decision coverage should also gain 100% statement coverage unless the program is such that there are, for example, no decisions or multiple entry points to the program or its subroutines exist.

While decision coverage can be considered stronger than statement coverage, it is not always enough either. Decision testing in its purest form is good for decisions that continue into two possible paths - e.g., true or false - but require additional cases to be created for handling decisions with more than two possible decisions, for example switch -statements. [23, pp. 43-46] To tackle such issues, white box technique of condition coverage may be used.

Perfect **condition coverage** consists of enough test cases to test every possible outcome of every decision at least once. While this technique again is a step up from decision testing in terms of meaningful coverage, condition coverage fails frequently in reality to truly achieve the goal of testing every possible outcome within the code structure by simply hitting every possible statement condition once. This is because certain condition combinations - especially with multiple condition statements - often cause situations where certain condition combinations have satisfied the requirement of testing each condition of that statement once, but some paths have been left explored afterwards since they were not reached with these condition combinations.

To really achieve full coverage of all statements, decisions, and conditions of the program, one needs to apply the **multiple-condition coverage**. The approach of this technique is to create test cases enough so that for each executable decision all possible condition combination outcomes and all program and subroutine entry points are tested at least once.

The reality of these techniques is, however, that often techniques such as multiple-condition coverage are not able to reach 100% coverage simply due to the number of resources it would take

to create the required amount of test cases. Condition combinations, for example, can easily become so numerous that testing all of them is not feasible unless critical for assuring certain non-functional requirements such as security related ones are met.

2.4 Test design and development

According to Myers et al. the key issue to consider when designing test cases is: “What subset of all possible test cases has the highest probability of detecting the most errors?” [23, pp. 41] In regard to automated assessment of assignments this piece of wisdom would most likely need to be transformed into such a perspective that the issue is to not necessarily detect most errors but detect the most relevant errors to form feedback to support the growth of learner’s skills and knowledge as well as verify to which degree the student solution manages to meet the assignment requirements.

Nevertheless, for test cases to be effective, efficient and drive their purpose, they need to be designed. The design process brings together all the aspects that have been discussed so far: test objectives, analysis, consideration of testing levels and choices regarding proper approaches and techniques to be made. The process of design starts with identifying the test conditions, continues to specify test cases and finally specifying the test procedures. [15]

Identifying the test conditions means mapping out what characteristics of software should be checked and verified by testing. These can and should be gathered from the software specifications such as requirements and other related work products. Conditions may of course vary depending on the context and scope of testing: conditions to be found from unit level are rarely equal to conditions to be found on system level, for example. Here the static analysis methods are also useful as deploying them is often the way to gain required insight for identifying conditions of the software under test.

Once the conditions to test are clear, a test case to carry out these conditions can be created. In many cases already existing work products may be of use here as well because work products such as user stories for example may be able to provide structure for the test case to follow, especially with the higher-level test cases. As for designing the test cases themselves, prior described white box and black box methods can be followed.

Finally, once a test case has been designed it can be built into a test procedure to be executed as to verify the expected outputs from identified conditions. The overall process of test development is quite simple when arranged into these three main steps, however, the process needs to be gone

through a volume of times to produce enough test cases for any meaningful amount of test coverage.

Some strategies exist to heuristically approach the decision of which techniques to field for creating test cases and suites. One such heuristic is “The Strategy” [23, pp. 82] that dictates the following when applied to what we know of testing techniques already:

1. For combination of input conditions, cause-effect technique should be used first.
2. Boundary value analysis should always be used.
3. The above techniques should then be supplemented by identifying valid and invalid equivalence classes for both the input and output values.
4. If enough experience, supplement techniques in 1. -3. with error-guessing.

Finally, examine the program logic and deduct if white box techniques are required to reach the desired coverage; apply decision coverage, condition coverage, combination of both or multiple-condition coverage as required to satisfy the set coverage criteria.

The concept of coverage when developing tests is an important one. The aim of test coverage is to quantitively assess the extent and quality of testing [19, pp. 80]. The meaning of coverage needs to be defined, though, before any percentages of coverage can be attempted to achieve; for structure-based testing conditions and statements may prove to be relevant metrics to measure test coverage but usually the more meaningful coverage metrics may be the number of requirements verified.

2.5 Challenges of web application testing

Due to their nature web applications present certain challenges to testing if compared to testing of so-called traditional software. Web applications are often mentioned to be considered as distributed systems built with various architectural choices. Some typical characteristics for such applications mentioned, for example, by Di Lucca and Fasolino are: (1) concurrent accessibility by many users, (2) varied execution environments, (3) systems often consist of components that may vary in their nature and even technology and (4) ability to create software components at run time. [8, pp. 220]

These characteristics are mentioned to inherently place certain testing requirements for commercial web applications in terms of non-functional aspects such as performance, availability, and security testing. Testing for functionalities is also affected by the large variance of components and separate services involved in a web application: test environments may often need to be set up to consider

multiple different technology choices and dataflows. As web applications often include both server-side logic and client-side logic, test environments need to take this into consideration when testing at system, integration or even at unit level: sometimes to test even the smallest component of an application, input or interaction with a server may be required. [8] However, ideally real backend interaction should not always be relied on especially at the lowest levels of testing as then the nature of testing itself changes and the focus changes if end-to-end services are employed.

Di Lucca and Fasolino also note that the client-server nature of web applications also often means that points of failure are plenty. This poses certain technical challenges when conducting tests at system or acceptance levels. [8] In complex and market level products pinpointing failures may not always be easy as issues may rise from client-side or server-side code interpretation, compatibility issues or from relied on backend services for variety of reasons. Common elements to note from server-side layers when testing web applications is the persistent data storage and API integrations. With more complex systems there is often also the need to consider server side load balancing, but in the scope of this study more relevant might be the contextual JavaScript generation which to a degree applies to both sides in modern web applications: JavaScript may be offered to the client quite dynamically from the server depending on the context of use but it also means that the client-side in-browser application renders different content depending on the context.

Many of the challenges involved in testing of web applications can be approached and explained from the aspects of observability and controllability. These aspects can be defined as follows:

- Observability: “How easy it is to observe the behavior of a program in terms of its outputs, effects on the environment, and other hardware and software components.” [2, Section 3.1, Definition 3.11]

- Controllability: “How easy it is to provide a program with the needed inputs, in terms of values, operations, and behaviors.” [2, Section 3.1, Definition 3.12]

In essence, observability dictates the difficulty of determining test results. With web applications and their multi-tiered architectures, true results of tests conducted at a higher level are rarely fully available and visible from the UI which in turn results to lower general observability of testing. Same can be said of controllability: rarely all web application testing can be executed by simply feeding inputs through a single source such as application UI; providing test values may require manipulation of URLs to feed parameters or manipulation of client storage solutions such as local

storage, session storage or cookies to properly execute a test case. Such things result into lower controllability and are not uncommon for testing web applications.

Of course, the challenges mentioned above are not only brought up in studies by the authors cited but also presented in very foundational works such as *The Art of Software Testing* by Myers. He proposes that to tackle the testing challenges of internet applications one needs to first and foremost understand the system under test at the very component level. This proposal is further clarified to include having documented knowledge available and understanding the expected behaviour of functionalities and performance of the website. [23, pp. 193-200]

Myers also outlines a strategy that relies on categorizing internet applications into three-tiered client-server applications: (1) presentation layer where the user interface is provided, (2) business layer that models processes such as authentication or transaction, and (3) data layer which considers the data application uses or is collected from the user [23, pp. 201]. These layers are encouraged to be tested independently to be able to narrow down and identify defects and their sources during testing; skipping the layered approach and conducting overarching end-to-end system tests instead may not tell where a defect springs from.

Not all the challenges mentioned in this sub-section of course necessarily apply to their fullest when supporting the assessment work of assignments with software testing activities, especially the non-functional challenges related to concurrent users and web application availability. However, even assignment tasks often include interaction between client and server and thus present these challenges at least to a degree. Handling such aspects is also identifiable to be one source of for errors in assignment submissions [22] and as such taking these into consideration during testing is relevant even for this purpose.

2.6 Foundation for the first main research question

Throughout this main Section there has been an overarching goal to cover the general theory behind software testing while also narrowing it down to the context of this study by taking into consideration how it would be applicable to DTEK2040. With the combined approach to covering the theoretical background there was also an agenda to gain answers for the sub-questions related to the first of the main research questions.

In the very basic sense software testing was determined to be about identifying defects, getting rid of the errors causing these identified defects and through that process raising the overall quality of a software product. While quality of the product is one thing, testing was also determined to build

confidence for those working on the product by providing them a safety net and possibly allowing for a more stress-free product development.

Software testing for programming assignment assessment purposes was mentioned to not pursue the quality aspects but to rather catch deviations from the assignment requirements and thus provide a supportive tool or a method for forming both formative and summative assessment of assignments.

2.6.1 Q1.1: Which testing levels should be focused on?

The first sub-question deals with the abstraction levels of testing and in essence is asking where the testing effort should be directed to best serve assessment purposes. Based on the foundational theory and the contents of the expert interview describing the details of assignments in DTEK2040, the most useful levels of testing would be unit testing and system testing.

Reasoning behind the focus on unit level testing is that the assessment in DTEK2040 is for a large part mentioned to consist of a set of clear functionalities. Assessment is then carried out so that a working functionality scores the student a point and the contrary results in missing a point from the total available. In many cases this will mean checking the existence and behaviour of an individual component that the student should have managed to create following the course material examples.

Focusing on system level testing is valid from multiple perspectives as well. For example, based on the expert interview it was clear that whether an assignment requires the student to create a React SPA or even just a single static html web page, the very basic expectation is that the student submitted work product should be able to compile and run or interpret properly. Another reason to focus system testing is that when assignments start dealing with the challenging issues of web applications, such as state transitions and data flows across multi-tier architecture, these can naturally be covered in system testing by implementing end-to-end tests or user interface tests. Overall, the choice to focus on unit level and system level testing also provide further guidance to choosing testing methods and techniques. This brings us to the next sub-question Q1.2.

2.6.2 Q1.2: Which testing techniques are applicable for testing student submissions?

As for testing methods in general, it was mentioned that a rather clear division into static and dynamic testing can be made. Static testing deals with test objects without requiring execution of any system-under-test code and was also noted to be a trending methodology when it comes to automated programming assignment assessment solutions. One technique to field static testing for assessment purposes is to simply perform model-based comparison of student solutions to model

solutions, given there is not much expected or allowed deviation in the submitted work products. For static assessment of source code, it was noted that a general technique is also to first transform the code into an intermediate representation such as characters, strings, abstract syntax tree or graphs to reduce the potential diversity and then perform the assessment against set metrics.

Static testing was described to be often complimented with dynamic testing that involves executing the system-under-test code to test it. Dynamic testing was also the approach of choice when dealing with functional testing, which also raises its importance for assessment purposes given the testing levels that were placed into focus when answering Q1.1. Techniques within dynamic testing were further categorized into black box and white box techniques based on the required knowledge about the inner workings of the system under test. White box techniques were not considered to be very useful for dynamic testing and traditionally were not favoured in automated programming assignment assessment tools either.

From the black box techniques covered during this Section, most suitable ones for web application assignment assessment purposes in DTEK2040 are perhaps the cause-effect techniques and the error guessing technique. Cause-effect techniques were noted to be effective in covering aspects and functionalities related to state transitions and expected outputs from multiple input combinations. These were also the aspects identified as being natural for web applications in general and very suitable candidates to be tested in system level testing. Error guessing, on the other hand, as an experience-based black box technique can transform the current knowledge of the course personnel into test cases that target the most-likely sources of errors in specific assignments. Therein also lies the challenge: designing and implementing experience-based test cases would most likely require very close collaboration with the course personnel who have the necessary expertise to say what kind of errors often appear in student works. This kind of collaboration within the scope of this thesis is not necessarily possible, given the scope, constraints on time and other resources.

White box techniques, though already mentioned to be less likely to be used, could still be considered for unit tests depending on how accurately the course material is expected to be followed in terms of the student solution to an assignment. From the interview it was gathered that the current material for DTEK2040 strongly guides the students to craft their submissions certain way, but some deviations or algorithmic level leeway is still possible, which at least complicates building very rigid white box technique-based tests for assessment. Thus, these are not considered to be very applicable either.

2.6.3 Q1.3: How to turn an assignment briefing into test cases?

In terms of test design and constructing test cases it was deduced that in general the goal would be design test cases so that they are able to maximize the probability of capturing errors. For assessment purposes it was also noted that this goal would be best understood slightly differently: the goal is to capture specific implementation errors based on tasks within the given assignment rather than attempting to cover all types of errors from the system under test.

A strategy was proposed in Section 2.4. which relied in starting with cause-effect techniques to map out a combination of test inputs and then proceed to supplement these techniques with boundary value analysis, applying equivalence classes and possibly perform additional error-guessing. Finally, white box techniques were suggested to be used as needed if the black box techniques cannot cover the program logic with desired coverage.

The proposed strategy seems to be for the most part appropriate and can be followed within the scope of this study to design and build test cases for automated assignment assessment support. Black box techniques, and especially cause-effect techniques, were already mentioned to be suitable techniques through the answers to sub-questions Q1.1. and Q1.2. However, techniques such as boundary value analysis may not always be required when assessing submissions since the functional requirements may not be so detailed that they set clear boundaries; though, they could be in the future as the course contents are updated or the proposed solution of this study is possibly applied to other kinds of programming assignments.

From coverage perspective the test cases should cover the required functionalities of any given assignment. The coverage should be enough to provide basis for at least deciding if the required functionalities are “pass” or “fail” to support straight-forward scoring. Coverage should also be considered and designed so that it would allow for spotting errors that do not necessarily result in a failed functionality but could be considered a qualitative or non-functional error when observed within the context of course material. Covering such cases is also the more challenging part in terms of design and where the experience-based techniques can prove valuable to provide insight as to what should be tested.

3 Test automation and RPA

3.1 Differentiating between RPA and test automation

One way to describe test automation is to say that it "is the task of creating mechanically interpretable representation of a manual test case." [35] Automated cases may be programmed with a programming or a scripting language and many languages, such as java and python for example, have evolved extensive support in the form of libraries to make test automation relatively easy and straight forward to implement. Frameworks built for automation exist as well; some market themselves as geared towards test automation [32] while others consider themselves more focused on RPA in general [31].

But how does RPA and test automation truly differ? Considering both can - and often are - executed with same tools and technologies, one might argue that the differentiation is at times more philosophical and deals with the context automation is being fielded and aimed to be used. RPA can be described as technology that aims to mimic human behaviour to achieve benefits such as reduced labour costs, increased productivity, reduced error rates [9]. Perhaps therefore RPA is often tightly coupled with the mental image of being a tool to automate general business processes by tackling many, traditionally human executed, mundane, and transactional tasks involved within said processes.

Classical automation works best when the process to be automated has explicit rules that can be followed. Leaning on this, Doguc for example mentions that best-suited processes to automate with RPA have (1) high transaction volume, (2) are highly standardized, (3) have well-defined implicit logic and (4) are mature, meaning that an automated solution will be usable into the future rather than becoming obsolete due to changes in the process structure or functions. [9] Jha et al. propose to [20] implement automation by following sequential levels that start from performing due diligence and end up with execution and maintenance. Interpretation of contents involved with these levels are shown in Table 7 [pp. 30].

Table 7: Sequential implementation levels of automation.

Levels of RPA implementation	
Level	**Contains**
Due diligence	Deciding tools that are a good match for the project. Investigating automation viability of processes and determining the return on investment. Technical feasibility assessment with a proof-of-concept.
Risk identification	Deciding whether a process is a preferred candidate for automation. Identifying stability repetitiveness level of organization / standardization
Bot creation and dry run	Identifying the steps and tasks that are to be automated and robotized. Performing a “smoke test” for the automated process to prove the steps can be carried through and the process itself is correct.
Execution and maintenance	Deploying the bot for execution. Maintaining dynamic parts of the process.

Just like Doguc, Jha et al. also underlines the importance of choosing steady, repetitive and highly organized processes as candidates for automation. They also mention these aspects as basis for risk identification, as can be seen from the description for the corresponding level. It is also worth noting that the process itself is unlike a waterfall: if bot creation and dry run, for example, fail to produce the expected successful outcomes for that level, it is entirely advisable to step back and perform risk identification or even due diligence levels again to further analyse the system under automation.

While automation is mentioned to offer concrete gains even as a short-term solution, it contains a challenge in that it is closely tied to the aspect of identifying potential use cases. Rarely any process is forever unchanging or without any dynamic parts and thus any automation solution that is to be built will also require maintenance to keep producing benefits as a long-term solution. Integrating artificial intelligence with RPA is considered as a potential supportive factor in the future, but as of now automated systems with artificial intelligence have not proven cognitive enough to meaningfully remove this challenge. [9; 20] RPA is also still seen to have somewhat limited use in a sense that fully implemented end-to-end automation solutions can be considered unrealistic in terms of resources required to build them. [20, pp. 256]

From RPA perspective and based on the results of expert interview, the assignment assessment process of DTEK2040 certainly offers potential steps to be automated. To describe the general process executed in a concise manner, following steps can be identified from expert interview [22]:

1) Student returns the solution for an assignment on course Moodle workspace.
2) The assessor downloads the student submission from Moodle.
3) The assessor extracts the downloaded submission to access the assignment files.
4) The extracted submissions are either run dynamically or opened with relevant tools such as VSCode.
5) Assess the submission both dynamically and statically against the task requirements. Form feedback based on observed errors.
6) Enter and upload the assessment results and feedback for the student to Moodle.
7) Clean up tasks and organize the assessed works to prevent mix ups when assessing other submissions for the same assignment in the future.

To further attempt to analyse the actual automation potential, each step can be transformed into a more high-level description and be identified to involve specific tasks from the assessor point of view. These tasks have been extracted from the interview and presented in Appendix A correspondingly while also attempting to present the related automation potential and risk based on theoretical background. A concise collection of overall estimated automation potential and risks for DTEK2040 are represented in Table 8.

Table 8: Automation potential and identified risk factors of assignment assessment in DTEK2040.

Automating the DTEK2040 assessment process	
Potential	**Risks**
a. Manual tasks related to navigating and fetching submissions from the learning platform (i.e. Moodle) are straight-forward to automate and can save work time in a compounding manner.	a. Automated solution to interact with the learning platform and the course workspace requires maintenance; even the simplest modifications to a website element may break the robot by affecting, for example, the navigation logic.
b. Automating manual tasks such as recursive extracting of submission files, book-keeping and collecting scores, summarizing feedback forms can save hours of manual work throughout the course.	b. The learning platform may not allow for robotic interaction or use CAPTCHA and other means - such as request limits - to hinder the use of RPA.

(to be continued)

Table 8 (continues)

c. Technology such as Docker can be in-built to the automation solution to provide a stable and secure environment for dynamic assessment. d. Automated static assessment can be more consistent and less reliant on assessor's experience than manually performed. e. The tasks required to perform manual labour in an organized manner, such as arranging submissions in local directories based on their assessed / not assessed status, can be cut out from the process.	c. Scripting course and assignment content specific static analysis logic for robot assessment purposes may prove to be challenging and not worth the return on investment. d. Automation itself may not be able to formulate in-depth, qualitative feedback from submissions. e. Introducing RPA and test automation to the assessment process requires related skills from the personnel to maintain the solution as course contents are updated. f. Incorporating automation may require adding to or reformatting of assignment instructions.

As shown in the table contents, DTEK2040 has a lot of identified automation potential related to even small individual manual tasks from simple website navigation to performing file manipulation and looking for errors in a source code file. However, there are also a lot of risks that most likely will stand to make a fully automated end-to-end solution a challenging task and not feasible in terms of return on investment.

3.2 Use of automation in web application testing

Amman and Offutt describe test automation as "The use of software to control the execution of tests, the comparison of actual outcomes to predicted outcomes, the setting up of test preconditions, and other test control and test reporting functions." [2, Section 3, Definition 3.9] They consider automated testing to be necessary for efficient and frequent testing but also mention that the task of automating may often prove challenging in the case of software with low controllability or observability. Some studies claim that the advantages to be achieved from using test automation include saved resources in terms of time and effort spent in making testing more efficient, improved accuracy and discovery of defects compared to manual efforts, increased test coverage and repeatability [12; 36].

As for web application testing, the related challenges and the overall process of coming up with a test design has already been covered as those are aspects that are applicable to software testing in general. The automation of web application testing brings along some nuances as the principles of

automation are combined to the art of testing. Some of these details are related to details such as what kinds of tests should be automated, what tools should be used to do so and what kind of basis for testing should be available to start implementing test automation.

Considering automation from the perspective of software and testing levels, automating unit and integration tests is common work during software development. In terms of responsibilities involved in testing activities, unit and integration level testing is usually attributed to developers as part of the development tasks whereas designated testers often focus on system level testing and beyond. These testing tasks may include functional as well as specialised non-functional tests such as security or performance testing.

At system level, attractive targets for test automation can be observed to follow the general principles for potential automation targets where the most likely candidates for automated tests are the ones that require a lot of data handling, are performed constantly and regularly, or require extreme precision during the test execution. Such tests are for example regression, end-to-end, performance, security, load, stress, and many of the usability tests [30]. User interface testing is an example type of testing that can be involved in many of the tests just mentioned and thus can have a fairly large role especially when it comes to testing web applications. [12; 30; 35] Going by the layered tiers presented by Myers, user interface testing would fall under the presentation layer and have three main test areas [23, pp. 203-205]:

- Content: testing the human-interface element, accuracy of the information presented to the end-user and features affecting the user experience.
- Architecture: testing navigational and structural errors such as broken links, missing pages or false redirects.
- Environment: testing aspects such as browsers and operating system configuration effects to web application and its functionalities.

All these test areas are likely to contain candidate cases for test automation but Myers himself proposes to at least migrate architecture tests into regression tests, which itself is already mentioned as one of the automation test examples earlier based on other cited sources.

There is a plethora of technology and environment specific automation tools available for conducting software testing. While commercial tools of course exist, many of the industry favoured tools are in fact open source and readily available. Among these tools the so called XUnit

frameworks are mentioned to be the most used ones. These frameworks provide the means to write test cases in a supported programming language - such as JUnit for Java or HtmlUnit for HTML - so that the tests can be implemented with oracles to determine whether a given test passes or fails as it is executed against the system under test. [28]

Another category of testing tools is the capture and replay tools which are in fact able to combine manual testing and automation testing to some extent by recording the manual actions to be performed automatically and repeated later as required [28]. However, even though tools such Selenium can be assigned to this category, they also provide the means to simply script actions rather than requiring the manual recording of a test case to turn it into an automated one.

Main tool to be used within the context of this study was mentioned to be the RF, which would perhaps fit into a third category not labelled among the two already presented by Polo et al. RF is a more generic test automation framework that can be used to create and execute automated tests by extending different libraries meant for specific purposes [32]. Some of the libraries that are available are created and based on tried-and-true web application testing technologies such as the Selenium based SeleniumLibrary or Playwright based BrowserLibrary. These example libraries often provide means for acceptance test driven automation testing and as such they are most suitable for testing that can be conducted through the web application user interface. Many other libraries are of course available that are better suited for cases that should be conducted by the means of API testing, for example. As with any web application automation tool, the potential and suitability of different libraries are to be considered before beginning test automation implementation tasks.

3.3 Automated formulating of feedback from an assignment solution

Hattie and Timperley describe feedback as information that is provided regarding aspects of performance or understanding of a given subject. Many different entities and sources can act as providers of such information but the feedback itself should in any case aim towards improvement of teaching and learning. [16] Paiva et al. on the other add to this notion by adding that assessment already in itself acts as feedback not only for the student but for the teacher also: the one learning will be kept aware of their success in reaching set learning goals and the teacher will be informed about the ongoing learning process in general [27].

Nicol and Macfarlane-Dick seek to discover principles of effective feedback in their study regarding formative assessment and self-regulated learning [24]. As the short description of this

study would suggest, the viewpoint taken towards what is good and effective feedback is that it should also help the students grow and guide their own learning in the future as well, not only in the framework of a specific study session or a single course. The study or its results are not per se about assessment of programming studies or problem-based learning, but the outcomes contain observations that seem quite general in terms of what could be considered as effective feedback.

The principles presented in the study by the authors can be categorized so that they have either a cognitive, motivational or behavioural rationale behind them. The seven principles are [24]:

1. Helps clarify what good performance is.
2. Facilitates the development of self-assessment in learning.
3. Delivers high quality information to students about their learning.
4. Encourages teacher and peer dialogue around learning.
5. Encourages positive motivational beliefs and self-esteem.
6. Provides opportunities to close the gap between current and desired performance.
7. Provides information to teachers that can be used to help shape teaching.

Principles 1 - 4 are presented by the authors from a very cognitive standpoint. The first principle is rationalized through the potential mismatch existing between the teacher's and the student's concepts of (a) what are the goals for learning, (b) what are the criteria for evaluating the learning process and (c) what are the expected standards. This mismatch is through existing research seen to negatively impact the student's ability to process received external feedback in a constructive manner and thus any feedback should aim to align the understanding of these concepts between the teacher and the student if it appears to be necessary. [24, pp. 206-207] The second principle in a way extends from the first one by noting that external feedback should also allow the student to develop their ability to individually judge one's own product against set standards and criteria, provided they are also clear for the student. [24, pp. 207-208]

The third principle of delivering high quality information is an interesting one in a sense that it considers more than just content. This principle considers high quality, effective feedback as one that is: (1) provided in a timely manner, (2) focused on strengths, weaknesses and corrective advice, and (3) contains both praise and constructive criticism. [24, pp. 208-210] Especially the timing as a key-component is something that can sometimes be observed missing from the external feedback being provided in courses with large amounts of participants and similarly large number of assignments to be assessed.

The fourth principle deals with the cognitive aspect of external feedback in quite a general manner. The principle attempts to underline that even though feedback may be of high quality and follow the rest of the principles uncovered prior, it can still be misunderstood by the student receiving the feedback. If feedback is misunderstood, it is often also at least partly ignored by the student. Thus, it is suggested that feedback should also incorporate an opportunity for engaging the teacher into discussion about the feedback to catch and potentially clear up any confusion or concern regarding the received feedback. [24, pp. 210-211]

The fifth principle is the first one that is clearly approached through the lens of motivation, as the short descriptive name also indicates. The core rationale behind this principle is that high-stake assessing, such as one-time assignments or traditional exams, is often found out to negatively impact the motivation of a student and should thus be avoided as the only channel of feedback. [24, pp. 211-212] Nicol and Macfarlane-Dick refer to existing research to show that such assessing usually leads to the students focusing on performance in a very metric-focused manner rather than attempting to master the concepts and achieve a more sustainable learning process. Relying purely on grading or marks as the form of feedback from assessments was also mentioned to have a negative motivational effect. Accompanying such feedback with comments was mentioned not to improve the impact since the numerical mark or grade was usually focused on and the supplementing commentary ignored by the students.

The authors suggest that comments alone without grades are the general superior format of external feedback in terms of encouraging positive motivation and self-esteem. Additionally, multiple assignments and tasks with low-stake assessment should be favoured. Many smaller tasks carry the benefit for the student in terms of providing the opportunity to receive more external feedback in concise pieces. Automated testing with incorporated feedback is also explicitly mentioned as a potential approach to help pursue this principle. [24, pp. 212]

The sixth and the seventh principle take a behavioural approach to the feedback process. To provide an opportunity to essentially catch up to the learning expectations: the student needs to have an opportunity to resubmit an assignment or at least repeat the learning cycle based on the contents of a received external feedback to put the feedback in use. If resubmissions cannot be offered or the assignments in a course are such that feedback received from an assignment will not directly carry over to the next assignment, then it would be beneficial to provide the students feedback while any assignment is a work-in-progress. [24, pp. 213-214] The final principle on the other hand raises the observation that any feedback that is provided to the students should be based on such assessment

or data that can be used to frequently deduce the learning level and the understanding of course contents. This in turn should provide opportunities to improve the teaching, course contents and the overall learning process of the given course.

When these principles are reflected to DTEK2040, it would seem like they are partly followed. For example, the first principle could be considered to manifest itself through the detailed assignment tasks which very illustratively and straightforward manner describe the features that are expected to be implemented as part of an assignment [37]. Additionally, during the interview feedback was a theme to be discussed and during this part to specific mention was made about any excessive need to ever provide the students with clarification about taskers; on the contrary, it was in many situations noted that the course material is quite explicit in what is being required from the student's solution [22].

On the other hand, based on the observations of feedback samples from the DTEK2040 Moodle-platform and the interview with the course instructors, the feedback models and the process would seem to partly go against the principles number three, four and five. The third principle is in a sense being broken against since timely feedback is not necessarily possible. However, this is not necessarily even because of the feedback workload or for the fact that an instructor would not have the time to provide feedback: because DTEK2040 allows the student to complete the content and continue onward at one's own pace, feedback from the previous assignment may not be available when the student continues to the next one. Thus, there is a chance that some feedback will not be affecting the next solution.

Fourth and fifth principles are also something that are not actively being carried out in the feedback process of DTEK2040. Regarding the encouragement of teacher and peer dialogue, no mentions arose during the interview related to this and providing the assignment specific feedback on the course's workspace in Moodle does not seem to integrate or provide means for the student to engage the feedback provider into a discussion. This sort of exchange could of course currently happen directly through email if the student decides to do so. With regards to the fifth principle of encouraging positive motivational beliefs and self-esteem, DTEK2040 could be mentioned not to strictly follow this principle due to the provided feedback being very grade centric. While it is true that the feedback includes commentary if a student has not managed to score the perfect grade from an assignment, the combination of comments and scoring was mentioned to lean towards negatively affecting the motivation rather than being any different from purely grade-based feedback.

The principles gathered by Nicol and Macfarlane-Dick are, as mentioned, regarding feedback in a more general manner while taking a certain point-of-view to consider the effectiveness in terms of also developing self-regulated learning of the student. Keuning et al. in their systematic literature review then again present five feedback types regarding automated feedback for programming exercises specifically [21]. The feedback types are:

1. Knowledge about task constraints
2. Knowledge about concepts
3. Knowledge about mistakes
4. Knowledge about how to proceed
5. Knowledge about meta-cognition

These feedback types by Keuning et al. have a lot of common surfaces to the seven principles of Nicol and Macfarlane-Dick. For example, the first type consists of components such as requirements of the task and general processing rules of the assignment which could be considered as clarifying the goals, criteria and expectations. [21]

Paiva et al. also consider the feedback types presented by Keuning et al. in their own review of automated assessment in computer science education [27]. In their study Paiva et al. deduce that in fact many of the modern automated solutions only extensively cover the third feedback type. Concretely the third type is told to include information about the test cases that the assessed code has failed, technical errors, solution related errors and issues related to quality aspects such as style and performance.

Paiva et al. mention that the knowledge about mistakes partly does tie in with knowledge about how to proceed. This fourth feedback type is also seen as rarity in today's automated assessment solutions, however, there are some advances regarding this. Some automated assessment tools can produce personalized feedback and offer guidance by recommending corrections to the tested source code to fix bugs or suggesting more optimal solutions even if the provided solution would be fundamentally correct. [27, pp. 1:15]

The first two types are traditionally not covered by automated assessment tools today in any meaningful way since they are seen more akin to matters of configuration or manual labour tasks by the instructors or exercise authors. It is also mentioned that the fifth feedback type, which aims to check whether the student understand why an answer is or is not correct, is also not commonly automated. In many cases the open-ended nature of solutions to most programming assignments

usually makes automating such feedback as knowledge about meta-cognition challenging. [27, pp. 1:15]

To reflect the previous theory and feedback types to DTEK2040 practices, it can be mentioned that the types are at least very applicable to the course contents as they are relevant in scope. While the assessment and feedback process of DTEK2040 is currently executed manually, the focus areas regarding feedback types are still noticeably the same: majority of the feedback provided to the students are knowledge about the mistakes, since the feedback is heavily reliant on using the assignment required functionalities as a basis; mistakes related to these implementations transform into feedback. Other types of feedback, such as knowledge how to proceed, seem to require activity from the student if this sort of feedback is wanted: if the student encounters bugs that impede progressing or finishing the assignment, feedback for these must be asked, for example, during voluntary workshop sessions arranged throughout the course.

3.4 Foundation for the second main research question

3.4.1 Q2.1: What manual work related to assessing and feedback is there to automate?

In the Section 3.1. three considerable primary components were brought up to answer sub-question Q2.1. These components are (1) the principals to use for assessing the suitability of a process from automation perspective, (2) the levels - or steps - to take when implementing RPA from ground up to a process and (3) the analysis of DTEK2040 assessment process.

The principals that largely determine the suitability of a process could concisely summarized into four points. These four points were also underlined to apply for automation in general; the principals could be followed when considering which business processes to automate with RPA or which test cases are most likely to provide the best return for investment. The four cornerstones mention that a potential candidate process should:

- have high transaction volume;
- be highly standardized;
- have well-defined implicit logic;
- be mature and preferably not dynamic in terms of future changes.

The second aspect to consider in implementation is to do it logically, following certain steps to build the automation system into a solution from ground-up. These steps are represented in the Figure 2:

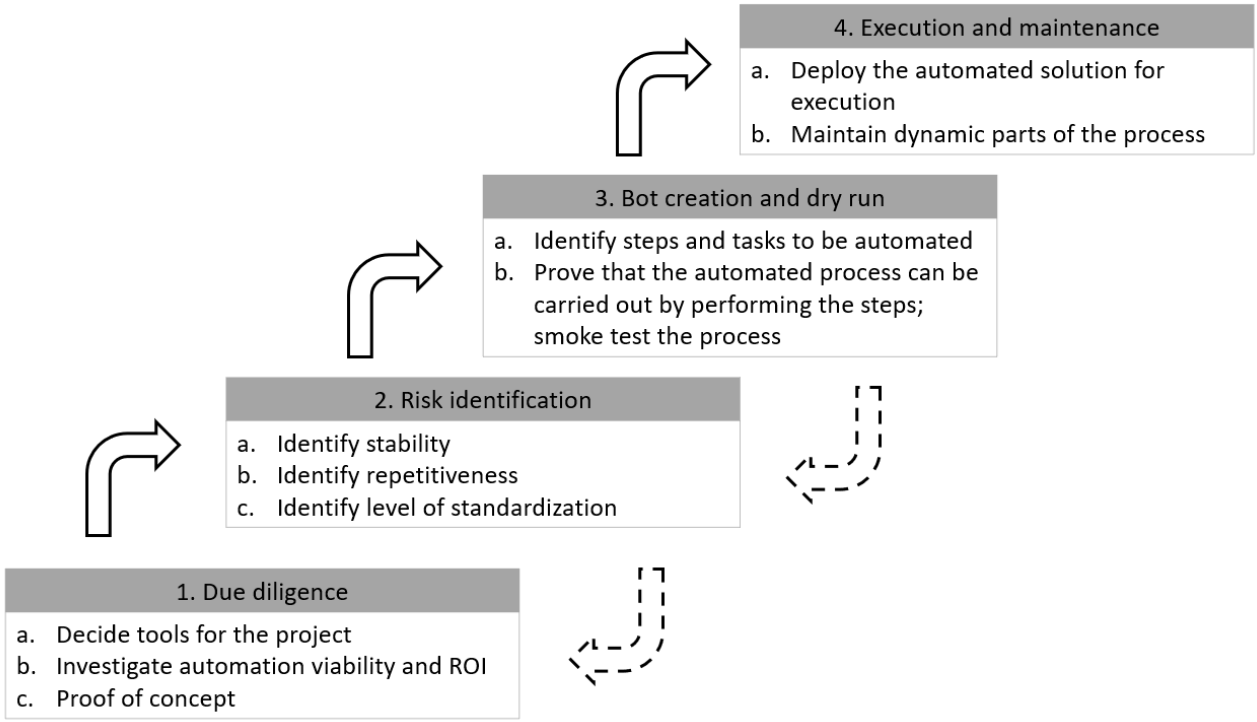


Figure 2: Suggested implementation steps to take for creating new automation systems.

Some targets within the assessment process of DTEK2040 were identified through analysis and presented in a Table 8 [pp. 31]. From RPA point-of-view, the identified targets included manual tasks to be executed on the Moodle platform for fetching student material and possibly uploading feedback and assessment results as well as file manipulation tasks related to handling student submissions in different formats and performing book-keeping of grades and feedback as the automated assessments are executed.

In the introduction of this thesis, it was also mentioned that the answer to this sub-question would be formed from the collective results of both the Section 3.1 and Section 3.2, the latter sub-section providing insight into use of automation in web application testing. Within Section 3.2 it was brought to attention that the principles of automating a process such as a manual test case are largely the same ones observed as general guiding principles for automation. It was also again confirmed that system level testing would perhaps be the best focus area for automated testing in terms of assessing web application programming assignments.

A more concrete suggestion extractable from the observations about automating web application testing was the division of test cases into three main test areas:

a) Content testing: focuses on the human-interface element, accuracy of the presented information and the features affecting user experience.

b) Architecture testing: focuses on testing for navigational and structural errors.

c) Environment testing: focuses on browser and system configuration effects to web application and its functionalities.

In the context of DTEK2040 the areas could perhaps be organized into order of importance followingly: 1) architecture testing, 2) content testing and 3) environment testing. From the expert interview and the course contents it would be fair to deduce that the assignment taskers themselves are highly focused on assessing functional aspects, which tend to be related to manipulating the application UI or state transfer functionalities of the application. It is also arguable the architecture testing contains tests targeting database interfaces and data validation.

Environment testing, on the other hand, is not seen as a focus-area in the context of DTEK2040; assignments do not explicitly detail environmental requirements such as the need to function on multiple browsers. The only environment related issues that would perhaps rise during the assignments and could be considered from test automation perspective are related to deploying some of the assignments to be executed on a cloud platform.

All in all, to answer the question *what manual work related to assessing and feedback is there to automate*, it can be said that the assessment process of DTEK2040 includes automation targets of opportunity for both RPA and software application tests. From RPA perspective the tasks have been analysed and presented to an extent include course workspace, file manipulation and book-keeping related tasks that seem to repeat with almost each individual student submission. While the tasks themselves may not be too time-consuming individually, it has been established by the expert interview that the cumulative benefits of automating such labour my end up saving a considerable number of manual workhours per instance of DTEK2040. The opportunities of automation testing were also determined to exist but mostly at the architecture and content testing realms for performing functional system testing. To support the assessment process these automated tests should target the verification and validation of assignment requirements.

While feedback will be considered in more detail with the next sub-question Q2.2, it can be mentioned that separation between traditional RPA and test automation can be made in terms of their benefit for collecting and providing feedback: while test automation can provide the basis for determining the content of feedback, RPA is able to support with the labour related to formulating and delivering that content to the student by, for example, gathering a concise summary of feedback based on test suite results and automating the process of reporting assessment results for the student to the course workspace.

3.4.2 Q2.2: What kind of feedback should be gathered from the student solutions to assignments?

The importance and the guiding principles of feedback were explored during Section 3.3. Within this section it was found that in general feedback should allow for student growth in self-regulation, clear confusions regarding the expected learning outcomes and performance details, as well as encourage and enhance the student's motivation to uphold effective learning process.

From the source material it was observable that the general guiding principles for effective, self-regulation enhancing feedback have plenty of common surface with the main feedback types currently seen to exist - and to some degree implemented - within today’s automated assessment and feedback solutions for programming exercises. Based on the review sources of automated feedback generation for programming exercises it was additionally notable that automation most often produces the feedback based on results of automated functional testing and the result is inclined to be more summative form of feedback rather than qualitative verbal commenting.

Modern automated assessment solutions have been noted to increasingly show interest in advancing the ability to provide “knowledge about how to proceed” type of feedback. The main challenge with other than functionalities targeting feedback for automation had traditionally been rooted in the open-endedness of programming solutions; it has been difficult to formulate personalized feedback for non-functional issues in an automated fashion since, for example, model-based evaluation can be hard to implement when there are no strict expectations for a detailed submission.

To answer the sub-question Q2.2, it is perhaps proper at first to introduce Table 9 [pp. 43] and attempt to present certain aspects to look for within an assignment submission. Afterwards the focus can be shifted on other details such as issues related to delivering the feedback properly.

Table 9: Issues to consider when gathering, constructing and providing feedback.

Points of interest from feedback perspective		
Issue to note from submission	**Rationale**	**Based on**
Lack of understanding the learning goals	Guide the student towards mastering the relevant concepts within the course's scope.	Principle #1, Knowledge about task constraints, Knowledge about concepts
Lack of understanding the evaluation criteria	Reduce the potential negative effects of mismatched expectations to student's motivation and receptiveness to feedback.	Principle #1, Knowledge about task constraints
Strong understanding of taught concepts	Acknowledging strengths is part of high-quality feedback. Enhances learning motivation.	Principle #3, Knowledge about concepts
Weak performance of taught concepts	Giving constructive feedback and corrective advice regarding observed weak performance provides the student learning opportunities.	Principle #3, Knowledge about concepts, Knowledge about how to proceed
Failures to implement a required feature	The assessment should be transparent in a sense that it provides the student exact knowledge about performed mistakes to focus on overcoming these issues and learn the concepts.	Knowledge about mistakes
Failures that prevent functional assessing or issues related to quality aspects	Some requirements may be implicit, such as the expectation that a submitted web application should be successfully deployable. Student should also receive feedback on technical errors and quality aspects of their products, i.e., coding style or solution style. These will help the student grow relevant skills even outside the course scope.	Knowledge about mistakes

As for the delivery of feedback, a few important observations were extractable from the research material: (1) effective feedback should favour comments over grades, (2) feedback process should happen in multiple cycles throughout the course instead of fewer high-stake formats, (3) high quality feedback should be provided even if the solution is correct, (4) providing the opportunity for the student to response to the given feedback increases engagement, decreases the chances of feedback being ignored due to misunderstandings and increases the positive impact and, (5) feedback should also act as data for potentially improving the learning process of the course as well.

While the sub-question *what kind of feedback should be gathered from the student solutions to assignments* could perhaps be presented an answer to with Table 9 [pp. 43], it is also worth noting that automation may be able to support feedback providing in many other ways than simply just assisting in gathering and noting feedback-worthy issues. During the exploration of related theory, it was quite implicitly mentioned also that automation is considered as a tool to possibly shorten the feedback loop: this in turn would mean more impactful – properly timed – feedback. Automation is also able to provide the opportunity for the student to self-regulate and independently subject his or her products for assessment, as it seems to be the case already with many modern programming courses today[2]. Some of these solutions of course go beyond simple automation, also requiring additional capabilities from the study platform itself.

[2] This can been observed to apply to courses where programming basics are studied with certain languages as well as intermediate courses that still contain, at least partly, assignments that are less freeform. This notion can also be considered from the perspective of how many fully or almost fully automated programming courses are being offered today as MOOCs or independently studyable online courses. Examples of such would include *Ohjelmointi C-kielellä* (https://fitech.io/en/studies/ohjelmointi-c-kielella/) offered by LUT university, Functional programming courses (https://fitech.io/en/studies/functional-programming-1/ and https://fitech.io/en/studies/functional-programming-2/) offered by TUNI and even the earlier mentioned web development course by Aalto university.

4 Combining test automation and RPA to assess assignments

4.1 Formulating a design

As it has already been established, Eggert's four stages of engineering design process [Table 2, pp. 4] guide the creation of an automated solution to be used for supporting the assessment efforts of DTEK2040 student submissions. Understanding of the problem – and aspects of potential solutions – has already been gathered by covering the relevant theory in Section 2 and Section 3. Based on the presented theory some constraints and considerations have been presented while formulating the answers to the sub-questions respective to their sections, but two very important aspects of Eggert's first stage remain yet to be established clearly: requirements and performance targets.

Requirements can be divided into various categories. However, while formulating the design for the proposed automated support system, three guiding categories have been identified through literature review: (1) general guidelines and automation targets, (2) guidelines for supporting feedback and (3) assessment guidelines for the exercises. Guiding principles that fit under the first two categories are gathered based on the domain theory presented throughout this thesis and the expert interview. Metrics for assessing the success of meeting these criteria are also somewhat qualitative and thus the assessment of the achieved end- product is also based on the opinions gathered from the DTEK2040 personnel.

Principles fitting the third category on the other hand are solely extractable from the course material and assignment tasks. While the scoring and the guidelines used for assessment are not always explicitly available from the material, anonymized student assignments and their results will be used as metrics for the quality of automated tests. The created test cases will also be iteratively developed by requesting feedback of the test cases and details to tasks presented in the course material from the course personnel.

4.1.1 General guidelines and automation targets

General automation guidelines for the prototype are crafted by combining the description of RPA from Section 3 together with the identified potential for automation within the DTEK2040 assessment process [Table 8, pp. 31; Appendix A]. However, not all potential will be implementable as dictated by the scope and delimitations of this thesis. For example, the solution will not emphasize automating the interfacing with Moodle, but it may present observations of the usability of such platform from the view of automation processes.

Based on the source material – and especially the expert interview – five general guidelines for the prototype system are identified:

1) System must save workhours;
2) System must be maintainable;
3) Human must be kept in the loop;
4) Automation flow begins after the submissions have been manually fetched;
5) Automation flow covers (a) assessment preparation, (b) numerical assessment, (c) feedback discovery and (d) summary reporting of assessment and feedback results.

The first guideline quite simply aims for time resource benefit stemming from organizing, testing and various other now manually performed tasks such as score and feedback book-keeping. The metrics used to determine whether this guideline is followed is quite simply the average time spent on each student and submission per exercise throughout the automation pipeline.

The second guideline is born from a couple of relevant points regarding the nature of DTEK2040 and programming courses in general: (1) As the course evolves and its exercise contents also possibly change as time goes on, the assessment support system needs to be modifiable so that the test cases can be updated to stay relevant, for example. (2) Intent at keeping the system as generalizable as possible has also been established in the beginning of this thesis for the system to potentially lend itself to other courses outside of DTEK2040 as well. Metric for measuring fulfilment of this guideline is admittedly somewhat subjective but has been determined to be documentation coverage: system architecture, main functionalities, all the test cases and any custom libraries should be documented so that the system may be adopted with relative ease.

The third guideline is also two-fold. The need for human in the loop has been identified due to accepting the fact that this sort of an automated system will most likely not be 100% flawless after its initial deployment and once fielded it will most likely stir new ideas for iterative improvements. However, since the system performed assessment will potentially have real impact on aspects such as an individual student's overall study performance, it will be sensible to not hand-over the whole assessment chain to automation without a human checking and either accepting or modifying the results. On the other hand, keeping the course personnel in the loop will also provide visibility to the inner workings of the system and stress the documenting of the reasoning and traceability

behind individual assessments as a part of the end summary. In this sense the third guideline will also be able to build trust towards the system.

The last two guidelines are quite simply guidelines for which steps from the current identifiable assessment process should be covered with automation. To meter the success of following these guidelines, the process presented within Appendix A will be used to determine successful fulfilment of steps 3 – 6 mentioned within that appendix.

Certain requirements conforming to these guidelines are identified from available source materials that are tied to metrics to gauge the success of each individual requirement. Metrics are also accompanied by target values that the system aims to fulfil to determine acceptable implementation of any given requirement. General requirements and their proposed metrics and target values are concisely presented in Table B1 [Appendix B] contained within thesis appendices.

4.1.2 Guidelines for supporting feedback

Guidelines for feedback support are mainly created from the basis of Section 3.3. Expert interview provided very little in the form of requirements for formulation of feedback but did, however, provide feedback models in the form of current operating procedures. The four guidelines identified for feedback support are:

1) System must be able to formulate numerical feedback and aid with qualitative feedback;
2) Feedback must be discovered from positive cases as well as negative cases;
3) Feedback discovered must be categorizable into one or more feedback types described by Keuning et al.;
4) Feedback discovered must follow one or more of the seven principles of feedback by Nicol and Macfarlane-Dick

The first guideline for feedback support is justified by the expert interview where it was mentioned that a half-automated solution that would, for example, annotate or tag qualitative feedback issues for further inspection by the course personnel would already be a meaningful step towards automating this part of the process. The second guideline closely ties into this by practically guiding the discovery of feedback points and test case designs to cover more than just the happy paths within the exercises.

Metrics for all the above guidelines are tied to the availability of both numerical feedback and tags for qualitative feedback issues marked for further inspection. Since the amount, tone, and practical discovery of feedback is respective to each individual student submission, hard targets for amounts or types of feedback to discover are in general challenging to determine. However, the following targets have been determined to provide some guidance and criteria to potentially consider when implementing automation regarding feedback:

- Final assessment summary contains a score for each exercise subtask. Summary will also always highlight qualitative notes from source or gather such points into a student specific document.
- Deduction in numerical score must be traceable to test case and reason included in a student specific document. Submission should always result in numerical and verbal feedback, even if submission is “perfect”.
- Implemented system logic for qualitative feedback tagging can be justified by referring to feedback types / principles and the justification is documented in the code, architecture, or instructional materials.

Discussion and analysis regarding how these guidelines could be implemented in an automated assessment system prototype will be presented later during Section 5 and Section 6.

4.1.3 Assessment guidelines for the exercises

The proposed assessment guidelines aim to detail what can be commonly assumed from subjects under test and provide common guidelines for the test basis and test case design. These requirements are formulated by reading into and analysing the taskers of all DTEK2040 exercises within the web materials [38]. Guidelines for assessing exercises are:

1) Each assignment is assessed as if it was the complete submission for the given assignment;
2) Non-functional aspects mentioned in assignment specific course material must be considered when forming test cases for assignment;
3) Automated assessment must provide visibility to the argument behind the numerical score and maintain traceability between individual parts of an assignment and the final scoring.

The first requirement is corresponding to the current manual processes. While the exercise material contains taskers for individual steps, these steps lead up to a complete solution that is then

submitted and eventually assessed. RF does provide means to run individual test cases from a suite by implementing run tags to each case and in this sense it would be possible to create a system that supports also assessment of exercise partials. The metric for this guideline is the step coverage of an exercise with test cases. Target for coverage is 100%, meaning each individual functionality or a separate task within an exercise must be covered with a test case.

Detailing the metrics and targets for the second guideline is the most challenging of the three. Non-functionalities themselves are extractable from the exercise specific web materials but hard targets – such as testing for the use of session storage when creating a single page application, for example – will be defined within the test suites themselves in the form of test cases and corresponding documentation.

The final guideline is partly overlapping with other guidelines from previous subsections, Section 4.1. One such example is keeping the human in the loop or the requirements related to feedback discovery. A soft metric for this assessment guideline is the amount of traceable test case results and the target for traceability is that each test case and its result should be tied to (a) exercise, (b) task within exercise and (c) in the case of point deduction / remark the reason for said action. Besides the soft metric, more important is to hold a human in the gatekeeping role for any results produced by the prototype at this stage of development. Thus, traceability needs to be available for the course personnel and – if requested – possibly for the student who did the submission.

4.2 Generating a design

In Section 3 the suggested implementation levels to consider when creating new automation systems were mentioned to be (1) due diligence, (2) risk identification, (3) bot creation and dry run, and finally (4) execution and maintenance. Detailing the general contents for the first three levels here is an attempt to complete Eggert's second stage of engineering design process and generate a design that also considers the requirements set forth during the previous Section 4.1. The fourth level dealing with execution and maintenance will be omitted as the contents within will be later presented in the form of system documentation and execution related instructions corresponding to the actual implementation.

When gathering the information on stages for Eggert's proposed implementation of engineering design process it was mentioned that this specific step should have a goal of producing alternative design candidates for later analysis and evaluation [Table 2, pp. 4]. However, within the context of this study only one proposed design will be generated in this Section. Potential alternative

implementations and design choices will be part of the critical discussion presented in the final Section 6 of this thesis.

4.2.1 Due diligence

Due diligence for the design has been largely performed by delimiting the technology choices within Section 1.3, analysing the flow of the current manual assessment process and then analysing automation viability of the process [Table 8, pp. 31; Appendix A]. Based on the sequential implementation levels of automation [Table 7, pp. 30], due diligence should on top of these also include determining the return on investment and a technical feasibility assessment with a proof-of-concept. Determining the return on investment nor the performing of a technical feasibility study are not included in this sub-section but rather covered by the descriptions and results reviewed within Section 5.

As for the tools to be used to create an automated solution for assessment support, one major choice that was established early on in this thesis: RF for test automation and RPA purposes. The framework is by its maintainers described to be Python-based and keyword-driven automation framework for acceptance testing and robotic process automation (RPA). RF at its core does the work of running the tests and tasks that are written but interactions with the target under automation are handled by functionalities from various imported libraries. The representation of RF architecture as detailed by Robot Framework foundation is shown in Figure 3. [32]

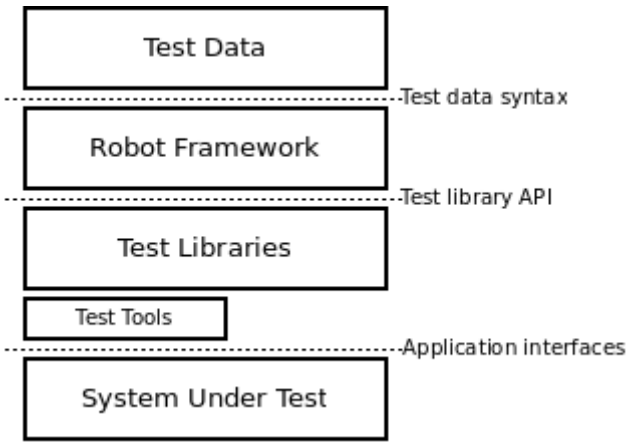


Figure 3: Robot Framework architecture by Robot Framework Foundation. Source: [32].

At a high-level the acceptance and keyword-driven nature of RF allows for clear, natural-language like presentation of automated tests and RPA tasks while still allowing for precisely coded

functionalities to be written and executed behind the said keywords. Also, because the framework is an open-source solution it is well maintained, transparent and has plenty of available libraries created by the community for a multitude of use cases. From generalization perspective, RF offers a low-code automation framework that is relatively easy to learn and can be widely applied to static and dynamic manipulation of different application types from traditional desktop applications to mobile and web applications. The framework is also compatible with all major operating systems and supports integration with Java and .NET based platforms by providing support to Jython and IronPython respectively. [32]

Maintainability is an aspect that could perhaps be described to cover concepts such as modularity, understandability, reusability and refactorability. Considering the framework from this perspective, the open-source tool may provide both opportunities and challenges. As an open-source framework, the framework does provide plenty of visibility and opportunity for the afore mentioned concepts. Keyword oriented nature also allows to easily understand and follow the process flows even without diving too deeply into the source codes or imported library methods if the solutions are coded with maintainability in mind. However, depending on the number of libraries used or custom libraries written, the amount of content to study to effectively maintain the created solution may quickly turn out to be quite excessive if libraries are incorporated haphazardly. From this perspective each used library within the created automation solution will have to go through an evaluation whether it is truly worth using or is it just an extra burden in terms of maintainability.

Aside from the framework another technological choice is to support the use of Docker[3], allowing the automation can be executed in an isolated environment, thus reducing the need for setting up dedicated workstation environments for running the automation process. Containerization will also produce additional security benefits as student submissions can be dynamically manipulated in an isolated container instance, which potentially reduces the effects of malicious intent if there would happen to be any, for example, infested URLs or files within a submission package.

Within this due diligence process it is noted that the current or possibly future course personnel of DTEK2040 do not necessarily have prior deep experience of the framework, Python or Docker. It is, however, reasonable to expect that the personnel have a grasp of programming - or perhaps even are experienced programmers - and that they are technically inclined in general. As such, it is

[3] Docker is a service that packages an application into a virtual container so it can be executed on various operating systems of choice. It is also able to offer flexibility in terms of where an application could be hosted; containers can be hosted on-premises or in cloud. Docker containers are technically hosted by software that is called Docker Engine, but the service and the underlying engine are, depending on use-purpose, available for free from https://www.docker.com/.

accepted that maintaining the created automation solution will require some familiarization with the applied technologies to maintain the solution.

4.2.2 Risk identification

The risk identification level of RPA implementation was determined to consist of identifying three aspects from business processes to determine the overall suitability for automation: (1) stability, (2) repetitiveness and (3) level of organization or standardization. To an extent this risk identification has been done throughout this thesis and condensed in the form of tables [Table 7, pp 30; Table 8, pp. 31].

The current process of manually assessing student submissions has been deemed stable, as in largely unchanging and routine-like, based on the conducted expert interview. As such the steps mentioned in general requirements GR-5 and GR-6 [Table B1, B-1] can be viewed as good candidates for automation. It is recognized that outside of these steps there may be potential for less stability as student study platforms or administrative tools can change or evolve, which may or may not cause changes in the future. The automation solution will attempt to keep this aspect in mind by assuming as stable of a starting and ending context for the requirement specified steps.

Repetitiveness is something that the solution views as an inbuilt quality within the context of assessing student works in a programming course: for mandatory courses even the basic business process needs to be repeated hundreds of times to assess each individual submission. Of course, additional repetitiveness is created from the number of exercises in the DTEK2040 course curriculum which effectively creates its own multiplier when each student will be submitting multiple assignments within the span of the course. Based on the interview any given exercise may receive 100 – 120 student submissions each: in Spring 2022 for example this meant an estimated total of 500 - 600[4] student submission to assess and provide feedback on throughout the span of the course.

The level of organization of the business process is perhaps the most controversial aspect of this level from the three. The very basic business process of receiving a submission, assessing it, providing feedback, and registering a score is notably very standardized. From a test automation

[4] DTEK2040 has assignments for five different study parts: Exercise 0, Exercise 1, Exercise 2, Exercise 3 and Exercise 4. Student numbers are based on estimates extracted from the expert interview and the total submissions number is assuming each student would return a complete set of assignments for each exercise. In reality the submission numbers might even be higher due to the course allowing for submissions of separate sub-tasks as well before the final complete submission.

perspective, however, the exercises themselves are in a way very softly standardized. The course material is crafted so that it guides the student towards certain kinds of solutions which for most of the so-called perfect submissions means that they are expected to be quite identical regardless of the student. This in turn means that test cases created based on these submissions will be able to test and assess the majority. On the other hand, the soft guidance means that there is also potential for alternative ways for the student to reach an acceptable solution and in these cases the student should not be punished for doing so. Such an alternative way can from testing perspective be seen as an edge case of sorts which is not necessarily though of when developing test cases that assess a given exercise.

All in all, the current process is seen as a good candidate for automation, but it is also identified to contain risks in terms of test automation coverage and standardization within task descriptions. Thus, it is accepted that using the most recent anonymized student submissions material to craft test automation cases can still leave room for development in the future if edge cases from student submission in future course iterations are found. To further mitigate this risk, the automated solution will have to create sufficient documentation of each assessment process to provide the system maintainers enough information to act upon. This will also be considered when implementing general requirements GR-2 and GR-3 [Appendix B].

4.2.3 Bot creation and dry run

The overall process for assessment has been identified from expert interview and the steps represented within Appendix A. The steps and tasks to automate have been further dictated in general requirements for the automated system by the requirements GR-4 and GR-5. To break these steps down further, the overall robot flow is represented in Table 10.

Table 10: Steps and tasks outline for robot implementation.

Prototype development steps and tasks	
Step	**Tasks**
1) Process starts from a state where the system is provided with packages of each individual student submission for an exercise	a) Verify directory structure b) Verify content is provided
2) Robot will prepare the submitted files for general assessment	a) Create student specific directories to track submissions b) Extract submission package
3) Create summary template	a) Create summary sheet template to keep track of scores and feedback issues

(to be continued)

Table 10 (continues)

4) Robot will perform the static assessment	a) Search for static page files / web app source files b) Execute test cases that assert from source code c) Update summary sheet as required based on test results d) Save test log into student directory
5) Robot will prepare the submission for dynamic assessment	a) Open a static page in a browser instance or run web app and open it into a browser instance
6) Robot will perform dynamic assessment	a) Execute test cases that assert from static page / web app b) Update summary sheet as required based on test results c) Save test log into student directory
7) Robot will create a summary containing test results, numerical score, and feedback issues	a) Calculate final score of the exercise for each student submission b) Collect additional feedback from student specific test logs and attach content to summary c) Collect and organize summary, individual logs and other artifacts such as screenshots into a review directory
8) Robot will output the summary files and artifacts for human review	a) Output the review directory

The step tasks consisting of scripts and test cases will be analysed in detail within Section 5 based on the actual implementation.

The smoke testing – or dry running – of the robot will be performed in a two-tiered fashion: Firstly, the overall pipeline consisting of scripts that handle general tasks such as executing the RF files will be ran with a goal of determining that each required Robot file can be found and the environment dependencies are installed as required. Secondly, during the pipeline dry run the Robot files are executed with RF *--dryrun* option which causes library keywords to not execute but verifies that everything is syntactically correct and all imports within the Robot files can be resolved.

5 Implementation and results

5.1 Architecture overview

Solution applies the pipe-and-filter architecture pattern which is a popular choice, for example, among workflow engines and scientific computation systems that need to process large streams of data. The pattern in general is based on building separate parts that process input from downstream to produce output to upstream for then to be used by the other parts present in the system. These filters often have very specified tasks which also adds to the ease of reasoning about the overall behaviour of the system, which in turn can be seen as a positive side in terms of learnability and maintainability. Architecturally the pattern is also said to support aspects such as reusability, flexibility scaling and parallelization. [5] Applying this pattern to the prototype is represented by Figure 4.

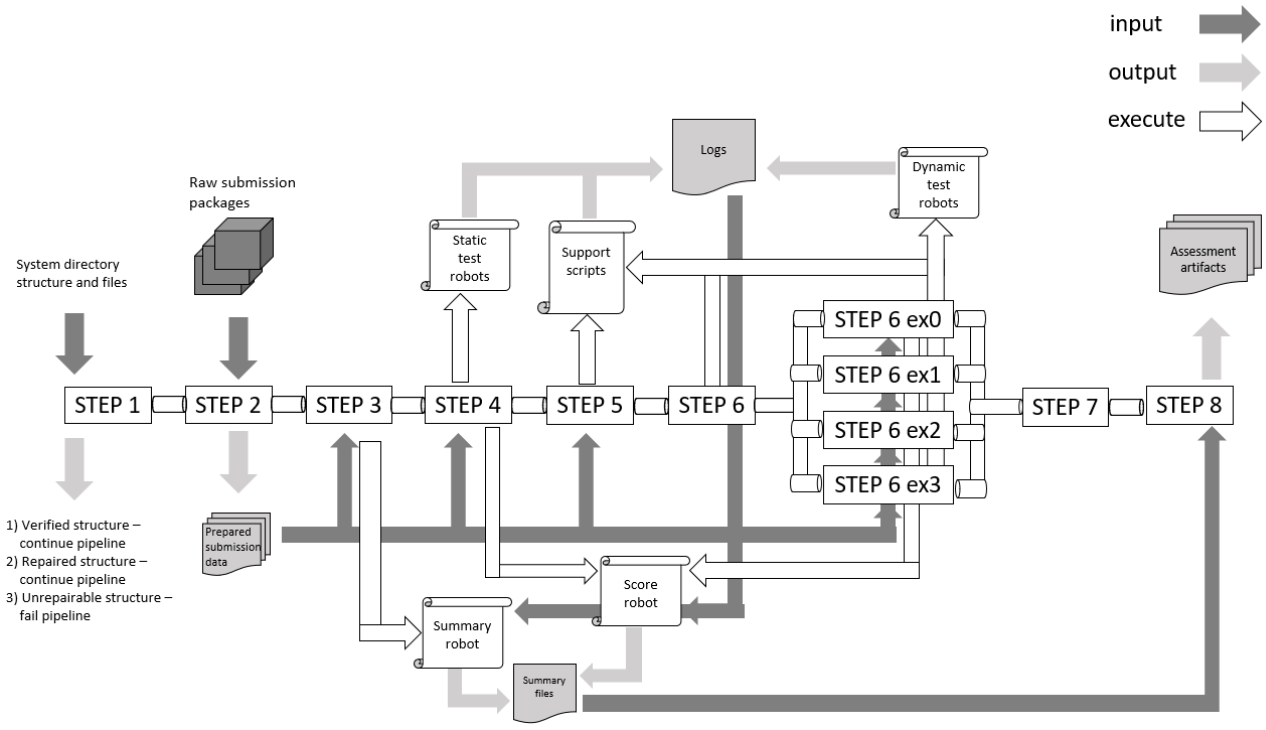


Figure 4: Pipe-and-filters implemented in prototype solution.

The pipeline has been structured from eight main steps – or filters – that contain separate tasks extracted from the current manual working process of assessing student submissions of DTEK2040. These steps handle and manipulate different data as well as execute automation scripts to produce the final output in the form of assessment artifacts: the numerical results gathered by running

submitted material against automated test cases as well as general feedback that has been formed by different tools and methods.

5.2 Environment

5.2.1 Development environment

To lay a baseline for the results to be presented later in the analysis section, it is necessary to mention specifications of the workstation that was used for development of the solution as well as gathering of result data. These specifications are displayed in Table 11.

Table 11: Development workstation specifications.

Development workstation specifications			
Component	**Manufacturer**	**Model**	**Additional information**
Motherboard	MSI	MSI Z170A GAMING M7	MS-7976
CPU	Intel	i5-6600K	CPU @ 3.50GHz (4 CPUs), ~3.5GHz
Memory	Micron Technology	DDR4 SDRAM	Clock rate 1200MHz; 4 * 4GB modules, 16GB in total
GPU	NVIDIA	GeForce GTX 1070	Graphics clock 1506MHz Processor clock 1683MHz Memory 8GB GDDR5

From performance point of view the GPU is not deemed as a crucial component for the overall results in terms of such results as execution time. In general, even those automation tests that dynamically test through GUI of a web application are not heavily GPU dependant since tests ideally are run in a headless mode. Instead, most important specifications are assumed to be processing power (CPU) and memory.

The workstation used Windows 10 Home 64-bit (10.0, Build 19043) as its operating system, but the development and test run executions were done on Windows Subsystem for Linux. While no recorded comparisons were made in the scope of this study between the prototype performance when executed on Windows versus when executed on a WSL2 Linux distro, it is worth mentioning that the WSL2 seemed to give significant performance boost in terms of execution time.

5.2.2 Software

The prototype was developed and tested on Debian GNU/Linux 11 Bullseye. Two aspects contributed to choosing this Linux distro for development OS. Firstly, it was the most readily available Debian distro for WSL2. Secondly, it allowed for debugging 3-slim-bullseye[5] based Docker image. 3-slim-bullseye as the base image was chosen due to it providing the required Python versions and a good support for GUI based automation testing while keeping the Docker container size relatively small. Python included with 3-slim-bullseye at the time of development was Python version 3.10.7.

The 3-slim-bullseye base image did not appear to contain absolutely everything that was required for development, a multitude of additional packages needed to be installed. These additional packages were mainly tied to RF libraries and their dependencies while some were simply tools that were required to perform certain bash scripted automated tasks. Packages mentioned in Table 12 were deemed necessary for the implemented prototype to function.

Table 12: Packages installed on top of base image.

Required additional packages by the prototype during development			
Name	**Apt(*) / pip(**) / npm(***) package name**	**Installed version**	**Type / additional info**
Node.js	nodejs (*)	18.7.0	JavaScript runtime environment
Npm	Installed along nodejs	8.18.0	Software registry, package manager and installer
create-react-app	create-react-app (***)	5.0.1	Package for creating react apps
json-server	json-server (***)	0.17.0	Node module for REST JSON services
MongoDB	mongodb-org (*)	6.0.1	Document database
netcat	netcat (*)	1.217-3	TCP / UDP utility
lxml	lxml (**)	4.9.1	Beautiful Soup dependancy
html5lib	html5lib (**)	1.1	Beautiful Soup dependancy
Beautiful Soup	beautifulsoup4 (**)	4.11.1	Python library; web scraping
Robot Framework	robotframework (**)	5.0.1	Python based automation framework
Browser library	robotframework-browser (**)	14.1.0	Robot Framework library; Playwright based browser / web GUI automation
Requests library	robotframework-requests (**)	0.9.3	Robot Framework library; HTTP api testing
Excel library	robotframework-excellib (**)	0.0.2	Robot Framework library; Excel file manipulation

[5] https://github.com/docker-library/python/blob/56d9977bf9a2e92882e71256dd288c8482233688/3.10/slim-bullseye/Dockerfile

From packages mentioned in Table 12 [pp. 58], Netcat is mostly utilised to check Node backend services and Mongo database availability as required and pace the pipeline execution. Node.js and npm are also fundamentally required due to DTEK2040 relying in these tools; they are a natural choice to use in the dynamic testing of submitted assignments. Same logic applies to the decision of integrating create-react-app, json-server packages and the use of Mongo database.

Beautiful Soup as a library specialises in web scraping and is partly relied upon in static testing. The library allows for pulling data from HTML content in a relatively straightforward manner by parsing source code into a parse tree. Lxml and html5lib are parsers that can be used by Beautiful Soup and they were chosen because they seemed to be the most popular choices based on supporting documentation and resources available.

RF has a dual purpose in both automation utility and test automation. Excel library is used for supportive tasks related to record keeping and formulation of final score summary files, but the other two framework libraries are at the core of providing tools for dynamic testing. Browser library allows for interacting with the apps through GUI by providing Playwright engine[6] based support for rendering engines and a ready set of keywords to be used after establishing a browser session. Requests library on the other hand provides a set of keywords for http requests which in turn allows for API testing alongside the GUI testing of web application functionalities.

All in all, the final prototype version can be taken into use in any of the three ways: (1) by creating a Docker container using the included Dockerfile; (2) on a Windows workstation that supports WSL2 by installing Bullseye Debian distro and the required packages; (3) by installing and executing on a Linux workstation, preferably one with the same distro as the one used for the Docker image for this solution.

5.3 Pipeline

5.3.1 Logical structure and shell scripts

The automation implementation consists of 17 separate shell scripts that are written in bash. Each script has their own role inside the architectural representation shown earlier in Figure 4 (pp. 55).

[6] Playwright is an open-source web testing and automation framework available at https://github.com/microsoft/playwright. It supports Chromium, Firefox and WebKit which means it is able to be used for automation testing web applications on all of the most popular browsers.

The scripts creating the overall system are divided into three separate logical layers, which aims to enhance the learnability and maintainability of the system. Separation is represented in Figure 5.

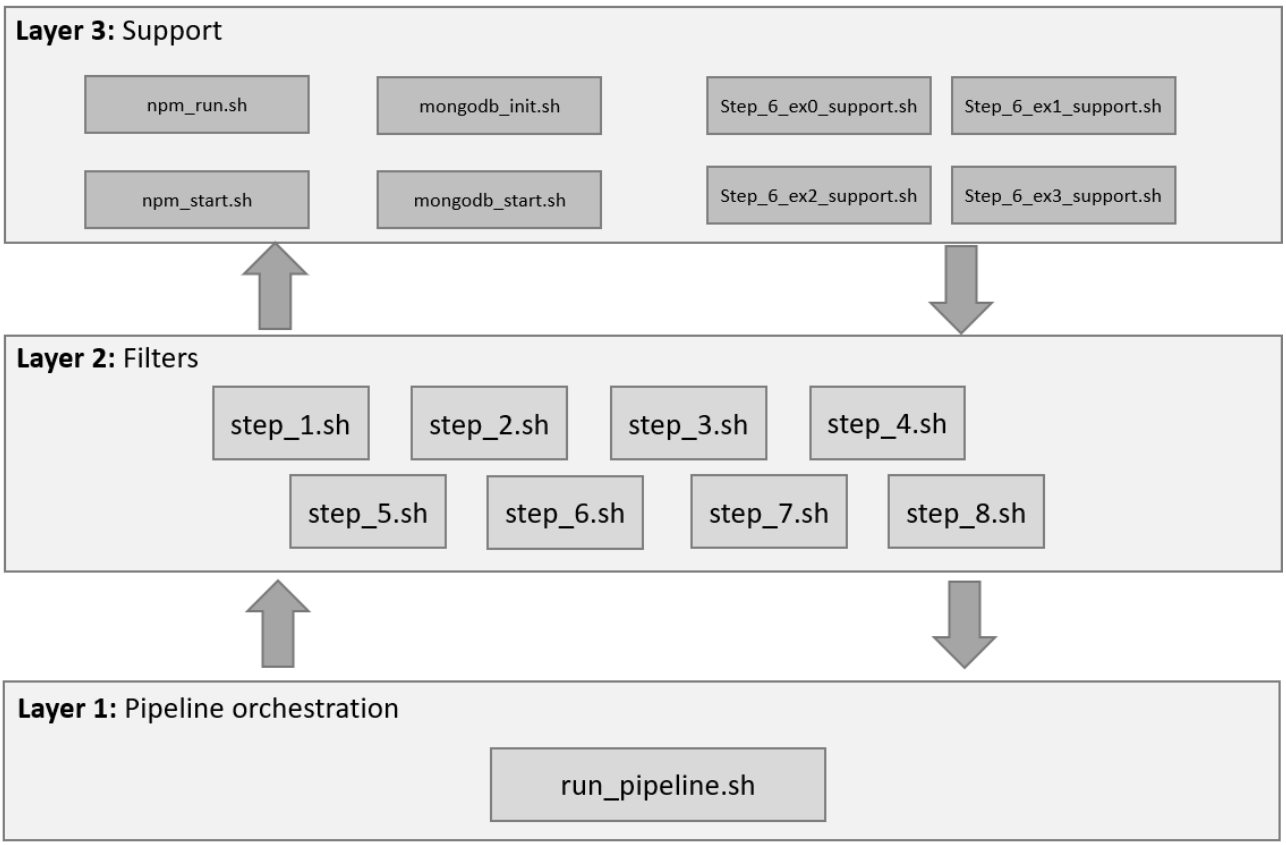


Figure 5: Shell script logical layers.

The layers as represented in Figure 5 above are:

1) Pipeline orchestration: Responsible for maintaining global variables, checking the execution status of each main step as the pipeline is running and maintaining situational awareness of the overall pipeline execution.

2) Filters: Contains main steps of the pipeline, each with their own processing responsibilities.

3) Support: Scripts that execute certain processes, such as database manipulation or robot scripts, by the request of main step scripts.

The logical layering is designed and implemented so that scripts belonging to a certain layer should only execute and provide input to other scripts belonging to the layer above but only provide output results to those below. This aims to provide easy-to-grasp concept for interactions between separate parts of the system.

In practice the first layer was implemented so that one bash script, the *run_pipeline.sh*, handles the triggering of all the primary step scripts. The decision to continue or terminate the pipeline is made based on the exit status received from each step; exit status of zero allows the pipeline to go forward while non-zero status will cause the pipeline to terminate. The main orchestrating script is also responsible for declaring and inserting values to global variables that it exports to be used by the step scripts. Examples of these variables are certain directory paths, tags to decide which tests should be executed as well as bookkeeping values such as exercise specific maximum score values.

The second layer houses all the main steps of the pipeline and they are, as mentioned above, triggered in order by the orchestrating first layer script. Each step and their corresponding step script has its own process responsibility to complete, and they execute third layer support scripts as needed to complete the tasks demanded by their processes.

In the first step the *step_1.sh* script is built to verify required files, directory structure and that any assessment data zip-packages are available to be processed for assessment purposes. The main reason for verifying the prototype system directory structure is to make sure the critical custom libraries, common resources or directories required by the applied tools – such as Mongo database – will not be a source of failure in case the pipeline continues executing. Some directories the script itself can create in case they are not found during the verification process but others will cause termination of the pipeline since they are expected to hold commonly used resources for the automation to function properly. For example, if in step 1 the directory containing automation test scripts is determined to not exist, it will cause the pipeline to terminate. The system files and directory structure are depicted in Appendix C. The appendix also described actions taken by the script depending on each file and folder state at the time of verification.

If file and directory verification is successful in step one, the step exits with status zero and pipeline can proceed to the second step. The second step with its *step_2.sh* script is responsible for preparing the student submission contents. In practice this means extracting the zip package contents to a corresponding test subjects directory to be available from that path for later process steps.

After extracting the submission contents to their respective test subject directories, the script finally verifies that the number of test subject directories is corresponding to the number of zip packages originally available inside the submission data directory. Overall successful completion leads to triggering the third step where *step_3.sh* executes the first robot scripts inside the pipeline: *summary_template.robot* and *update_score.robot*. The accomplishment of these robots fulfils the responsibility of creating initial summary template excel sheet with exercise specific test

identifications and student submission identifications as seen in Appendix E which depicts the template created for DTEK2040 Part 0 exercises.

Fourth step inside the pipeline performs duties related to static testing of submissions. In practice *step_4.sh* first executes the static testing robot corresponding to the exercise currently being assessed and afterwards updates the summary template with score results for the student. This two-part process is repeated for each student submission that needs to be assessed within the current pipeline run.

After static testing, fifth and sixth steps of the pipeline are both related to dynamic testing of the submissions. Firstly, *step_5.sh* has the role of handling preparations required for dynamic testing. For the prototype developed during this study, the step handles tasks such as initial creation of system-under-test react application for exercises one, two and three as well as the setting up of Mongo database for exercise three.

Once the services set up by step five are verifiably responding, then step six will use support scripts respective to each exercise that trigger the dynamic testing and score updating robots for each student submission. Support scripts are heavily utilized during *step_6.sh* execution because this allows for exercise specific variable inputs for the corresponding dynamic testing robots; for example, the DTEK2040 first exercise dealing with html tasks has no need for react server or Mongo database related variables to synchronizing the test efforts with backend services whereas the third exercise will require both. Thus, the use of support scripts here allows for not declaring irrelevant variables for robots executing dynamic tests.

Finally, steps seven and eight have responsibilities related to wrapping up the pipeline execution and whatever artifacts have been created during the testing. During seventh step *step_7.sh* performs the calculation of total score for each student. This is done by using a pre-determined max score for the exercise being assessed and then triggering the *update_score.robot* to update the assessment summary template with student's total score achieved over all the static and dynamic tests. Once step seven is complete, *step_8.sh* will perform pipeline teardown activities by killing active react and Mongo processes and perform tasks related to collecting artifacts as necessary.

The supporting scripts contained within the logical boundary of the third layer were already slightly described in conjugation with step six. Aside from the *step_6_ex#_support.sh* scripts mentioned already, the other support scripts are very limited and specific in nature. These scripts have been

devoted to certain tool use, namely npm and Mongo, and they trigger actions such as starting a react app or connecting to Mongo test database and then waiting for these services to answer.

5.3.2 Robot scripts

RF scripts are files ending with *.robot* or *.txt* extensions that are constructed from four sections: (1) settings, (2) variables, (3) test cases or tasks and (4) keywords as shown in Code 1.

Code 1: Example Robot Framework script contents.

```
*** Settings ***
Library     String
Library     .${/}MyLibrary.py
Variables   .${/}common_variables.py
Resource    .${/}common_keywords.resource

*** Variables ***
${HELLO}  Hello World

*** Test Cases ***
First Example Test Case
  [Tags]  tc1  full
  Log Hello World From Test Case  1

Second Example Test Case
  [Tags]  tc2  full
  Log Hello World From Test Case  2

*** Keywords ***
Log Hello World From Test Case
  [Arguments]  ${test_case_number}
  Log  ${HELLO} from test case ${test_case_number}
```

Settings section is used to import external libraries and resources such as custom keyword sources or collections of custom determined variables detailed in an external file. Settings section also allows for declaring suite setup and teardown commands if needed. The robot scripts developed for the prototype take advantage of in the whole scope mentioned earlier and the robot files containing test cases perform setup and teardown processes as well as to avoid cases failing due to environment, or other than system-under-test related, reasons.

While scripts can import variable collections from external sources, there is also the option to declare variables within the variables section. In both cases the variables declared will be treated as global variables. RF also supports a third way of providing global variables for the script by

declaring them through a command line interface when executing a robot script. An example of providing a global variable this way can be seen in Code 2 where the variables are declared with *-v* option. Alternatively, *--variable* can be used.

Code 2: Example of executing the support_tasks.robot script with declared global variables.

```
robot \
  -i "$et" \
  -d $RESULTS_DIR/"$ASSESSMENT_EX"/step_6/ex2_support_step/"$(basename "$sut")"/ \
  -v STUDENT_ID:"$(basename "$sut")" \
  -v EX_NUM:"$ASSESSMENT_EX" \
  -v DYNA_DIR:"$REACT_PROJ_DIR" \
  -v GLOBAL_ROBO_VARIABLES_DIR:"$GLOBAL_ROBO_VARIABLES_DIR" \
  -v RESOURCES_DIR:"$RESOURCES_DIR" \
  -v REPORTS_DIR:"$REPORTS_DIR" \
  -v TEST_SUBJECT_DIR:"$sut" \
  -v LIBRARIES_DIR:"$LIBRARIES_DIR" \
  $TASKS_DIR/support_tasks.robot
```

The depicted way of providing variables is also very much utilized by the prototype as it proves to be a simple way to provide, for example, directory paths and tags for the robot scripts to use. Whichever way is used to declare a variable, it will in all the aforementioned cases be treated as a global variable and thus they are usable everywhere within the script.

The test cases section contains the very core of any robot script. Test cases section can also be called the tasks section since this is only a semantical way of differing the purpose of a script in RF; if the script is more RPA than test automation in nature, the section is often titled “*** Tasks ***” instead of “*** Test Cases ***” but it will function all the same. For the purposes of describing robot scripts henceforth test cases and tasks can in this context be considered synonymous.

In the test case section, a set of keywords can be declared under any given test case which will then be executed in order of appearance. Syntactically the test case name is declared first and the keywords that form the test logic are indented with a minimum of two spaces below it, as seen in Code 1 [pp. 62]. Test cases can be assigned multiple tags, which quite simply allows for creating collections. When the robot script is executed from command line it may be declared a tag option value and only tests corresponding to that tag will then be executed. Again, to take advantage of Code 1 as an example, one could perform a command *robot -i full script-one.robot* to execute both test cases or alternatively use *-i* with value *tc1* or *tc2* to only run one or the other. Tags are also relied on in the prototype to create collections on both static and dynamic tests for any given

exercise set. This is because some of the DTEK2040 exercises are divided between two different web applications in which case they must also be tested with separate sets of tests.

Keywords used in test cases can come from multiple sources. They may be keywords from basic libraries that are included with RF installation, keywords from imported libraries, keywords from imported external keyword resources or keywords that are declared within the script itself inside the keywords section. While keywords have the potential to make the script contents easier to follow, they can also be a source of confusion from maintainability perspective. Initially it is very hard to know where an externally declared keyword is coming from unless you have prior knowledge of the imported libraries.

The prototype relies on 11 separate robot scripts which are divided categorically into automation tests and RPA tasks. Eight of these scripts are test automation in nature while three serve the purposes of RPA. Robot scripts are utilized by both layer two and three pipeline scripts and as such they are not additionally categorizable with the same logic as previously described pipeline shell scripts. Robot scripts that are responsible for test automation are organized into static and dynamic test robots and one of each exists for every exercise set of DTEK2040 as per the scope of this study. The static testing robot scripts, named as *ex#-static-tests.robot* respective of each exercise set, contain the test cases which only interact with the submitted source codes to assert certain expectations of the source code content. For example, one of the assignment tasks given to students in exercise set 0 that deals with html basics is to include a table within their static web page. Such a task is tested in a static manner by *ex0-static-tests.robot* in practice with two atomic test cases: first parsing all table elements from submitted raw source code and then (1) verifying that at least one table element exists and (2) verifying at least one whole table element is built according to the hierarchy rules of permitted content for such an element[7]. This sort of atomic division of tests in general allows for partially scoring tasks such as the one mentioned in the example, which in turn allows rewarding the student for partial successes. In principle the same practice is followed in static testing of all the web programming exercises of DTEK2040.

The dynamic testing robots, named in the prototype following a convention of *ex#-dynamic-tests.robot*, interface with the submissions through either graphical user interface on an established browser session or through API requests. Interacting through graphical user interface is a stable method among all the exercises in terms of dynamic testing methods. Interaction is for the large part

[7] Rules for allowed elements and their hierarchical relations have been gathered from https://developer.mozilla.org/en-US/docs/Web/HTML/Element/table (16/10/2022).

done by utilizing the Browser library available for RF since it offers practically all the required functionalities out-of-the-box with its available keywords. API testing serve more of a purpose for the latter two exercise sets of DTEK2040 where tasks and requirements for details such as API paths and requests are extractable from tasks and course material. It should also be mentioned that while RF offers the core functionalities for dynamic testing, to meaningfully execute these robots require the pipeline to setup services such as React servers and / or Mongo databases locally to create the backends and synchronize their availability in relation to test execution. This part is handled by the layer two and three shell scripts as described earlier.

Aside from the eight test automation robots, the prototype also contains three more robots that serve roles more devoted to RPA tasks, as was briefly mentioned earlier. These robots are *summary_template.robot*, *update_score.robot* and *support_tasks.robot*. The first one, as the name suggests, is responsible for initially creating and preparing a master summary template for each pipeline execution run. Tasks include removing the old exercise specific template, if any is available, and creating a fresh new one to be filled with submission ids of those students who are part of the current run.

Record keeping on the template is handled by *update_score.robot* once the template has been created. The tasks it performs are divided logically and by script tags to pre-filling submission names into a prepared template, score keeping for identifications already existing in the template and calculating the total score from the current state of the template. Observing chronologically, the first task of filling a freshly created template with the ids of those submission included in the current pipeline execution is done so that for this purpose the robot uses the names of the submission zip packages, i.e. the name of the package will be used in the template and any test results will be tied to it. Once the ids have been filled into the template, this robot is responsible for updating the score state after every execution of a static or dynamic test set. It does so by parsing and extracting test results from the result logs of the test robots and marking these into the template for score keeping purposes. Once all the testing during the pipeline is done, the robot calculates a total score for each submission based on the amount of passed tests and exercise specific max score value that can be declared as a global variable by the pipeline user.

The third RPA robot, *support_tasks.robot*, performs preparative tasks to submissions contents as they come up for static or dynamic testing. These responsibilities include tasks such as copying test automation resources and submission content into system-under-test project directory before dynamic testing so that submission content can be used to setup and execute React backend to test

the subject against. For the prototype this robot handles certain tasks to harmonize certain parts of submission source code. For example, to test database interactions in DTEK2040 exercise number three, this robot replaces the original Mongo server connection strings with a test database connection string to use the one created during pipeline execution and test each submission against an equal database.

5.4 Test cases

Test cases implemented in the prototype were extracted from the programming exercise descriptions of DTEK2040. In practice the requirements engineering part of automated test cases were done quite lightly, relying only on the understanding of the researcher. Even though some specifics were additionally discussed with the course personnel, requirements engineering was not a focus-point in this study and as such no iterative methods of refining test cases was done, for example. However, 43 individual test cases were created and documented as test case cards with individual identification designations that would tie the case to a specific exercise set, such as the one example shown in Figure 6 [pp. 67].

(1) Test Scenario ID	E0: DTEK2040 Exercise 0	(2) Test Case ID	T1-6: Page contains a form with input elements
(3) Test Case Description	Page has at least one form element containing input elements. Action attribute need not be defined.	(4) Test Priority	Adds to score
(5) Pre-Requisite	E0_T1-1	(6) Post-Requisite	If not the final test of the html test suite, the browser session and the index page will be left open. If the final test, close browser session

TEST EXECUTION

Step	Action	Inputs	Expected output	Test comments
(7) 1	Create a list for <form> elements from the open page	DOM	Amount of elements found and added to the lists is greater than 0	Can be tested from html parse as there is no need for retention of the original form
2	Choose the first available form from the list	list of <form> elements	A form element is available	
3	Collect input elements from the form	<form> from step 2	Element has more than one input element available	
4	Verify input elements include the "type" attribute with an assigned value	List of input elements from step 3	Type attributes are found and contain proper, allowed values, e.g. type="text"	Input attributes and types are described here: https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input#input_types

1: Which exercise (0-3) does the test belong to
2: Test case unique ID and name
3: Short description of what will be tested
4: Describes criticality of the test
5: Other test cases that need to PASS in order to potentially PASS this one
6: State of the test suite after executing this test case
7: Test steps; includes action to take, input to use and expected outcome

Figure 6: Example of a test case card as a document.

In this example the test case was derived from part 0 sub-exercise *0.1 HTML* where the course material asks the student to design a HTML page with "*at least an anchor (link), a table, a list, an image and a form with some input elements. – You can leave the "action" attribute of the form undefined*." [38] Similarly cases were also crafted for testing the implementation of other elements mentioned in this sub-exercise as well as the other exercises covering parts 0 – 3 of DTEK2040.

As it was briefly discussed during Section 2.4 regarding test design and development, the goal while designing these tests was not to detect the most errors with the subject under test but rather to detect the most relevant errors in terms of meeting the learning goals.

In terms of the prototype, following task descriptions often meant the tests were designed to verify quite straightforwardly any mentioned functionalities while also relying to part specific example

material to gain an understanding of what was the expected style of execution for any given functionality.

5.4.1 Part 0 – Basics of web applications

DTEK2040 part 0 included in total five task sections, of which two dealt with coding tasks. The two coding sections of part 0 focused on HTML and CSS respectively. In general, the student was asked to design a web page that would fulfil a given set of requirements and then additionally style it using CSS in a way that would again conform to given styling rules.

Test case designs for these tasks favoured the static method of testing. One major factor that contributed to the decision of designing static tests was that dynamic testing appeared to be too forgiving: once the student submitted source code was handled by a HTML parser to be displayed in a browser, many of the errors in the raw source code were in fact corrected. For example, a web page that was scraped for elements through a browser session would always pass verification of correct HTML anatomy since a parser would add any missing elements. The same observation also applied to table and list anatomy: child elements could be left undeclared and parsers would appear to fix the hierarchy when source code was viewed through a dynamic session.

Out of the 17 automated tests for part 0 exercises, 14 fall under the static testing methods. These tests were written from almost a unit level testing perspective, and they seek to verify the presence and implementation of specific HTML elements and CSS stylings as required and guided by the course material or explicitly linked external sources. Due to the already mentioned issues in using HTML parsers when scraping student submission source codes, many of the test cases could not take advantage of Beautiful Soup to perform required source code scraping with readily available tools. Instead, regular expressions were heavily utilized throughout static tests because RF can support the usage through libraries such as *BuiltIn* and *String*.

Part 0 is unique amongst the exercises in a sense that it had plenty of explicit references to an external resource: MDN Web Docs[8]. Thus, test assertions were formed by cross-referencing the course study material for Part 0, the explicitly linked resources and requirements given by exercise tasks. Basis are presented alongside each static test on the next page in Table 13 [pp. 69].

[8] MDN Web Docs is a repository that contains a lot of documentation regarding web standards as well as developer guides. Available at https://developer.mozilla.org/en-US/ (16/10/2022).

Table 13: Static automation tests implemented for DTEK2040 exercise 0.

Exercise 0 static test cases		
ID	**Test case name**	**Basis From Course Material**
E0-T1-1	Verify Html Anatomy	Part 0 study material, “Traditional web application”, example source code. Part 0 exercises, Section 0.1 HTML: explicitly provides a link to Mozilla HTML tutorial that describes the anatomy of an HTML document.
E0-T1-2	Page Contains A Valid Anchor Element	Part 0 exercises, Section 0.1 HTML: requires that the page must contain an anchor (link) and explicitly links to <a> element on mdn web docs. Documents mention a hyperlink is created in combination with <a> and its href attribute.
E0-T1-3-1	Page Contains A Table Element	Part 0 exercises, Section 0.1 HTML: requires that the page must contain a table and explicitly links to <table> element on mdn web docs.
E0-T1-3-2	Page Contains A Table Element With Proper Hierarchy	Mdn web docs reference clearly dictates permitted content for <table> and element hierarchy.
E0-T1-4-1	Page Contains A List Element	Part 0 exercises, Section 0.1 HTML: requires that the page must contain a list and explicitly links to element on mdn web docs.
E0-T1-4-2	Page Contains A Valid List Element	Mdn web docs reference dictates permitted content and that at least one must be included as a child.
E0-T2-1	CSS File Exists And It Is Referred In The Index Html File	Part 0 study material, “CSS”, teaches the method of separating CSS from HTML and referencing it in <head>. Part 0 exercises, Section 0.2 CSS: Explicit link to Mozilla’s CSS tutorial, which also teaches the same method.
E0-T2-2	Hover Style Is Defined	Part 0 exercises, Section 0.2 CSS: requires a style rule for anchor element on hover action.
E0-T2-3-1	List Style Is Defined	Part 0 exercises, Section 0.2 CSS: requires a style rule for lists.
E0-T2-3-2	Table Style Is Defined	Part 0 exercises, Section 0.2 CSS: requires a style rule for tables.
E0-T2-5-1	Id Selector Is Used	Part 0 exercises, Section 0.2 CSS: requires the use of id selector to define styles.
E0-T2-5-2	Class Selector Is Used	Part 0 exercises, Section 0.2 CSS: requires the use of class selector to define styles.
E0-T2-6-1	Element Position Options Are Used	Part 0 exercises, Section 0.2 CSS: asks to experiment with ways to position elements.
E0-T2-6-2	Element Sizing Options Are Used	Part 0 exercises, Section 0.2 CSS: asks to experiment with ways to adjust the size of elements.

Dynamic testing was applied in test cases that would still be able to successfully fulfil their verification goal even without the mentioned prettification of source code by a HTML parser.

Essentially the dynamic tests for Part 0 exercise could have been created as static tests as well using the regular expression approach, but from development perspective dynamic versions were not only more straightforward but also faster to create. This was due to the utilization of *Browser* library and the ability to, for example, integrate asserts to keywords that would also interact with the web page. All in all, three dynamic cases were identified, and they are briefly introduced with their basis in Table 14.

Table 14: Dynamic automation tests implemented for DTEK2040 exercise 0.

Exercise 0 dynamic test cases		
ID	**Test case name**	**Basis From Course Material**
E0-T1-5-1	Page Contains An Image Element	Part 0 exercises, Section 0.1 HTML: requires that the page must contain an image and explicitly links to <img> element on mdn web docs.
E0-T1-5-2	Page Contains A Valid Image Element	Mdn web docs dictate that src attribute is required.
E0-T1-6	Page Contains A Form With Input Elements	Part 0 exercises, Section 0.1 HTML: dictates that the page must contain a form and explicitly links to <img> element on mdn web docs.

Decision to use dynamic testing in these cases was additionally affected by the perception that doing so would in fact increase maintainability. Utilization of library offered keywords without extensive regular expression variables means that if something in tasks or course material changes, there are less variables such as regular expressions that would need to be re-engineered to fix any broken tests.

5.4.2 Part 1 – React and JavaScript

Part 1 exercises introduce the student to React and JavaScript. Exercise consisted of 10 sub-tasks that were divided between the student developing two projects: (1) *Courses* application and (2) *Feedback* application. With Part 1 the test cases were numerically designed to separate into static and dynamic tests quite evenly. However, since the tasks regarding *Courses* application had plenty of code examples available in the exercise material, this reinforced the decision to favour static testing for these tasks.

As it was noted in Section 2.3 of this study static testing of source code should always start with reducing diversity. In this case, it seemed reasonable to hypothesize that such abundance of examples would naturally lead reduced diversity in student submitted source codes for these tasks, which would work well for static assessment of objectives such as code structure. Though,

Feedback application also had a few code structure related requirements extractable from the tasks, as can be seen in Table 15, which presents the collection of static test cases.

Table 15: Static automation tests implemented for DTEK2040 exercise 1.

Exercise 1 static test cases		
ID	**Test case name**	**Basis From Course Material**
E1-T1-1	Course App Consists Of Three Main Components	Part 1 exercises, Section 1.1 Refactoring components and using props: requires the application to consist of *Header*, *Contents* and *Total* components.
E1-T1-2	Course App Contents Consist Of Part Components	Part 1 exercises, Section 1.2 More components: requires the *Contents* component to consist of three *Part* components.
E1-T1-3	Course App Data In Objects	Part 1 exercises, Section 1.4 Objects in an array: provides an example code for variable data and requires the application to be modified accordingly.
E1-T1-4	Course App Props Are Passed Directly	Part 1 exercises, Section 1.4 Objects in an array: requires that objects should not be passed between components but rather be passed directly using an array. Provides an explanative example code.
E1-T2-3	Feedback App Contains Several Components	Part 1 exercises, Section 1.8 Feedback app, part 3: Requires the application to contain at least *Button*, *Statistics* and *Statistic* components.

As might be expected from the prior description of static cases covering most of the *Courses* application tasks, the opposite is true for dynamic test cases. While *Courses* has only one dynamic test case dedicated to it to test a baseline requirement for any web application, bulk of the dynamic tests cover the *Feedback* application functionalities. Designing the tests in Table 16 leaned towards black box approach and applied Myers' heuristics [pp. 17-18] to identify relevant equivalence classes and extract inputs from task descriptions. Part 1 exercises offered the chance to apply state transition testing with test cases E1-T2-2 and E1-T2-4.

Table 16: Dynamic automation tests implemented for DTEK2040 exercise 1.

Exercise 1 dynamic tests		
ID	**Test case name**	**Basis From Course Material**
E1-T1-5	Browsing The Course App	Expert interview, discussion regarding baseline requirements: application must fulfil the basic requirement of compiling in case of a react app and it must open in browser.
E1-T2-1	Feedback App Contains Three Buttons	Part 1 exercises, Section 1.6 Feedback app, part 1: requires the application to include three buttons.
E1-T2-2	Feedback App Statistics Are Hidden When Empty	Part 1 exercises, Section 1.9 Feedback app, part 4: requires no statistics are initially displayed.

(to be continued)

Table 16 (continues)

E1-T2-4	Feedback App Displays Statistics	Part 1 exercises, Section 1.7 Feedback app, part 2: requires the application to display statistics based on the given feedback (button choices).
E1-T2-5	Feedback App Statistics Contents Are In HTML Table	Part 1 exercises, Section 1.10 Feedback app, part 5: requires the statistics to be shown as an HTML table.

Overall, both test case types were quite easily implemented with available RF libraries and did not require extensive efforts to craft custom keywords to achieve test goals. The static test cases in Part 1 exercises again had to rely a lot on utilization of regular expressions while still managing to stay in relatively low numbers of lines of code per test case. This is most likely due to the reliance on low diversity of source code for submitted test objects. Also, most of the assert operations in static test cases were possible to be implemented by mainly relying on libraries *Builtin* and *String*, which were stable sources of keywords in Part 0 static tests as well. This allowed the amount of custom coded keywords to stay low, reducing the total lines of code for the tests as well.

5.4.3 Part 2 – Communication with server

Part 2 extended the study of topics covered during Part 1, but also introduced web application interaction with server. Part 2 exercises covered in total 10 sub-tasks that were again split between two different projects: (1) continuing the *Courses* application from Part 1 and (2) *Phonebook-front* application that also included the interaction between frontend and server.

As for the test cases for Part 2, static and dynamic tests were not as disproportionately divided between applications as they perhaps were with previous exercise, which was probably due to both projects in Part 2 having tasks focusing on functional aspects rather than code form or content details. While assessing content structure was still implemented as a goal for majority of static tests in Part 2, as a method it also usable for generally fulfilling the set requirement of verifying requirements set upon certain files external to the core application itself in the *Phonebook-front* project. Static tests for part 2 are presented in Table 17 [pp. 73].

Table 17: Static automation tests implemented for DTEK2040 exercise 2.

Exercise 2 static tests		
ID	**Test case name**	**Basis From Course Material**
E2-T1-1	Content Is Structured Correctly	Part 2 exercises, Section 2.1 Course contents: gives a required component structure of the application.
E2-T1-3	Module Separation Is Implemented	Part 2 exercises, Section 2.3 Separate module: requires the component *Course* to be implemented as a separate module that is imported by *App*.
E2-T2-2	App Is Divided Into Several Components	Part 2 exercises, Section 2.7 Telephone directory, part 4: requires the application to be refactored and separating a minimum of two components from the root component.
E2-T2-4	Initial App State Is Stored Into File	Part 2 exercises, Section 2.8 Telephone directory, part 5: requires that initial entries are stored into *db.json* file and file placed into root directory of the project. Json format of entries is given as an example.

As can be seen in Table 17, some of basis for static tests in Part 2 regarding contents and structure are already familiar from Part 1. Designing these test cases thus followed same principles as earlier. However, initiating these static tests for automation proved to be a lot more challenging than previously due to the file and directory structure varying a lot between student submissions.

Dynamic test cases presented in Table 18 relied on a database state especially crafted so that the content could be predicted and counted on for test assertions and only the submitted applications interactions with the server would be under test.

Table 18: Dynamic automation tests implemented for DTEK2040 exercise 2.

Exercise 2 dynamic tests		
ID	**Test case name**	**Basis From Course Material**
E2-T1-2	Exercises Total Number Is Calculated And Displayed	Part 2 exercises, Section 2.2 Number of exercises: requires the total number of exercises be added to the application and displayed as shown in an example image that is provided.
E2-T2-1	Prevent Adding Already Existing Name UI	Part 2 exercises, Section 2.5 Telephone directory, part 2: requires that application prevents adding the proposed entry if the name exists.
E2-T2-3-1	Contact Can Be Deleted UI	Part 2 exercises, Section 2.10 Telephone directory, part 7: requires that an entry can be removed with a button attached to each directory and provides an example image.

(to be continued)

Table 18 (continues)

E2-T2-3-2	Contact Can Be Deleted API	Part 2 exercises, Section 2.10 Telephone directory, part 7: requires that an entry is deleted from the server with an HTTP DELETE request to *localhost:3001/persons/:id*
E2-T2-5	Initial State Is Fetched From Server	Part 2 exercises, Section 2.8 Telephone directory, part 5: requires that application to return entries as a list with HTTP GET request to *localhost:3001/persons*. Part 2 exercises, Section 2.9 Telephone directory, part 6: requires that application state is synchronized with server state.

Part 2 exercises offered chances to further apply black box techniques in terms of designing tests presented in Table 18. E2-T2-1 and E2-T2-3-1 as well as E2-T2-3-2 were prime candidates for cause-effect graphing while mapping rule sets with expected outputs in a decision table. Testing the interactions between frontend and server also presented the opportunity to apply condition coverage dynamic white box technique while designing E2-T2-1 and E2-T2-3-2. Though, it must be said that using them in this context perhaps would technically fall under so the called grey box technique as it cannot be known for certain in detail what the student submissions implementation of API interactions or UI design is, which in turn may cause the test cases to be unreliable if assumptions of submission diversity are wrong.

5.4.4 Part 3 – Web application with database

Part 3 built on top of the prior parts by introducing basics of document databases and switching from interacting between frontend and a local JSON-server to interacting between frontend and Mongo database. Part 3 exercises contained 12 sub-tasks in total which also included a single task about deploying an application to a cloud application platform called Heroku. However, verifying the cloud application deployment had to be excluded from the scope of test automation for this prototype due to so many case study materials containing either old or invalid Heroku application addresses, no address at all or assumably an old or invalid Mongo backend connection for their cloud application.

Unlike other exercises of DTEK2040, a decision was made to approach test case design for Part 3 purely from dynamic testing approach. The main reason for this decision was that the exercise tasks set very few strict guidelines or code-level detailed hard expectations for student implementations. In other words, while Part 3 obviously has quite specific requirements for application functionalities, it still leaves the student room to manoeuvre with implementation details and as

such designing static tests to compliment verification of required functionalities was seen as too much of a challenge in terms of return on invested time resources. Thus, all the required functionalities are tested by the seven dynamic test cases presented on the next page in Table 19.

Table 19: Dynamic automation tests implemented for DTEK2040 exercise 3.

Exercise 3 dynamic tests		
ID	**Test case name**	**Basis From Course Material**
E3-T1-1	Json Array Is Returned From A Localhost Server	Part 3 exercises, Section 3.1 Phone directory backend, part 1: requires that entries are returned from *localhost:3001/api/persons* as a json array. Provides an example image of entries returned in json format. Part 3 exercises, Section 3.10 Phone directory and database, part 1: requires that the entries are fetched from database.
E3-T1-2-1	Single Id Request Successful	Part 3 exercises, Section 3.2 Phone directory backend, part 2: requires that a single entry is returned from *localhost:3001/api/persons/:id*.
E3-T1-2-2	Single Id Request Unsuccessful	Part 3 exercises, Section 3.2 Phone directory backend, part 2: requires that if id is not in directory, must respond with appropriate status code. Part 3 study material, “Fetching a single resource”, mentions the use of status code 404 in such cases as an example.
E3-T1-3	Contact Can Be Deleted Using DELETE	Part 3 exercises, Section 3.3 Phone directory backend, part 3: requires that a HTTP DELETE request to *localhost:3001/api/persons/:id* removes the entry with given id. Part 3 exercises, Section 3.12 Phone directory and database, part 3: requires that removal must also affect database contents.
E3-T1-4	Contact Can Be Added Using POST	Part 3 exercises, Section 3.3 Phone directory backend, part 3: requires that a new entry can be added with HTTP POST request to *localhost:3001/api/persons*. Part 3 study material presents HTTP POST request examples that are sent with content data in json format.
E3-T1-5-1	Entry Adding Error Handling UI	Part 3 exercises, Section 3.5 Phone directory backend, part 5: requires that adding a new entry should not be allowed if a) name or number is missing or b) name already exists in directory. Provides an example error response.
E3-T1-5-2	Entry Adding Error Handling API	Part 3 exercises, Section 3.5 Phone directory backend, part 5: requires that adding a new entry should not be allowed if a) name or number is missing or b) name already exists in directory. Provides an example error response.

Test designing was approached like in Part 2 with emphasis on black box techniques. To be able to trust and make certain expectations regarding the database connection and contents of the document database, a decision was made to use a local Mongo database as the test backend. For this to work the preparation process leading to dynamic testing needed to parse and replace proper database connection strings into each submission currently coming up as a system-under-test. The preparation process also needed to include verification of other minute details such as which ports were defined for frontend and backend applications.

Preparative tasks required prior to executing these test cases rely on certain heuristics to attempt and locate parts of the submitted material that needs to be edited to get all the connections working. During the design and development process it was often noticeable that discovering such a heuristic to reliably work for every submission currently provided for case study purposes would be a challenge. Due to this the prototype can function with certain kinds of submission directory and file structures but cannot guarantee to work with all the possible variations.

5.5 Analysing and evaluating the design

5.5.1 Meeting the set requirements

Earlier in this study a few requirements for automation were crafted. These requirements are presented in detail within Appendix B, but the targets of these requirements were:

- GR-1: Processing takes less than 5 minutes per submission.
- GR-2: System contains documented and explanative general architectural overview; test case descriptions and custom keywords are 100% commented; custom library methods are 100% commented.
- GR-3: Course personnel can access the final summary document and change individual scores if needed; summary provides reasoning for individual task scores.
- GR-4: System can successfully cover step number 3 of Appendix A: Prepare the fetched submission for assessment.
- GR-5: System can successfully cover steps number 3 – 6 of Appendix A from technical perspective.

The metric used to observe whether the prototype would meet GR-1 was the time it took in seconds to execute the whole pipeline from start to finish. Total time of a single execution from start to finish was determined by integrating the measurement into the pipeline: when pipeline started execution, time current time was captured with command *date -u +%s*[9] and before the final *exit 0* command *date -u +%s* was used again and the difference between end time and start time calculated and logged by appending the result into an external log file. Static and dynamic test steps, the total execution time for both steps four and six, were recorded the same way to gather results of test set execution times as well.

Each exercise set was executed 10 times and the execution times logged as described. More detailed results of these runs are available in Appendix D, but on average the execution times were observed to be as represented in Table 20.

Table 20: Average execution times per submission.

	Average execution time in seconds per submission			
	Static test step	**Dynamic test step**	**Other processes**	**Pipeline total**
Exercise 0	3,10	26,36	1,36	30,82
Exercise 1	1,77	30,51	1,81	34,09
Exercise 2	8,51	43,06	2,12	53,69
Exercise 3	No tests	35,06	2,20	37,26

Based on the observed execution times, the prototype was able to clearly fulfil GR-1 [Appendix B]. Exercise 2 was observed to have the longest total running time on all measured categories with a total execution still managing to stay below one minute when averaged for a single submission.

It must be considered that when averaging times for a single submission of a batch run that includes multiple submissions, some of the pipeline steps are repeated for each while some are performed only once. Overall performance benefits from multiple submission batch runs as then steps 1, 2, 3, 5, 7 and 8 are only executed once in total and steps 4 and 6 for each submission. Thus, there is relatively less time spent on other-than-testing processes for each submission.

The average times as presented in Table 20 can also only be used for certainty to provide evidence that there is potential to perform assessment, and especially functional assessment, of student submissions in a considerably faster paced manner than the suggested 5 minutes on average per submission estimate raised in expert interview, not including the gathering and providing of

[9] Command results into a value presenting seconds (%s) since 1970-01-01 00:00:00 UTC.

qualitative feedback to the student. This claim ties into the quality of the currently implemented prototype test cases, which will be analysed later in a sub-section of their own.

The documentation requirements of GR-2 [Appendix B] are fulfilled with the materials produced through this study. The general architecture overview is included as contents of Section 5 with any relevant supporting material. As for the comment requirements for source codes, the prototype project is available online[10] with custom keywords and custom library content commented to 100% coverage. Common keywords resource and the custom library are also added as Appendix F and Appendix G of this thesis. Coverage to measure was selected to be method description coverage where a custom keyword or a method included inside a custom library was determined to be covered when comments described (1) what does the keyword / method do, (2) what does it require as input and (3) what does it produce as output.

GR-3 is fulfilled by the pipeline managing to produce a summary file that includes PASS / FAIL results of each submission at the level of individual test cases. Test cases themselves aim to test individual features or aspects of features so that the results can be used as guidance for feedback. Additionally for dynamic tests, whenever necessary, the system gathers screenshots that are also included as final artifacts. These screenshots of student submitted applications in action can be used to verify if the system has seemingly made a just call in passing or failing the subject-under-test for any given dynamic test.

GR-4 and GR-5 are fulfilled as described in the prior sub-sections of Section 5, especially the sub-section 5.3 where individual steps of the pipeline are recognizably corresponding to the respective steps mentioned in Appendix A. However, it is arguable that the prototype receives a pass with comments in terms of these general requirements. Since the prototype is built to assume certain kinds of submissions it is not able to cover for all the possible variety among submissions.

5.5.2 Quality of the test cases

As mentioned during this study, test cases were designed by extracting requirements from the course assignments and study materials and then developed so that the provided case study materials could be used as test subjects. Due to the diversity of submissions in various aspects, some of the provided case study material had to be rejected because attempting to cover every

[10] Prototype repository is available on UTU GitLab (https://gitlab.utu.fi/tesalo/salomaa-thesis), GitLab (https://gitlab.com/tomisalomaa/salomaa-utu-thesis-prototype) and GitHub (https://github.com/tomisalomaa/salomaa-utu-thesis-prototype). GitLab repositories have a working example of the CI-implementation. Main branches are frozen to present the state of the implementation described in this thesis.

possible student approach to any given task simply was not realistic. Realisation of this during the development process also spoke for the notion discovered from background theory that standardization of the target process and its aspects is a precondition for successful RPA implementation.

To gain insight into the usability of test cases as they are, result artifacts from 10 pipeline executions per each exercise were compared to manual results provided with the case study data. Subjects that received no points at all for a given exercise during the 10 pipeline executions were discarded from the comparison since the failure to gain any points at all was in these cases observed to be the result of an unorthodox submission that deviated from the standard assumed. In total, results for seven submissions of exercise 3 had to be removed from the comparison. Reasons for removal included

a) three cases of missing component files;

b) two cases of unorthodox directory structure causing the prototype not to be able to find and/or repair submission contents to run the application;

c) one case of using JSX syntax extension instead of JS;

d) one case of implementing material UI library resulting in application crash due to pipeline not supporting this.

Table 21: Comparison of average prototype results versus manual results.

		Points available		Proto versus manual average result difference	
Test set	**Submissions**	**Course**	**Prototype**	**Points received**	**Percentage points**
Exercise 0	96	5	2	0,13	6%
Exercise 1	104	10	10	-3,40	-34%
Exercise 2	68	10	10	-1,28	-13%
Exercise 3	47	12	12	-2,57	-21%

Based on the measured comparisons presented in Table 21, the Exercise 0 test set managed to score quite close to the manually scored results, granting on average 0,13 points per submission more than the provided manual results. Exercise 0 of DTEK2040 consists of two coding tasks and three

non-coding tasks which causes the total points in manual results to receive a maximum value of 5. As the prototype only considers and tests coding tasks, the maximum point was set to value of 2.

The case study data did not break down the manually given total score per subject and this must be considered when attempting to analyse these results; unless a student has received a perfect score from manual assessment, it would be impossible to tell for certainty how successful were the submitted coding tasks. If all the subjects scoring less than the maximum from manual assessment were removed to be sure comparison is done against subjects that received full score from coding tasks, and then the results compared again, the average points received from prototype assessment of Exercise 0 test set is -0,15 points or -8% percentage points. This would then be in line with rest of the test sets, as they commonly resulted into worse total scores on average than their manual comparisons.

From the score results alone, it seems like dynamic tests contribute to greater difference between manual and current prototype assessment scores. This is most likely since the automated tests cover one type of a solution whereas manually assessing the submissions allows for granting points for a variety of ways to solve the given task. For example, the course assignments dictate API paths to use in exercises 2 and 3 and these have been used precisely in the design and development of automated tests. If a submission then happens to use request paths such as */persons/:id* instead of */api/persons/:id* this would always result in a failed results as the automated tests would fail their API requests whereas a human manually assessing the submission might be inclined to not consider this a problem as long as the functionality is correct.

Static tests seemed to also be a bit less reliable for exercises 2 and 3 since source code contents had more variance in component and overall variable naming. While the course material provided some examples in the assignments of these exercises, the actual functionalities are of course achievable without using the exact component content structure and naming as in the examples. Currently the prototype, however, is not able to cover all such cases and it might fail a test case for a submission simply because a regular expression phrase used in a static test does not cover the exact form a student has used for a variable in his or her source code, for example.

All in all, the current prototype test cases are very strict in terms of interpretating the course assignments and the individual tasks contained within. However, the assignments themselves are not at times standardized or detailed enough in terms of dictating strict implementation, which leaves students some room for manoeuvring when it comes to their source code. In fact, the assignments seem to drive student implementations towards a development style one might describe

as example-driven development. From a pedagogical standpoint allowing the student to express oneself and even perhaps attempt alternative approaches is not necessarily a bad thing, especially for the latter exercises, but it also creates considerable challenges for automation.

5.5.3 Answering the main research questions

The first research question presented at the beginning of this thesis was Q1: *How to support the assessment of web application programming assignments with test automation?* To build upon the foundation already formed by answering the related supporting research questions in Section 2.6, by also integrating the results and data gathered from prototype executing, the following answer to Q1 is collected:

1) Course assignments must be designed or adapted to support the idea of using test automation to gain significant benefits; building test automation on top of an already existing course material is a possibility but the existing assignments may allow for too much submission variance in terms of submission content or the process of returning said submissions.

2) In general, the test design strategy of starting with cause-effect mapping of potential test input combinations and proceeding with supplementary boundary value analysis and equivalence classes was verified empirically to work well. Further enhancing automated test sets through experience-based error-guessing technique is seen as a viable continuation for this strategy. This strategy should be preceded by gathering the test basis together with the relevant personnel who are familiar with the requirements set for assessing submissions during the manual process to ensure test assertions are in line with what is currently being tested; extracting requirements from assignment descriptions alone may lead to very strict asserting where tests are failing even if the desired functionality exists.

3) The focus of automated testing can be kept in functional testing, which naturally lends itself to applying dynamic testing approaches with black box techniques while developing test cases. While the functional testing can be further supported with complimentary tests taking the static testing approach, this focus can also be seen to free up manual testing efforts and allow the course personnel to focus more on other quality aspects of the submission.

The second research was Q2: *How to support the assessment and feedback process of assignment assessing with RPA?* This question was also based by answering its respective sub-questions in Section 3.4. The full answer, however, could be collected as follows:

1) First step is to identify the overall process that is currently being followed to assess the assignments. This will allow for discovering automation potential within the process and determine overall whether the process is a good candidate for RPA; the automation candidate should ideally have high transaction volumes, be highly standardized, have well-defined implicit logic and be mature enough so that the process itself will not be dynamically changing as the automation is being implemented.

2) Within the context of this thesis RPA was observed to be successful in performing file manipulation tasks in terms of assessment record keeping and tracking, manipulating the student submissions file and directory structures to perform test preparations, verifying submission contents, and summarizing quantitative feedback.

3) RPA potential for discovering and presenting qualitative feedback was not empirically proven within the scope of this thesis, but it was theoretically researched and discussed. As an aspect it is something that could be integrated by the means of RPA to support the feedback process while also offering potential to integrate novel technologies to make it happen.

4) RPA has big potential to shorten the feedback loop if the course assignments are built to allow RPA support. For example, by integrating RPA together with test automation into a study platform such as the *Web and Mobile Programming* website, the technology would perhaps allow for partial submissions to be reviewed and assessed instantly while also providing feedback to the student as he or she is working on the assignment solution.

Overall, the collected answers for Q1 and Q2 are written through what additional observations were made through analysing the results gathered from the design, development and testing process of the prototype. As the empirical observations did not contradict the answers and the discussion related to the respective sub-questions, it is noteworthy to mention that the details discovered in those answers are also relevant even if they were not repeated here.

5.5.4 Suggestions and further potential

As it was demonstrated, in general the pipeline and automation is able to meet the requirements set for the prototype in the context of this study. The pipeline itself, however, can surely be further optimised in terms of performance by introducing parallelization, for example. The system could perhaps also be further extended to cover more ground by integrating it to either Moodle workspace, which is currently used as a platform for students to submit their assignment works and

course personnel to mark assessments and provide feedback, or the DTEK2040 course website. Integration with the course website could even provide opportunities to automate assessment of partial assignment submissions and thus allow the students to receive feedback while they are implementing their full assignment submission.

The static and dynamic test cases themselves are also an obvious area of potential improvement, as can be seen from the average results displayed earlier. The fact that presenting their potential and possibility to integrate them into a pipeline was at the centre of design and development work rather than the usability also means that they cannot be suggested to be taken into use without further development work. One way to make the tests more usable and able to serve the needs of DTEK2040 – or any other course for that matter – would be to first focus on the requirements engineering of these tests together with course personnel and then start iteratively improving them. Preferably course assignments and their content would also be modified so that the assignment contents could possibly be further standardized or detailed from test automation perspective given it does not degrade respective assignment's pedagogical worth.

As far as pedagogy is concerned, the ability to help provide the student formative feedback is an area to improve the prototype in. Attempting to integrate the theory discussed in Section 3.3 could even provide interesting venues to integrate novel technologies from the field of artificial intelligence to discover and formulate qualitative feedback points from student submitted materials. Other, perhaps a more realistic first step, would be to integrate static analysis methods and tools such as so-called "linters"[11] to flag programming errors and smells from submitted source codes and then further transform these into more constructive feedback to be added as part of the summary artifacts at the end of the pipeline.

[11] A linter is often described as a static analysis tool that specifically analyses code quolity by discovering redundant code, inefficient pattens, anti-patterns, smells and much more. These tools are commonly found in many IDEs and often integrated within CI pipelines in professional software development. Using linters could prove useful in discovering points for qualitative feedback, and many such tools exist for JavaScript. Some examples include ESLint (https://eslint.org/) , JSLint (https://www.jslint.com/) and Standard JS (https://standardjs.com/#javascript-style-guide-linter-and-formatter).

6 Conclusions and future work

This thesis studied test automation, RPA and the principles of valuable feedback in the context of automated assessment tools for programming assignments. The main research methods applied during the study were literature review, expert interview and design science. Thesis verified the test automation and RPA research empirically by using available anonymized student submissions from the course *DTEK2040: Web and Mobile Programming 2022* as case study material. As a design science artifact, a prototype capable of performing programming assignment submission assessment within the boundaries of set requirements was developed.

In total, the thesis answered to two main research questions and five supporting research questions. The first question, *Q1: How to support the assessment of web application programming assignments with test automation*, was answered by first researching which testing levels and techniques would be most suitable in general to test programming assignment submissions. Afterwards research was done on test design and development to implement the prototype automation test cases. The study discovered that unit tests and system tests were the most popular levels to focus on due to the heavy emphasis on functional testing in other similar systems. This was further supported by the expert interview, based on which it appeared to be the case with the currently manually performed assessment work as well. Tied to the focus on functional testing, research also suggested that dynamic testing performed with black box techniques would be the most likely approach to designing test cases while static testing would assume the more minor role of complimenting them.

The second question, *Q2: How to support the assessment and feedback process of assignment assessing with RPA*, was answered by first discovering what manual labor related to the assessment and feedback process there potentially is to automate and then extending the research to the pedagogical side of formulating feedback for programming assignments. To answer this question the study found out that the current processes of DTEK2040 contain elements such as (1) interacting with the learning platforms and workspaces, (2) file manipulation in terms of assessment record keeping and tracking, (3) manual testing of student submitted applications for structure, (4) functionalities and content, (5) gathering and representing quantitative feedback, (6) noting issues within submission for qualitative feedback and (7) maintaining feedback traceability were all potential targets for automation. As for the quality of the feedback itself, it was discussed that automation would be able to improve aspects such as shortening the feedback cycle and providing feedback even for the so called perfect submissions, which currently might be left with little

attention. A challenge for automation, however, was noted in formulating formative feedback that would appropriately target student's knowledge of concepts, made mistakes, expansion possibilities of current skill sets, capabilities of self-assessment and understanding of what is being expected of him or her i.e., knowledge about task constraints. All these feedback aspects were determined to be important when it came to effectively supporting student's growth in programming.

The main research questions were verified, and answers expanded upon, by empirically designing and implementing a prototype that integrated RPA and test automation. The prototype was able to process batches of student submissions continuously and reliably for each exercise set in timespans averaging less than a minute per submission. A total execution of the automated pipeline consisted of record keeping RPA tasks, preparative tasks for the static and dynamic tests, automation tests and finally summarization of results. Unfortunately, the prototype was not able to extensively integrate the pedagogical aspects of feedback formulation as it only focused on providing score summaries and highlighting failed functionality implementations within a given subject-under-test by including relevant parts of test logs into the final summary file. The analysis of the prototype did, however, provide insight into how more quality feedback could be integrated into the system as future suggestions. Overall, the most notable expansion to the answers already presented was that in order to effectively support assignment assessment with test automation and RPA, the use of automation needs to be considered already when designing course assignments; building automated solutions on top of already existing case study proved that while the course contents lead students to certain kinds of programming implementations, the submission content variance still proved to be a challenge for the prototype.

For future work two main aspects related to the created prototype were identified: (1) improving the test cases and (2) integrating a more extensive feedback ability. The first suggestion is to perform requirements engineering and iterative development processes with the personnel of DTEK2040, or any other course wishing to apply the prototype, to convert manual test cases into automated ones and take advantage of experiences in assessing submissions of that course for error-guessing tests. The second suggestion is that research be done onto possibilities of integrating static analysis tools into the current prototype to gather and provide more code quality feedback from submissions. This aspect most likely offers novel research potential in terms of attempting to integrate AI solutions for feedback formulation. Aside from improvements to an assessment support system, a third suggestion could also be proposed from another perspective: research on how a programming course is able to support and be ready to integrate automation tools for better learning quality.

References

[1] Ala-Mutka, K. M. (2005). A survey of automated assessment approaches for programming assignments. *Computer Science Education, 15*(2), 83-102.

[2] Amman, P., & Offutt, J. (2017). *Introduction to Software Testing* (2nd edition ed.). Cambridge University Press.

[3] Arora, A., & Sinha, M. (2012). Web Application Testing: A review on Techniques, Tools and State of Art. *International Journal of Scientific & Engineering Research, 3*(2).

[4] Baresi, L., & Pezzè, M. (2006). An introduction to Software Testing. *Electronic Notes in Theoretical Computer Science, 148*, 89-111.

[5] Bass, L., Clements, P., & Kazman, R. (2013). *Software Architecture in Practice* (3rd ed.). Addison-Wesley Professional.

[6] Bruns, A., Kornstädt, A., & Wichmann, D. (2009). Web Application Tests with Selenium. *IEEE Software, 26*(5).

[7] Daka, E., & Fraser, G. (2014). A Survey on Unit Testing Practices and problems. *2014 IEEE 25th International Symposium on Software Reliability Engineering.* IEEE.

[8] Di Lucca, G. A., & Fasolino, A. R. (2006). Web Application Testing. In E. &. Mendes, *Web Engineering* (pp. 219–260). Springer. doi:https://doi.org/10.1007/3-540-28218-1_7

[9] Doguc, O. (2020). Robot Process Automation (RPA) and Its Future. In U. Hacioglu, *Handbook of Research on Strategic Fit and Design in Business Ecosystems* (pp. 469-492). IGI Global. doi:https://doi.org/10.4018/978-1-7998-1125-1.ch021

[10] Douce, C., Livingstone, D., & Orwell, J. (2005). Automatic test-based assessment of programming: A review. *Journal on Educational Resources in Computing, 5*(3), 4.

[11] Eggert, R. J. (2010). *Engineering Design* (Second edition ed.). Meridian, Idaho: High Peak Press.

[12] Garousi, V., & Elberzhager, F. (2017). Test Automation: Not Just for Test Execution. *IEEE Software, 34*, 90-96. doi:10.1109/MS.2017.34

[13] Garousi, V., & Mäntylä, M. V. (2016). A systematic literature review of literature reviews in software testing. *Information and Software Technology, 80*, 195-216.

[14] Gupta, S., & Dubey, S. K. (2012). Automatic Assessment of Programming assignment. *Computer Science & Engineering An International Journal, 2*(1).

[15] Hambling, B., Samaroo, A., & Morgan, P. (2010). *Software Testing - An ISTQB-ISEB Foundation Guide* (2nd edition ed.).

[16] Hattie, J., & Timperley, H. (2007). The power of feedback. *Review of Educational Research, 77*(1), 81-112. doi:10.3102/003465430298487

[17] Hollingsworth, J. (1960). Automatic graders for programming classes. *Communications of the ACM, 3*(10), 528–529.

[18] Ihantola, P., Ahoniemi, T., Karavirta, V., & Seppälä, O. (2010). Review of recent systems for automatic assessment of programming assignments. *Koli Calling '10: Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, (pp. 86-93).

[19] International Software Testing Qualifications Board. (2021). Certified Tester Foundation Level (CTFL) Syllabus. Version 2018 v3.1.1.

[20] Jha, N., Prashar, D., & Nagpal, A. (2021). Combining Artificial Intelligence with Robotic Process Automation - An Intelligent Automation Approach. In K. R. Ahmed, A. E. Hassanien, & (eds), *Deep Learning and Big Data for Intelligent Transportation. Studies in Computational Intelligence, vol 945.* Springer, Cham. doi:https://doi.org/10.1007/978-3-030-65661-4_12

[21] Keuning, H., Jeuring, J., & Heeren, B. (2018). A systematic literature review of automated feedback generation for programming exercises. *19*, 1–43.

[22] Lehto, J., & Papalitsas, J. (2022, April 20). Expert interview. (T. Salomaa, Interviewer)

[23] Myers, G. J., Badgett, T., & Sandler, C. (2012). *Art of Software Testing (3rd Edition)* . John Wiley & Sons.

[24] Nicola, D. J., & Macfarlane-Dick, D. (2006). Formative assessment and self-regulated learning: a model and seven principles of good feedback practice. *Studies in Higher Education, 31*(2), 199-218. doi:10.1080/03075070600572090

[25] Nidhra, S., & Dondeti, J. (2012). Black box and white box testing techniques - A literature review. *International Journal of Embedded Systems and Applications (IJESA), 2*(2).

[26] Ouadoud, M., Nejjari, A., Chkouri, M. Y., & El-Kadiri, K. E. (2018). Learning management system and the underlying learning theories. *Proceedings of the 2nd Mediterranean Symposium on Smart City Applications* (pp. 732–744). Springer.

[27] Paiva, J. C., & Leal, J. P. (2022). Automated Assessment in Computer Science Education: A State-of-the-Art Review. *ACM Transactions on Computing Education*.

[28] Polo, M., Reales, M., & Ebert, C. (2013). Test Automation. *Software, 30*, 84-89.

[29] Rahman, K. A., & Nordin, M. J. (2007). A review on the static analysis approach in the automated programming assessment systems. *National conference on programming.*

[30] Ricca, F., & Stocco, A. (2021). Web Test Automation: Insights from the Grey Literature. *SOFSEM 2021: Theory and Practice of Computer Science: 47th International Conference on Current Trends in Theory and Practice of Computer Science*, (pp. 472–485). doi:https://doi.org/10.1007/978-3-030-67731-2_35

[31] Robocorp. (2022). *Basic concepts of Robot Framework*. Retrieved from https://robocorp.com/docs/languages-and-frameworks/robot-framework/basics

[32] Robot Framework ry. (2022). *Robot Framework User Guide*. Retrieved from https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html

[33] Romli, R., Sulaiman, S., & Zamli, K. Z. (2010). Automatic programming assessment and test data generation: a review on its approaches. *Int. Symp. in Information Technology*, 1186-1192.

[34] Stepien, B., & Peyton, L. (2009). Integration Testing of Web Applications and Databases Using TTCN-3. *Innovation in an Open World, 4th International Conference 2009.* Ottawa, Canada: MCETECH .

[35] Thummalapenta, S., Sinha, S., Singhania, N., & Chandra, S. (2012). Automating test automation. *2012 34th International Conference on Software Engineering (ICSE)*, 881-891. doi:10.1109/ICSE.2012.6227131

[36] Umar, M., & Chen, Z. (2019). A Study of Automated Software Testing: Automation Tools and Frameworks. *International Journal of Computer Science Engineering (IJCSE), 8*, 217-225. doi:https://doi.org/10.5281/zenodo.3924795

[37] University of Turku. (2022). *Study Guide 2020-2022: DTEK2040 Web and Mobile Programming, 5 ECTS*. Retrieved from https://opas.peppi.utu.fi/en/course/DTEK2040/8780?period=2020-2022

[38] University of Turku. (2022). *Web and mobile programming 2022*. Retrieved from https://tech.utugit.fi/education/webprog/web-material/

Appendix A: DTEK2040 assessment process automation potential

Table A1: Automation potential within steps extracted from the manual assessment process.

	Step	Manual tasks for assessor	Automation potential	Automation risks
1	Submit a solution	None		
2	Fetch a submission	1) Log into learning platform 2) Navigate to course workspace 3) Navigate to assignments and choose submission(s) to download 4) Assign the submission to yourself	Automation of tasks such as logging in, navigating and fetching content are among the most common website related RPA. The whole chain can be automated.	- The platform, in this case Moodle, may experience updates that break the robotic process; simple modifications to website element attributes such as id may cause navigation failures. - The platform may not allow RPA interaction, i.e. it attempts to block actions with the use of CAPTCHA or setting limits to requests.
3	Prepare the fetched submission for assessment	1a) Extract the compressed submission package, and/or 1b) Fetch submission from GitLab based on the link provided in a text file. 2) Set up environment and run the dynamic parts of the submission 3) Open-source files with proper tools for static analysis	- Recursive extraction of compressed packages is automatable. - Environmental set up can be performed with the help of technology such as Docker, which in fact increases security of assessing dynamic parts of the submission. - Automated static assessment requires less tools from preparation standpoint	- While the submissions are instructed to be returned in a certain way, there may still be variance between the contents and structure of different submissions for same assignment. - Relying on container technology also requires maintenance and knowledge of said technology.
4	Perform static assessment	1) Assess how the source code corresponds to the course subject contents 2) Search source code for known “anti-patterns” that commonly present themselves 3) Keep notes for scoring and feedback	- Automation can make static analysis and assessment more consistent and less reliant on assessor’s experience. - Experience-based observations of anti-patterns per assignment may prove to be easily automated for notating / flagging. - If any assignment requirements are clearly statically verifiable, they are also simple to score and note for feedback.	- Scripting the rules for more intermediate and advanced level static analysis may prove to be challenging and not necessarily be a good return on investment. - Providing in-depth feedback in fully automated manner may prove to be difficult without advanced features such as introduction of AI solutions.

(to be continued)

Table A1 (continues)

5	Perform dynamic assessment	1) Open / navigate to submission page / site 2) Test and assess required functionalities 3) Keep notes for scoring and feedback	- The assignments often lead to certain expectable solutions. - Functional assignment requirements can be verified with test automation. - If any assignment requirements are clearly dynamically verifiable, they are also simple to score and note for feedback.	- To truly achieve consistent behaviour from automated test cases, additional instruction for assignments may be needed. - Automated tests need to be maintained as course contents, such as assignments or technology stacks, are updated.
6	Form feedback	1) Gather a total score based on static and dynamic assessment 2) Provide written feedback based on errors and missed points	- This step contains administrative tasks such as forming a summary of results from previous steps; this is what RPA is generally considered to target most often in a business sense as well.	- The summarized contents may be likely to require manual validation before continuing forward from here.
7	Post results to learning platform	1) Make sure the just assessed submission is organized and filed as assessed 2) Mark score for the student submission in learning platform 3) Provide written feedback (if any) in learning platform	- Organizing the assessed workloads is something that helps manual working; automation may get rid of this task all-together or take a more archiving approach to it. - Automating website interactions to fill in submission scores and feedback from a source is quite straight-forward.	- See platform related risks from step number two.

Appendix B: General requirements for DTEK2040 automated assessment system

Table B1: General system requirements built based on theory sections.

General requirements for DTEK2040 automated assessment system		
ID	**Requirement**	**Metric**
GR-1	System must save human workhours in assessment and related bookkeeping tasks	Time spent per student submission. With batch runs the total time may simply be divided by the number of processed submissions. Target: Processing takes less than 5 minutes per submission
GR-2	System must be maintainable by course personnel	Documentation and instruction coverage. Targets: 1) System contains documented and explanative general architectural overview; 2) test case descriptions and custom keywords are 100% commented; 3) custom library methods are 100% commented.
GR-3	Human must be kept in the assessment loop	Assessment transparency and result modifiability. Targets: 1) Course personnel can access the final summary document and change individual scores if needed; 2) summary provides reasoning for individual task scores.
GR-4	Automation flow begins from the work step when packaged submissions have been fetched from course platform	Coverage of assessment process steps identified for DTEK2040. Target: System can successfully cover step number 3: Prepare the fetched submission for assessment.
GR-5	Automation flow covers preparation, numerical assessment, discovering issues for feedback and generation of summary report from the processed submissions	Coverage of assessment process steps identified for DTEK2040. Target: System can successfully cover steps number 3 – 6 from technical perspective.

Appendix C: Prototype directory and file structure tree

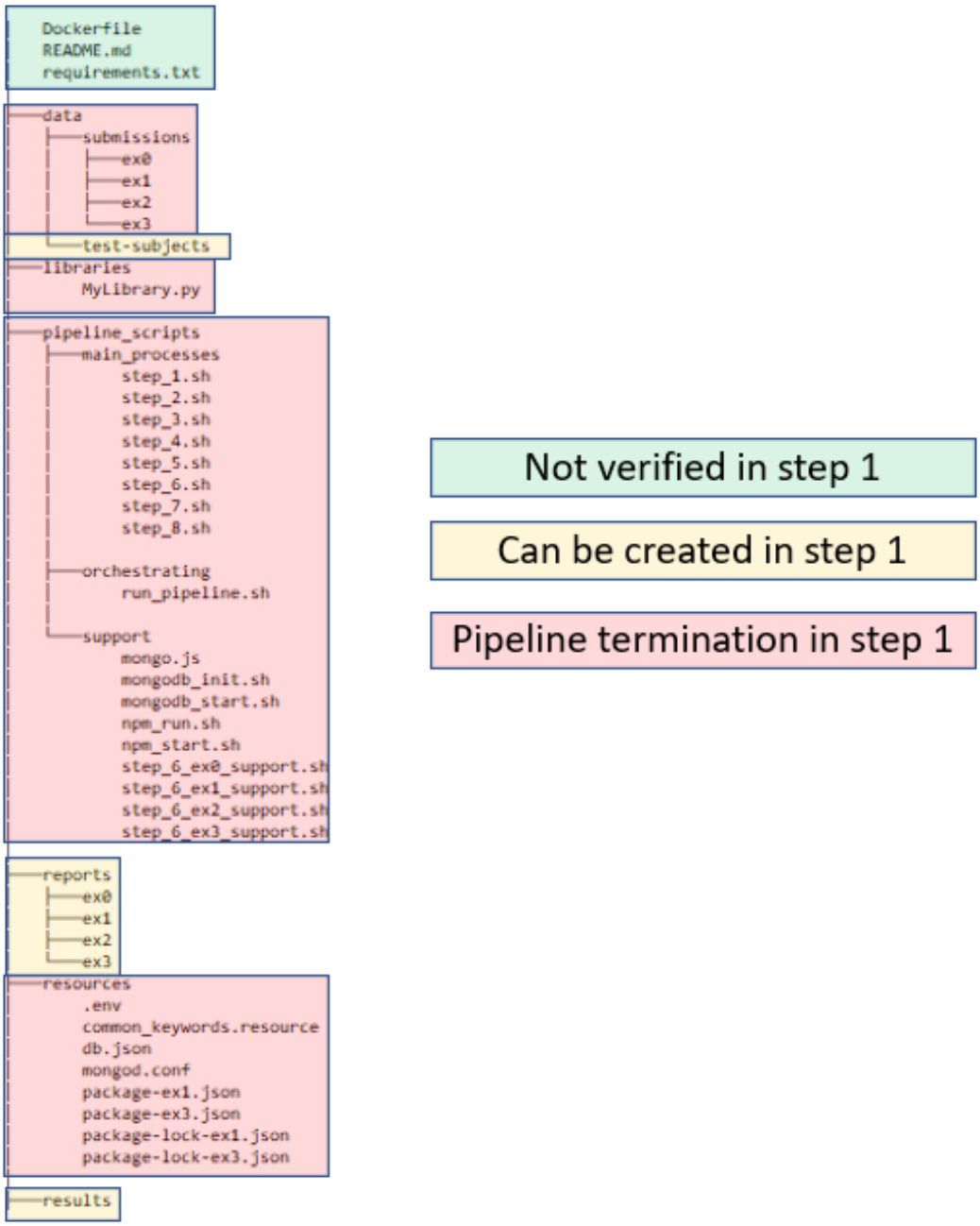


Figure C1: Prototype system file and directory structure.

The image represents files and directories found within the prototype system as they are by default. When verification of internal structure by the prototype is executed during pipeline step 1, the missing files or directories will have a certain effect to the continuation of the pipeline. Colour codes for files and directories in this image indicate how the system will act if the respective file or directory is not found when pipeline is executing the verification during step 1.

Appendix D: Pipeline execution times

Table D1: Pipeline execution times in seconds.

		Execution iteration											
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**average**	**average per submission**
Exercise 0	total	2974	2907	2884	2894	2903	3002	3005	3004	3005	3005	2958,3	30,82
	static	336	280	269	271	274	309	309	309	309	309	297,5	3,10
	dynamic	2498	2494	2487	2491	2501	2566	2567	2565	2568	2567	2530,4	26,36
	other processes	140	133	128	132	128	127	129	130	128	129	130,4	1,36
Exercise 1	total	3520	3540	3578	3558	3558	3521	3529	3554	3536	3561	3545,5	34,09
	static	194	182	189	185	182	181	181	182	181	182	183,9	1,77
	dynamic	3127	3172	3195	3178	3191	3154	3164	3185	3170	3194	3173	30,51
	other processes	199	186	194	195	185	186	184	187	185	185	188,6	1,81
Exercise 2	total	3658	3645	3568	3575	3651	3722	3616	3727	3706	3641	3650,9	53,69
	static	583	584	574	580	583	577	580	570	567	589	578,7	8,51
	dynamic	2917	2915	2850	2853	2922	3003	2896	3016	2999	2911	2928,2	43,06
	other processes	158	146	144	142	146	142	140	141	140	141	144	2,12
Exercise 3	total	1894	1961	1893	1925	1962	1973	2143	2080	2136	2155	2012,2	37,26
	dynamic	1769	1844	1776	1803	1844	1857	2026	1962	2018	2036	1893,5	35,06
	other processes	125	117	117	122	118	116	117	118	118	119	118,7	2,20

Table displays recorded execution times for all four exercises. Each exercise pipeline was executed ten times. Recording was done so that the orchestrating script *run_pipeline.sh* saved a running time value at the beginning and at the end of each step and before fully completing the pipeline, calculated the (1) total execution time, (2) static tests execution time and (3) dynamic tests execution time. These values were logged and appended into a log file automatically. Submission numbers used to calculate average per submission are: 96 for exercise 0, 104 for exercise 1, 68 for exercise 2 and 54 for exercise 3.

Appendix E: Example summary template

Exercise 2		COURSES APP			PHONEBOOK APP					
		Content structuring	Exercises total number is displayed	Module separation	Prevent adding already existing name	App is divided into several components	Contact can be deleted UI	Contact can be deleted API	Initial state is stored in a file	Initial state is fetched from server
	TOTAL	E2-T1-1	E2-T1-2	E2-T1-3	E2-T2-1	E2-T2-2	E2-T2-3-1	E2-T2-3-2	E2-T2-4	E2-T2-5
oppilas 10	4,444	PASS	FAIL	PASS	FAIL	PASS	FAIL	FAIL	PASS	FAIL
oppilas 100	8,889	PASS	FAIL	PASS	PASS	PASS	PASS	PASS	PASS	PASS
oppilas 104	7,778	PASS	FAIL	PASS	PASS	PASS	FAIL	PASS	PASS	PASS
oppilas 106	7,778	PASS	FAIL	PASS	PASS	PASS	FAIL	PASS	PASS	PASS
oppilas 109	8,889	PASS	FAIL	PASS	PASS	PASS	PASS	PASS	PASS	PASS
oppilas 11	6,667	PASS	FAIL	PASS	PASS	PASS	FAIL	PASS	FAIL	PASS
oppilas 112	7,778	FAIL	FAIL	PASS	PASS	PASS	PASS	PASS	PASS	PASS
oppilas 113	6,667	PASS	FAIL	FAIL	PASS	FAIL	PASS	PASS	PASS	PASS
oppilas 114	6,667	PASS	FAIL	PASS	FAIL	PASS	FAIL	PASS	PASS	PASS
oppilas 115	6,667	PASS	FAIL	FAIL	PASS	FAIL	PASS	PASS	PASS	PASS
oppilas 116	8,889	PASS	FAIL	PASS	PASS	PASS	PASS	PASS	PASS	PASS
oppilas 123	4,444	FAIL	FAIL	FAIL	PASS	FAIL	PASS	PASS	FAIL	PASS
oppilas 126	7,778	PASS	PASS	FAIL	PASS	PASS	PASS	PASS	PASS	FAIL
oppilas 13	8,889	PASS	FAIL	PASS	PASS	PASS	PASS	PASS	PASS	PASS
oppilas 131	8,889	PASS	FAIL	PASS	PASS	PASS	PASS	PASS	PASS	PASS
oppilas 136	0,000	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL
oppilas 138	8,889	PASS	FAIL	PASS	PASS	PASS	PASS	PASS	PASS	PASS
oppilas 142	6,667	PASS	FAIL	PASS	PASS	FAIL	PASS	PASS	PASS	FAIL
oppilas 145	8,889	PASS	FAIL	PASS	PASS	PASS	PASS	PASS	PASS	PASS
oppilas 146	7,778	PASS	FAIL	PASS	PASS	PASS	FAIL	PASS	PASS	PASS
oppilas 148	7,778	PASS	FAIL	PASS	PASS	FAIL	PASS	PASS	PASS	PASS
oppilas 15	8,889	PASS	FAIL	PASS	PASS	PASS	PASS	PASS	PASS	PASS
oppilas 151	8,889	PASS	FAIL	PASS	PASS	PASS	PASS	PASS	PASS	PASS
oppilas 154	7,778	PASS	FAIL	PASS	PASS	PASS	FAIL	PASS	PASS	PASS
oppilas 155	8,889	PASS	FAIL	PASS	PASS	PASS	PASS	PASS	PASS	PASS
oppilas 157	3,333	FAIL	FAIL	FAIL	PASS	FAIL	FAIL	PASS	FAIL	PASS
oppilas 161	3,333	PASS	FAIL	PASS	FAIL	PASS	FAIL	FAIL	FAIL	FAIL

Figure E1: Example representing summary template with results from ex2.

Appendix F: Common keywords

Code F1: common_keywords.resource.

```
# Common keywords contain keywords that are used or have potential to be used
# by multiple robot files now or in the future development.

*** Settings ***
Library  ..${/}libraries${/}MyLibrary.py
Variables  ..${/}variables${/}common_variables.py

*** Keywords ***
# Opens the summary report file using common variables.
# Using the keyword does not require inputs but the values for report path and
# summary file name in common_variables.py need to respect the project structure.
# File needs to be found for the keyword to PASS.
# @output: summary file is opened with id 'doc01'
Open Report File
  Open Excel Document  ${REL_REPORTS_PATH}${/}${SUMMARY_FILE_NAME}  doc01

# Iterates through a list of files and compares their line count.
# @input:   ${files}        a list of file paths
# @returns: ${result_file}  file path of the file containing most lines
Get File With Most Lines
  [Arguments]  ${files}
  ${line_count}  Set Variable  ${0}
  ${result_file}  Set Variable  ${files}[0]
  FOR  ${file}  IN  @{files}
    ${file_content}  Get File  ${file}
    ${new_line_count}  Get Line Count  ${file_content}
    IF  ${line_count} < ${new_line_count}
      ${line_count}  Set Variable  ${new_line_count}
      ${result_file}  Set Variable  ${file}
    END
  END
  [Return]  ${result_file}

# Searches a directory by using a keyword defined in MyLibrary.
# The search returns all .html file paths from that directory.
# From the returned list of file paths, the html file with most lines is chosen.
# @input:   ${submission_dir}     directory path containing the wanted html file
# @returns: ${html_file}        file path of the html file containing most lines
Search Local HTML Main Page Location
  [Arguments]  ${submission_dir}
  ${html_file_locations}  Search File With Extension  ${submission_dir}  html
  ${html_file}  Get File With Most Lines  ${html_file_locations}
  [Return]  ${html_file}

# Respects the current design of results summary sheet and inserts the given
submission id
# into the first column of the template.
# @input: ${student_row}    row number of the submission in summary sheet,
#         ${id}             id given to the submission
# @output: writes id into the summary sheet cell at column=1, row=given
```

(to be continued)

Code F1 (continues)

```
Insert Submission Id To Results Summary Sheet
  [Arguments]  ${student_row}  ${id}
  Write Excel Cell  row_num=${student_row}  col_num=1  value=${id}

# Searches in a given directory for files with a given extension that contain given
keywords.
# All the declared keywords must be found from the contents of a file in order to
determine that
# the file be returned. If a even a single word is not found from contents, the
file will be discarded.
# Keywords are treated as patterns. This means, for example, that given an input
keyword 'light',
# a match is found from contents including 'lightning', 'blight', 'lighter', ...
# Patterns are also treated case insensitively.
# @input:     ${parent_search_dir}    directory path to search files from
#             ${required_keywords}    a list of keywords to be pattern matched
within file contents
#             ${file_extension}       extension to determine what files should be
considered while searching,
#                                     example: js html
# @returns:   ${result_files}         a list of files that have positive matches
from each and every input keyword
Find Files With Content Containing Keywords
  [Arguments]  ${parent_search_dir}  ${required_keywords}  ${file_extension}
  ${keywords_amount}  Get Length  ${required_keywords}
  @{result_files}  Create List
  ${files}  Search File With Extension  ${parent_search_dir}  ${file_extension}
  FOR  ${file}  IN  @{files}
    ${total_found}  Set Variable  ${0}
    ${file_contents}  Get File  ${file}
    FOR  ${word}  IN  @{required_keywords}
      ${matches_found}  Get Regexp Matches  ${file_contents}  (?i:.*?${word}*.*?)
      ${amount_found}  Get Length  ${matches_found}
      IF  ${amount_found} == ${0}
        BREAK
      ELSE
        ${total_found}  Evaluate  ${total_found}+1
      END
      IF  ${total_found} == ${keywords_amount}
        Append To List  ${result_files}  ${file}
      END
    END
  END
  [Return]  ${result_files}

# Uses 'Get Modified Time' from OperatingSystem library to compare which of the
input files
# is most recently modified. Keyword is used with 'epoch' option which means the
modification
# time of a file is returned in seconds after the UNIX epoch.
# Use of this keyword required the robot file to import the OperatingSystem
library.
# @input:     ${list_of_files} a list of file paths to compare
# @returns:   ${recent_file} file path of the most recently modified file
```

(to be continued)

Code F1 (continues)

```
Determine Most Recently Modified
  [Arguments]  ${list_of_files}
  ${base_time}  Set Variable  ${0}
  ${recent_file}  Set Variable  None
  FOR  ${file}  IN  @{list_of_files}
    ${time}  Get Modified Time  ${file}  epoch
    IF  ${time} > ${base_time}
      ${base_time}  Set Variable  ${time}
      ${recent_file}  Set Variable  ${file}
    END
  END
  [Return]  ${recent_file}

# See 'Find Files With Content Containing Keywords' and
# 'Determine Most Recently Modified' keywords above.
# Uses the results of afore mentioned keyword as input for the latter.
# @input:   ${location}             directory path to search files from
#           ${keywords}             a list of keywords to be pattern matched within
file contents
#           ${file_extension}       extension to determine what files should be
considered while searching,
#                                   example: js html
# @returns:   ${most_recent_file}   file path of the most recently modified file
Find Most Recent File Based On Keywords
  [Arguments]  ${location}  ${keywords}  ${file_extension}
  ${files}  Find Files With Content Containing
Keywords  ${location}  ${keywords}  ${file_extension}
  ${most_recent_file}  Determine Most Recently Modified  ${files}
  [Return]  ${most_recent_file}

# See 'Find Files With Content Containing Keywords' and
# 'Determine Most Recently Modified' keywords above.
# Uses the results of afore mentioned keyword as input for the latter.
# Further filters the results by only considering files that fit the name
requirement.
# @input:   ${location}             directory path to search files from
#           ${keywords}             a list of keywords to be pattern matched within
file contents
#           ${name}                 expected file name without an extension
#           ${file_extension}       extension to determine what files should be
considered while searching,
#                                   example: js html
# @returns:   ${most_recent_file}   file path of the most recently modified file
Find Most Recent File Based On Keywords And Name
  [Arguments]  ${location}  ${keywords}  ${name}  ${file_extension}
  ${all_files}  Find Files With Content Containing
Keywords  ${location}  ${keywords}  ${file_extension}
  @{files}  Create List
  FOR  ${file}  IN  @{all_files}
    ${directory}  ${file_name}  Split String From Right  ${file}  /  1
    ${file_name}  Convert To Lower Case  ${file_name}
    IF  '${file_name}' == '${name}.${file_extension}'
      Append To List  ${files}  ${file}
```

(to be continued)

Code F1 (continues)

```
    END
  END
  ${most_recent_file}  Determine Most Recently Modified  ${files}
  [Return]  ${most_recent_file}

# The keyword takes parent element:child element key:value pairs as input and goes
through
# each key to verify that child elements are as dictated by web standards.
# If at any point a child element is found to be incorrectly under a certain parent
element,
# the keyword determines FAIL status and acts as a test case would when keyword
fails.
# This keyword can be used together with 'Run Keyword And Return Status' to catch
the status
# as a variable without failing a task / test.
# @input:   ${parent_child_dict}    a dictionary where keys are parent elements and
values the child elements found under that parent
# @output:  PASS / FAIL status
Verify Table Element Hierarchy
  [Arguments]  ${parent_child_dict}
  FOR  ${relations}  IN  &{parent_child_dict}
    IF  '${relations}[0]' == '<table>'
      Should Not Contain  ${relations}[1]  <table>
    ELSE IF  '${relations}[0]' == '<caption>'
      Should Not Contain
Any  ${relations}[1]  <table>  <caption>  <colgroup>  <thead>  <tbody>  <tr>  <th>
 <td>  <tfoot>
    ELSE IF  '${relations}[0]' == '<colgroup>'
      Should Not Contain
Any  ${relations}[1]  <table>  <caption>  <colgroup>  <thead>  <tbody>  <tr>  <th>
 <td>  <tfoot>
    ELSE IF  '${relations}[0]' == '<thead>'
      Should Not Contain
Any  ${relations}[1]  <table>  <caption>  <colgroup>  <thead>  <tbody>  <td>  <th>
 <tfoot>
      Should Contain  ${relations}[1]  <tr>
    ELSE IF  '${relations}[0]' == '<tbody>'
      Should Not Contain
Any  ${relations}[1]  <table>  <caption>  <colgroup>  <thead>  <tbody>  <tfoot>
    ELSE IF  '${relations}[0]' == '<tr>'
      Should Not Contain
Any  ${relations}[1]  <table>  <caption>  <colgroup>  <thead>  <tbody>
    ELSE IF  '${relations}[0]' == '<th>'
      Should Not Contain
Any  ${relations}[1]  <table>  <caption>  <colgroup>  <thead>  <tbody>  <tr>
    ELSE IF  '${relations}[0]' == '<td>'
      Should Not Contain
Any  ${relations}[1]  <table>  <caption>  <colgroup>  <thead>  <tbody>  <tr>
    ELSE IF  '${relations}[0]' == '<tfoot>'
      Should Not Contain
Any  ${relations}[1]  <table>  <caption>  <colgroup>  <thead>  <tbody>  <td>  <th>
 <tfoot>
      Should Contain  ${relations}[1]  <tr>
```

(to be continued)

Code F1 (continues)

```
    ELSE
      Should Not Contain
Any  ${relations}[1]  <table>  <caption>  <colgroup>  <thead>  <tbody>  <tr>  <th>
 <td>  <tfoot>
    END
  END

# Uses regular expression to verify that a table element does not contain other
than hierarchically
# correct child elements. The keyword can take a list of found table elements. This
keyword does not produce
# a PASS / FAIL status but rather the number of verifiably correct tables.
# @input:     ${table_elements}       a list of table elements
# @output:    ${proper_table_amount}  number of correctly constructed table
elements
Verify Table Elements
  [Arguments]  ${table_elements}
  ${table_elem_regex}  Set
Variable  table|caption|colgroup|thead|tbody|tr|th|td|tfoot
  ${proper_table_amount}  Set Variable  ${0}
  FOR  ${table}  IN  @{table_elements}
    @{elements}  Split String  ${table}  ${SPACE}
    FOR  ${element}  IN  @{elements}
      ${is_table_element}  Run Keyword And Return Status
      ...  Should Match Regexp  ${element}  ${table_elem_regex}
      IF  '${is_table_element}' != 'PASS'
        Remove Values From List  ${elements}  ${element}
      END
    END
    &{relations}  Parent Child Relations From List  ${elements}
    ${verification_result}  Run Keyword And Return Status
    ...  Verify Table Element Hierarchy  ${relations}
    IF  ${verification_result}
      ${proper_table_amount}  Evaluate  ${proper_table_amount} + 1
    END
  END
  [Return]  ${proper_table_amount}
```

Appendix G: Custom library

Code G1: MyLibrary.py.

```
# Library to support various parsing tasks.
# Each method is usable as a keyword when imported in a Robot file.
# Robot framework logic:
# - Method name translates into a keyword when underscores are left out.
#   For example: def prepare_soup() in this library ==> 'Prepare Soup' in robot
file
# - When used as robot keywords after library is imported,
#   keywords require input values unless a default value is declared for an input
variable in this library
# - Return in method means the keyword will return content when used
#
# Do note that whenever html parser is used on source content, a lot of syntactic
errors present in the
# raw source code will be corrected by the html parses in the process. If the
purpose is to receive
# raw content, regular expressions are recommended.

# coding=utf-8
import os
import re
import fnmatch
import shutil
import errno
from bs4 import BeautifulSoup, Doctype

class MyLibrary:
    # While this will be available as a keyword when imported,
    # this is not meant to be used as such.
    # Used as a support method within this class to parse content
    # with Beautiful Soup.
    def prepare_soup(self, src_file, parser):
        '''
        :param src_file: file path
        :param parser: parser to use for parsing content from src_file; current
prototype includes support for html5lib and lxml
        '''
        with open(src_file) as src:
            soup = BeautifulSoup(src, parser)
        return soup

    # Searches a given path contents for files matching a given file extension.
    # The path given as input is a starting point, will also search from all sub-
directories.
    def search_file_with_extension(self, path, file_extension):
        '''
        Returns all file locations found with specified extension
        from the given path.
        :param path: Path that will be searched within. Must end with '/'.
        :param file_extension: The extension (type) of the file being searched for.
            Must in form without '.' i.e. 'html' or 'css',
```

(to be continued)

Code G1 (continues)

```
            NOT '.html' or '.css'.
        '''
        file_extension = str.lower('*.' + file_extension)
        file_locations = []
        file_location = ''
        for root, dirs, files in os.walk(path, topdown=True):
            for name in files:
                if fnmatch.fnmatch(str.lower(name), file_extension):
                    file_location = os.path.join(root, name)
                    file_locations.append(file_location)
        return file_locations

    # Parses the source contents for all html elements containing the attribute
'id'.
    # Found elements form a bs4 result set (a list) of bs4 tags.
    def find_all_ids_from_html(self, src, parser='html5lib'):
        '''
        :param src: file path
        :param parser: html5lib by default; if provided as input,
                       make sure another parser library is supported.
        '''
        soup = self.prepare_soup(src, parser)
        found_ids = soup.find_all(id=True)
        return found_ids

    # Searches for specific element from the source contents.
    # Search ends as soon as the very first matching instance is found.
    def find_element_from_html(self, src, elem, parser='html5lib'):
        '''
        :param src: file path
        :param parser: html5lib by default; if provided as input,
                       make sure another parser library is supported.
        '''
        soup = self.prepare_soup(src, parser)
        found_element = soup.find(elem)
        return found_element

    # See find_element_from_html(); this one does the same but
    # returns all found elements as a list of bs4 tags.
    def find_elements_from_html(self, src, elem, parser='html5lib'):
        '''
        :param src: file path
        :param parser: html5lib by default; if provided as input,
                       make sure another parser library is supported.
        '''
        soup = self.prepare_soup(src, parser)
        found_elements = soup.find_all(elem)
        return found_elements

    # Finds elements that have a specific attribute. Attribute values are of no
concern.
    # Returns all matching elements as tags.
    def find_elements_with_attribute(self, src, elem_tag, attr, parser='html5lib'):
```

(to be continued)

Code G1 (continues)

```
        '''
        :param src: file path
        :param elem_tag: html element tag to search for, i.e. a, table, li, ul...
        :param attr: attribute the elements should contain, i.e. name, id, class...
        :param parser: html5lib by default; if provided as input,
                       make sure another parser library is supported.
        '''
        soup = self.prepare_soup(src, parser)
        found_elements = soup.find_all(elem_tag, {attr:True})
        return found_elements

    # Searches for and lists all child elements of a given bs4 tag.
    def find_immediate_child_elements(self, src):
        '''
        :param src: a bs4 html element tag
        '''
        children = [child for child in src if child.name != None]
        return children

    # Searches for elements that have a given class.
    def find_elements_by_class(self, src, elem, cls, parser='html5lib'):
        '''
        :param src: file path
        :param elem: html element tag to look for
        :param cls: class that element should contain
        '''
        soup = self.prepare_soup(src, parser)
        elems = soup.select(f'{elem}.{cls}')
        return elems

    # Finds elements from source contents. Does not fix the raw content as html
parsers do
    # due to not using Beautiful Soup and html parsers.
    # Returns a multi-line string containing child elements from all the found
matches.
    def find_elements_from_raw_source(self, src, elem):
        '''
        :param src: file path
        :param elem: element to look for; case is ignored
        '''
        element_results = []
        with open(src) as src_open:
            source_raw = src_open.read()
        regex = rf'(<{elem}.*?>.*?<\/{elem}>)'
        elems = re.findall(regex, source_raw, re.IGNORECASE | re.DOTALL)
        regex_clean_elements = r'<([^\/]\s*[a-zA-Z]*).*?>'
        regex_clean_content = r'>(.*?)<'
        for elem in elems:
            elem = re.sub(regex_clean_elements, r'<\1>', elem, flags=re.IGNORECASE
            | re.DOTALL)
            elem = re.sub(r'<\s*([aA]).*?>', r'<\1>', elem, flags=re.IGNORECASE
            | re.DOTALL)
            elem = re.sub(regex_clean_content, '> <', elem, flags=re.IGNORECASE
            | re.DOTALL)
```

(to be continued)

Code G1 (continues)

```
            element_results.append(elem)
        return element_results

    # Expects a list of elements contained within html ul/ol/menu element.
    # Forms a dictionary based on input list contents so that
    # each key in the dictionary has child elements as values.
    def parent_child_relations_from_list(self, str_list):
        '''
        :param str_list:    a list of elements contained within a single html ul /
                            ol / menu element;
                            elements must be in string format.
                            A proper input can be got from
                            find_elements_from_raw_source(),
                            for example.
        '''
        length = len(str_list)
        parent_child_dict = {}
        met_parents = []
        for i in range(length):
            if str_list[i][1] != '/':
                if i-1 >= 0:
                    if met_parents[0] in parent_child_dict:
                        parent_child_dict[met_parents[0]].append(str_list[i])
                    else:
                        parent_child_dict[met_parents[0]] = [str_list[i]]
                met_parents.insert(0, str_list[i])
            else:
                met_parents.pop(0)
        return parent_child_dict

    # Copies source directory contents recursively to destination.
    def copy_directory_contents(self, src, dst):
        '''
        :param src:    source directory path
        :param dst:    destination path
        '''
        try:
            if os.path.exists(dst):
                shutil.rmtree(dst)
            shutil.copytree(src, dst)
        except OSError as exc:
            if exc.errno in (errno.ENOTDIR, errno.EINVAL):
                shutil.copy(src, dst)
            else:
                raise
```