UNIVERSITY OF VAASA

SCHOOL OF TECHNOLOGY AND INNOVATIONS

AUTOMATION TECHNOLOGY

Antti Hannonen

AUTOMATED TESTING FOR SOFTWARE PROCESS AUTOMATION

Master´s thesis for the degree of Master of Science in Technology submitted for inspection, Vaasa, 15 April 2020.

Supervisor	Jarmo Alander
Instructor	Hannes Hudd

ACKNOWLEDGMENTS

I would like to extend thanks to everyone involved in helping me with this thesis. From Wärtsilä, where this work was carried out, I would like to thank my instructor Hannes Hudd and my co-workers for their continued support and feedback.

Additionally, I would like to thank Jarmo Alander, my supervisor for this thesis, for his patience in guiding me through the writing process.

Vaasa, 15.04.2020

Antti Hannonen

TABLE OF CONTENTS

SYMBOLS AND ABBREVIATIONS

API	Application Programming Interface
COE	Center of Excellence
CD	Continous Deployment
CI	Continuous Integration
DSR	Design Science Research
GUI	Graphical User Interface
IT	Information Technology
IPA	Intelligent Process Automation
MVP	Minimum Viable Product
OCR	Optical Character Recognition
PDF	Portable Document Format
QA	Quality Assurance
RPA	Robotic Process Automation
SUT	System Under Test
UUT	Unit Under Test
UVT	User Validation Testing
VDI	Virtual Desktop Interface
WF	Workflow Foundation

VAASAN YLIOPISTO
Tekniikan ja innovaatiojohtamisen yksikkö

Tekijä:	Antti Hannonen	
Diplomityön nimi:	Ohjelmistoautomaation automaattinen testaus	
Valvojan nimi:	Professori Jarmo Alander	
Ohjaajan nimi:	DI Hannes Hudd	
Tutkinto:	Diplomi-insinööri	
Oppiaine:	Automaatiotekniikka	
Opintojen aloitusvuosi:	2013	
Diplomityön valmistumisvuosi:	2020	**Sivumäärä:** 76

TIIVISTELMÄ

Ohjelmistoautomaatio on nopea tapa automatisoida liiketoimintaprosessien rutiineja. Tapausyrityksessä automaation luovat ohjelmistonkehittäjien sijasta liiketoiminnan asiantuntijat. Käyttämällä alkukehittäjiä, joilla on vähäisesti kokemusta ohjelmistokehityksestä, saadaan nopeita ratkaisuja, mutta samalla yrityksellä on huolia laadusta. Laatua voidaan mitata testaamalla automaatioratkaisuja laajasti, mutta tähän menee huomattavasti aikaa. Tämän tutkielman tarkoituksena on testiautomaatiota käyttämällä nostaa kehitysprosessin laatua ilman että työmäärä kasvaa merkittävästi.

Tutkimus suoritettiin osana tapausyrityksen ohjelmistorobotiikkaprojektia. Tutkielmassa luotiin prosessi, jossa automaattisesti testataan ohjelmistoautomaatioprosesseja. Testaus todettiin tutkimuksessa mahdolliseksi mutta käytännössä haasteelliseksi. Testauksessa ilmeni useita ongelmia, mutta muutamat ratkaisut kuten regressio- ja integraatiotestaus todettiin kuitenkin hyödyllisiksi. Lähestymistavan hyödyiksi todettiin laadun jäljitettävyyden, kehittäjien itseluottamuksen ja kehitysnopeuden kasvu. Lisäksi testiautomaatio mahdollistaa nykyaikaisten ketterien menetelmien kuten jatkuvan integraation käytön. Jatkuvan integraation käyttömahdollisuus demonstroitiin uudistetulla työtavalla.

AVAINSANAT: ohjelmistoautomaatio, vähäkoodinen kehitys, optinen kirjaimentunnistus, testiautomaatio, jatkuva integraatio

UNIVERSITY OF VAASA
School of Technology and Innovations

Author:	Antti Hannonen
Topic of the Thesis:	Automatic Testing of Software Process Automation
Supervisor:	Professor Jarmo Alander
Instructor:	M. Sc. Tech Hannes Hudd
Degree:	Master of Science in Technology
Major of Subject:	Automation Technology
Year of Entering the University:	2013
Year of Completing the Thesis:	2020 **Pages:** 76

ABSTRACT

Robotic Process Automation is a way of automatizing business processes within short timespans. At the case company the initial automation step is implemented by business experts, rather than software developers. Using developers with limited software engineering experience allows for high speed but raises concerns of automation quality. One way to reduce these concerns is extensive testing, which takes up much time for integration developers. The aim of this thesis is to increase the quality of the development process, while minimizing impact on development time through test automation.

The research is carried out as a part of the Robotic Process Automation project at the case company. The artifact produced by this thesis is a process for automatically testing software automation products. Automated testing of software automation solutions was found to be technically feasible, but difficult. Robotic process automation provides several novel challenges for test automation, but certain uses such as regression and integration testing are still possible. Benefits of the chosen approach are traceability for quality, developer confidence and potentially increased development speed. In addition, test automation facilitates the adoption of agile software development methods, such as continuous integration and deployment. The usage of continuous integration in relation to Robotic Process Automation was demonstrated via a newly developed workflow.

KEYWORDS: robotic process automation, low-code development, optical character recognition, test automation, continuous integration

1 INTRODUCTION

Robotic Process Automation (RPA), is a method of software automation aimed at executing repetitive processes normally carried out by humans. It is similar to Information Technology Process Automation (ITPA) or Run Book Automation (RBA) (Fung, 2014). Despite its name, which suggests robotic arms and conveyor lines, in RPA the “robot” is a program mimicking the actions of a user on a desktop computer, akin to an older screen scraping macro. RPA processes are often used for automating data entry, bridging data between legacy software projects and retrieving and filtering data for reporting. RPA development is essentially a simplified form of software development, with a focus on interacting with existing programs rather than creating new ones.

A practical example of an RPA process is automating data entry. The process starts with the robot receiving an email message containing data, which is attached as a Portable Document Format (PDF) document. The function of the bot is to read the email and to add the data it contains to a database through a web form. Reading the input is achieved via a screen scraping feature in the RPA software that the developer uses to indicate relevant parts of the PDF document containing the data to be entered. Once the data is scraped into memory, the robot opens the web form and inputs the data into fields previously indicated by the developer. The process is created using low-code functionality, in which the developer acts out their normal workflow step-by-step. This workflow can then be repeated by the robot in the future, if the structure of the input data and the web form remain unchanged. Even though the new process decreases the manual workload on one side, it poses challenges on the technical side. Over time, the process needs updates, which in turn need to be tested. This need for additional work poses another opportunity for automation, which is further explored in this thesis.

Companies creating RPA software focus on creating low code platforms that can be interacted with by people with no or minimal previous software engineering experience. Characterizing RPA as physical robots is currently quite standard, due to the similarity of process design when automating physical and software tasks (Baranauskas, 2018). By utilizing RPA, organizations aim to develop quick solutions for business specific problems, without involving expensive solutions through normal software development with long development times (Penttinen, Asatiani & Kasslin, 2018).

A key element of RPA development at Wärtsilä is the use of citizen developers. These developers are experts in their own fields, be it accounting, finance or engineering, without necessarily possessing software engineering experience. Typically, these people already have some experience in software automation through writing simple scripts, or macros, to ease their own workloads. Practically, this is often experienced via software such as Excel. RPA software is designed to provide similar scripting functionalities, while easing integration to other systems. The developers’ expertise in their own fields allow them to better build automation solutions that suit their needs exactly, while the usage of a low-code platform allows for quick adoption by people without programming experience. However, as the automation is created by people with no or minimal software engineering skills, there is a need for a quality assurance process carried out by experienced developers. One of the goals of this way of working is to assure the quality of the citizen developers’ work while having as few software engineers as possible. Test automation is seen as a way of allowing the software engineers to work more effectively with RPA.

Automatic testing is seen as a way of increasing software quality, especially when it is used to facilitate continuous integration of code (Hilton, Nelson, Dig, Tunnell, & Marinov, 2016). Testing can be focused on the functionality of a module or the integration of the module to other modules and the execution environment (Baresi & Pezze, 2006). Even though test automation as a topic is well researched, it is not yet explored when it comes to RPA. RPA brings several novel challenges to testing, due to decreased control over the runtime environment and long execution times caused by imitating a typical user. An additional challenge is the lack of dedicated testing tools.

1.1 Research motivation

The quality assurance process at the case company is currently a mostly manual one. Bugs and other issues with the execution environment will either be discovered through executing tests manually in User Validation Testing (UVT), or in production runs. Testing is carried out by asserting the functionality of a module and executing it with test arguments. Increased testing of the runtime environment and the products should in theory result in earlier discovery of problems. However, time for testing activities is limited, as there is an increasing amount of MVP (Minimum Viable Product) stage RPAs waiting for quality assurance and the availability of experienced developers for this stage is limited. Automatically running and reporting test sets is proposed as an alternative solution.

While test automation is a heavily researched topic (Rafi, Moses, Petersen & Mäntylä, 2012) (Wiklund, Eldh, Sundmark & Lundqvist, 2015), it is not yet explored with respect to RPA. A couple of factors differentiate testing RPA from typical software testing. First, the tests are long lasting, due to their interaction with Graphical User Interfaces (GUI), leading to long execution times. While this is typical for testing GUI based programs, the difference is that in RPA, most of the tested components interact with the GUI, unlike in classical programs where most of the code is testable outside of the GUI. Second, since the strengths of RPA are its integration with system resources and its resistance to slight changes in GUI elements, less control is naturally imposed on the production environment, compared to classical software development. While this approach allows for quick adoption of new programs to the production environment, it introduces a risk of breakage, which can be alleviated by testing. The third difference to a classic software project is that the RPA codebase is split into tens of small projects, each of which executes independently of each other. This may lead to problems when creating automation, due to the increasing overhead cost of frequently creating new projects. On the other hand, a benefit of this is that one broken build does not necessarily indicate a breakage in the entire codebase. Exceptions to this are components that are shared between projects, which carry out routine tasks such as sending email and closing applications. One of these components breaking will probably lead to several separate projects failing and as such they should be treated differently.

1.1 Research questions

The aim of this thesis is to improve the testing of RPA solutions at a case company by implementing test automation. The output will be a test automation system, that facilitates the use of Continuous Integration and Delivery. As such, the research question takes the form: "*How automated testing can be applied to robotic process automation?*" The secondary research question takes the form: *"How test automation can facilitate continuous integration?"*

1.2 Thesis structure

The thesis is structured as follows. Chapter 2 consists of a literature review of works on robotic process automation, test automation, continuous integration and testing. Chapter 3 contains a plan for carrying out the research. Chapter 4 describes the current state of testing at the case company. Chapter 5 contains the documentation of the implementation of the test automation system. In Chapter 6 the effectiveness of the implemented system is evaluated. Finally, chapter 7 consists of conclusions on the project.

2 RESEARCH PLAN

This chapter outlines the plan for the research, including the primary method to be used.

2.1 Research plan

The research is carried out as a part of a project to adopt modern software development methodology, i.e. continuous integration and deployment (CICD), to RPA at the case company. The research will be carried out as a form of design science research, where test automation is applied to the company's RPA project. The output of the project will be an environment able to automatically execute tests and guidelines/templates for creating the tests in the first place. Design science research was chosen since the output of the research is an artifact, the test automation system for RPAs.

The first stage of the research is to map out requirements for the test system. This is done by analyzing results of literature on the subject and informal workshops with developers. The second stage is to create an implementation, while consulting with developers. Third, the implementation is tested by using it while finishing a specific under development RPA solution. The fourth and last step is to evaluate the end results effectiveness at increasing quality.

2.2 Design science research

Design science research (DSR) is used as a method of creating a product while analyzing its' function. DSR is considered research, when the product is new knowledge of applying new solutions or solving new problems, unlike in routine design, where generally known solutions are applied to known problems (Vaishnavi, Kuechler & Petter, 2017).

The process of DSR is similar to modern design principles, in that it is an iterative process where phases can be revisited from other phases (Vaishnavi et al, 2017). The problem is first analyzed to identify its characteristics. Then, a solution is suggested by defining its objectives. A design of the solution is then developed, which produces the artifact for analysis. The usage of the artifact is demonstrated by solving the defined problem. The performance of the artifact is then evaluated. Different methods of evaluation are measuring the system availability or client feedback. Conclusions are finally drawn based on the evaluation. This process flow can be iterated, creating continual discovery and improvement of the design. A chart outlining the DSR process is presented in Figure 1. (Peffers, Tuunanen & Rothenberger, 2007.)

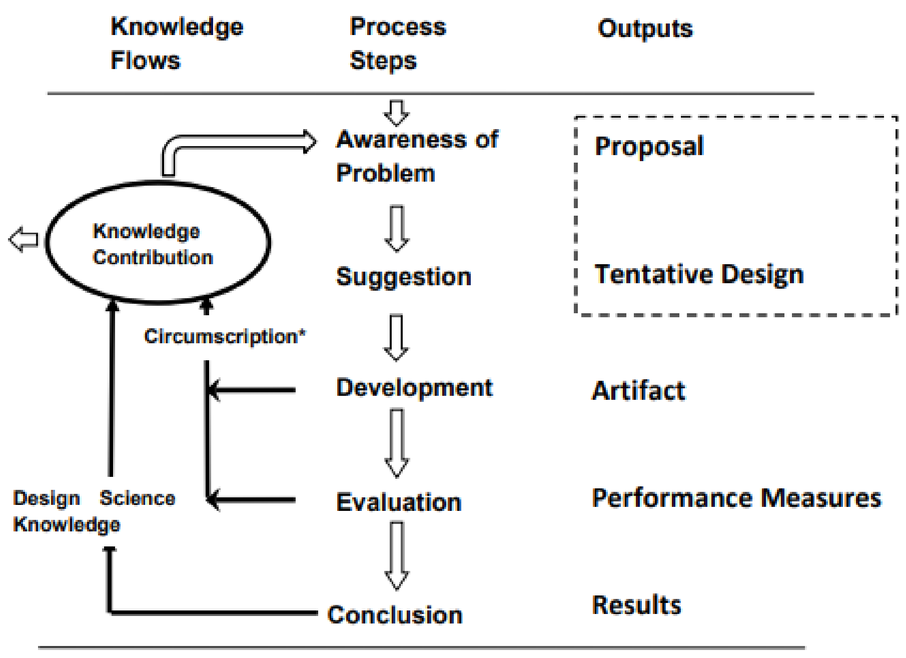


Figure 1.	Process flow of DSR (Vaishnavi et al, 2017). Iterations are key to the process, as at each stage of the design the process is analyzed and improved upon. The key outputs are the artifact, which is the design being created, and the knowledge contribution gained from analyzing the process and the artifact.

Evaluation measures relevant to the case project are error rate in production and number of errors caught by testing. The primary goal of the system is to avoid breakage in production, for which both error rates in production and testing provide a measure. Frequently testing the pilot product might reveal underlying problems with infrastructure, repairing some issues with other projects even though the testing is not directed at them, thus decreasing error rate in production. Because of this, the measurement for comparison should be taken at the time before beginning the pilot project. However, for errors caught in testing, there is no real comparison point as those bugs are not currently logged. Because this metric cannot be compared numerically, a log will be made of errors caught during development of the pilot project and the errors will be evaluated as to their likelihood of causing problems without the automated testing system. In addition, since projects significantly vary in scope, the measurement of a single project should not be considered entirely accurate. As such the measurements made should not be considered conclusive and the system evaluation will have to be ongoing with development of new projects.

While there are several possible outputs of DSR, the ones most relevant to this project are frameworks, methods and instantiations. Frameworks are guides or guidelines on how to achieve a design. Methods are a step by step solution to achieving the things outlined in the framework. Finally, instantiations are an actual implementation of the design in a practical environment. (Vaishnavi et al, 2017.)

3 LITERATURE REVIEW

This chapter contains a review of available literature on the topics being covered. It is split in three parts: RPA, test automation and continuous integration. The section about RPA contains information on RPA companies and motivations behind using RPA. The test automation section contains review of distinct types of automated tests and their benefits. Finally, the continuous integration contains information on the methods and benefits of continuous integration.

3.1 Robotic Process Automation

Robotic Process Automation (RPA) is a lightweight IT method of automating repetitive office work, similar to software macros. The term RPA was originally coined by the company Blue Prism and is an amalgamation of screen scraping, Optical Character Recognition (OCR) and low-code software development techniques (Issac, Muni & Desai, 2018). The very first RPA solution is difficult to pin down as certain software, e.g. AutoHotkey, could be described as an RPA solution if developed today. Screen scraping is a method of retrieving data displayed by software on a computer screen. OCR is a method for a computer to analyze data written in images. Low code software development is an easier to learn version of normal software development methods, where writing code is replaced by dragging and dropping ready-made components to a screen.

Macros and screen scrapers share many similarities with RPA and RPA software also has support for both methods. In addition to accessing API functions e.g. to send e-mail and modify tables in Excel, modern RPA software can in addition access the base structure of Graphical User Interfaces (GUI), thus being more resistant to slight changes in software versions. Optical Character Recognition methods also help to bridge the gap in the case of legacy software or remote desktop scenarios. The different modes of interaction between RPA and other software are depicted in Figure 2.

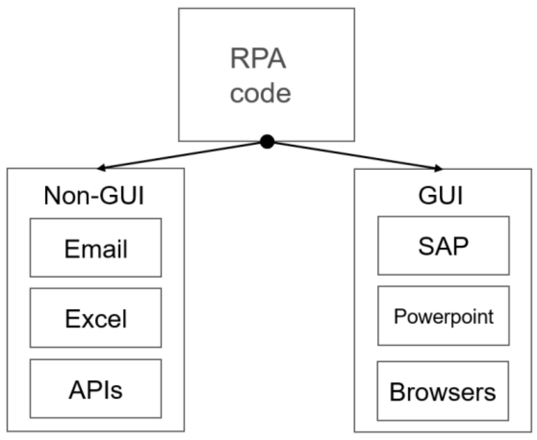


Figure 2. Depiction of RPA interactions. Actions happen either in the background, or against the GUI. Background actions are faster but take more time to develop. GUI level actions are fast to develop, but slower to run.

These methods allow RPA to be more resistant to change than traditional methods of software automation, resulting in more easily maintained solutions. In addition, RPA providers increasingly seek to provide a user-friendly interface, which requires little to no programming experience, so that RPAs can be created by experts in their own fields, rather than software developers.

3.1.1 Current state of RPA

There are several companies currently providing RPA software. The largest companies in the field are Automation Anywhere, Blue Prism and UiPath. They provide products that are very similar in scope, with slight variations in focus between them. There are also several smaller providers such as Kryon and Pegasystems, with significantly smaller market share. These smaller companies are often specialized into some specific sector, such as finance, or are especially focused on achieving Intelligent Process Automation (IPA) solutions via machine learning. Many processes that were previously targets for outsourcing are now a target for automation via RPA (Baranauskas, 2018). Several case studies have been made on RPA implementations in businesses, resulting in high return on investment. Examples of organizations that utilize RPA are the North American Space Association, Hewlett & Packard, Google and AutoDesk (Uipath, 2018).

While RPA is a rapidly growing field, it is best applied in certain kinds of processes. Like processes suitable for physical automation, these processes are characterized by high volume, predictable load and monotonous nature. According to Baranauskas, RPA is best characterized as a physical process, due to the similarity in nature and the recognizability of physical process automation robots. In addition, determining the nature and amount of human interaction with the software robots is a problem for designing them, just as it is with physical robots. (Baranauskas, 2018.)

Studies on successful RPA implementation also remark on the different structure of RPA projects to general software projects. Successful implementations of RPA are business led with IT assistance, instead of the other way around as is the case with normal software development (Stople, Steinsund, Iden & Bygstad, 2017). Being driven by business needs, the resulting products better match actual company processes and through this brings more value to the company. This development model however brings some concerns about security from the IT side. One approach to solving this problem is setting up an RPA Center of Excellence (COE), which maintains the business led development model (Anagnoste, 2018). This model adds some dedicated professionals to the process to ensure quality and security compliance.

3.1.2 Impact of RPA

While RPA is a relatively new approach to software automation, several case studies have already been carried out on its impact on business processes in several sectors, for example telecommunications and finance. Similar to backend system automation, RPA can increase speed of processes and reduces costs (Aguirre & Rodriguez, 2017).

Even though the field is still relatively new, the effectiveness of utilizing RPA has been subjected to some case studies. In one such study, carried out at a telecommunication company, over a hundred robots were deployed reducing the time taken for some processes from days to minutes. (Lacity, Willcocks & Craig, 2015.) Telecommunication is not the only sector where RPA has been utilized. A case study was carried out at an outsourcing company to automate payment receipt generation by Aguirre et al. However, the benefits observed, are very similar. Increase in process speed, error reduction and improvement in productivity were reported (Aguirre & Rodriguez, 2017). Similar results have been reported in the banking sector as well at a Norwegian bank by Stople et al. (2017).

3.1.3 RPA compared to IT integration

IT processes can be split into two categories, heavy- and lightweight. Heavyweight IT refers to the classic methods of solving software automation problems by creating heavily integrated new products and creating new infrastructure. Lightweight IT however refers to methods such as RPA or macros which work on top of existing IT solutions. (Bygstad, 2017.)

While the software automation solutions created with RPA tools can be achieved in other ways through heavyweight IT, RPA provides some unique advantages. There are several considerations when choosing between a heavyweight IT solution and RPA. These factors include the number of systems, volume of transactions, stability and availability of interfaces, and the time to market of the solution. (Penttinen, Asatiani & Kasslin, 2018.)

Problems when adopting a heavyweight IT approach can be the lack of interfaces, especially with legacy systems. In addition, where interfacing is possible this might be on the database level. While this is more efficient, long term planning, approvals by the company and significant amounts of developer time are required. (Penttinen, Asatiani & Kasslin, 2018.)

Table 1. Choosing between RPA and an IT integration solution. A proper integration solution will beat RPA in speed and is such more applicable to very high-volume transactions. RPA solutions on the other hand can be utilized even if no back-end interface is exposed for integration. The downside of this, is that the RPA solution will be vulnerable to updates to the front end. A normal IT integration will then be affected by back-end changes. Due to the higher investment, a heavyweight IT solution is expected to last longer, while an RPA can be used on a temporary timescale. An RPA process is fundamentally easier to develop and requires a smaller team to go to production. (Penttinen, Asatiani & Kasslin, 2018)

Feature	RPA	IT integration
Transaction volume	High	Very high
API availability	Not needed, but can be useful	Required, can often be developed
System update frequency	Resilient to back-end updates	Resilient to GUI updates
Process life-expectancy	Short term	Long term
Expertise requirement	Mostly business process expert	Both business process and software expertise.

If the number of systems used for the process is large, having working interfaces between them is less likely, making RPA more viable (Fung, 2014). On the other hand, if the volume of the processes is very high, the efficiency of a backend process will be more relevant, making RPA a less viable choice.

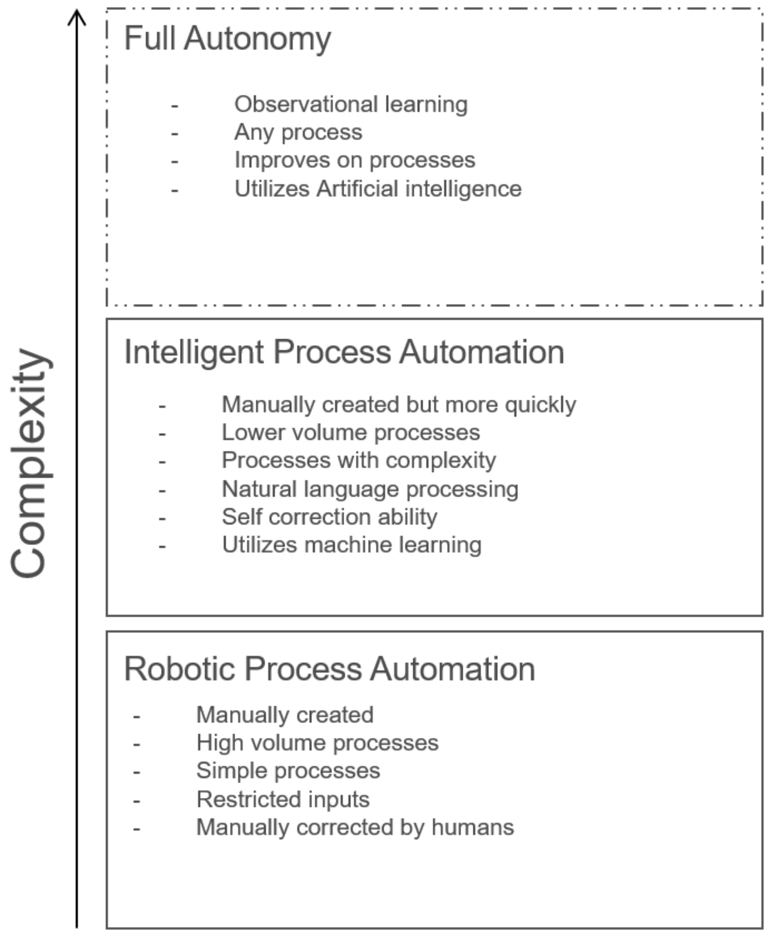
Stability of interfaces is also a factor. If the backend changes often, an RPA solution interfacing only with the front end will be more viable. On the other hand, a stable backend with a variable frontend will provide the reverse result, with backend system automation being more viable. The problem with stability is especially clear in the case of light integration with the IT function, where changes may appear without forewarning thus breaking automations (Stople et al, 2017). Using backend IT also results in a more permanent, difficult to change, system than its' lightweight counterpart. (Fung, 2014.)

Finally, time to market for RPA is much lower than for backend IT (Penttinen et al, 2018). This is since pre-existent infrastructure is being used, leading to less need for building or purchasing systems. In addition, since the RPA software interfaces with existing systems similarly to human users, the need for approvals is reduced. On the other hand, if time to market is not critical but the process will be long standing, the benefits of the increased efficiency of a heavyweight IT process become more apparent.

3.1.4 Future of RPA

While RPA is a good fit for simple problems with high volume, the value of the software quickly drops as the processes become more complicated and of smaller volume. Functions that are more difficult to automate include those that require unstructured communication with people (Anagnoste, 2018). Some RPA providers are actively working on creating so called Intelligent Process Automation (IPA), which could bring down development time, making it more worthwhile to automate complicated or low volume processes.

Software process automation is a tiered process, with increasing complexity of processes. This tiered structure is visualized in Figure 3. RPA is on the bottom tier, capable of automating high volume monotonous processes. RPA has been gaining popularity rapidly in the past few years and is becoming widely utilized. IPA extends on RPA, making RPA creation easier via methods such as natural language processing. Utilizing IPA allows for automating lower volume processes with a similar amount of return on investment. Some attempts on utilizing IPA already exist and some companies, for example XPertrule claim to offer IPA solutions. The highest tier of technology in this sector is independent automation. An independent automation would be capable of creating its' own processes either by watching user actions or by studying relationships between input and output data. However, currently this tier of automation is not technologically feasible. Further developments in this area are heavily reliant on the progress of Artificial Intelligence (AI) systems.



Complexity
Full Autonomy
- Observational learning
- Any process
- Improves on processes
- Utilizes Artificial intelligence
Intelligent Process Automation
- Manually created but more quickly
- Lower volume processes
- Processes with complexity
- Natural language processing
- Self correction ability
- Utilizes machine learning
Robotic Process Automation
- Manually created
- High volume processes
- Simple processes
- Restricted inputs
- Manually corrected by humans

Figure 3. Tiers of RPA technology. Currently the technology is on the lowest tier, approaching the second. The second tier is dependent on the development of current machine learning methods. Full autonomy is still an undeveloped concept.

Some functionality to this end already exists in modern RPA software via natural language processing functionality. Currently the large RPA providers do not create these solutions in-house, opting instead to utilize software by other, large providers specializing in modern machine learning methods such as Microsoft and Google (UiPath, 2019).

3.2 Test automation

This chapter outlines different types of software testing and their suitability for automation. Test flakiness, the lack of long-term stability and reliability in a test, is also discussed. In addition, the impact of test automation both in terms of time saved and resources utilized is reviewed. Software testing methods can later be applied for use with RPAs. Understanding the different types of software testing should help with applying automation to them, as well.

3.2.1 Testing levels

Software testing is split into three separate levels which are from lowest to highest, unit, integration, and system level. The unit level is used for testing against the basic functionality of a program. The integration level is used for testing the integration of the units together and integration with outside components and environments may also be included. System level testing generally includes the end user and is the highest level of testing, including both the functionality of the program and user satisfaction. (Baresi, 2006.)

As the lowest level of testing, unit testing is used to make sure the individual components of software function correctly. These tests are generally quick to execute. A proper unit test is limited to testing a single functionality at a time. The unit level generally does not include GUI components, which makes these tests the fastest. (Garousi, 2018.)

Integration tests generally follow successful unit testing. In an integration test, the separate units described on the previous level are combined, and their functionality together is tested as subsets of the whole functional program. At this stage any outside dependencies, such as external libraries, may also be included. Additionally, the runtime environment may be considered. At this stage, GUI components may also already be included. (Baresi, 2006.)

System testing is considered the highest level of testing and it focuses on the behaviour of the system as a single entity. It is used to test for the compliance to specifications. This stage includes security, performance, recovery and compatibility testing. Performance testing is focused on the speed of execution and long-term reliability of software. These tests can be executed on varying levels, either testing the speed and reliability on the unit level or testing the flow of the entire program at the system testing level. System response time is easily quantifiable, so these types of tests are easily automated. Security testing can take the form of static code analysis methods on the unit level, looking for known security risks, which is also a task well suited for automation. On the system level penetration testing, where a security professional actively tries to find security risks in a software product, can also be carried out but is more difficult to automate. Since these tests are built top-down and are GUI focused, they become more expensive to automate due to difficulties generating user input and increased test times. Test flakiness is often also a problem with automation on this level. (Hooda et al, 2016)

In recovery testing, the software is subjected to unpredictable circumstances to test out its' resiliency. This can be done by using random inputs or mutation testing. Finally, compatibility testing is used to find out if applications on the system under test conflict. (Gaur et al 2016)

Acceptance testing can be considered a part of system testing. At this stage, the program is considered to function fully, however it still lacks validation from the end user. Acceptance testing consists of both initial verification of the functional program by the developers and the end stage of validation by the users of the software. This part is often split into alpha and beta testing (Baresi & Pezzè, 2014). An alpha test is done on software that is not yet feature complete and is aimed at feedback on the feature level, while beta testing precedes a final release and focuses on bug discovery (Gaur, Goyal, Choudhury & Sabitha, 2016). Acceptance testing is the most difficult part to automate, since it usually requires the presence of end users. This category also includes the usability testing, which focuses on intangibles such as user experience (Baresi & Pezzè, 2014).

3.2.2 Types of tests

The different types of testing are tools used for both verification and validation. Many of the types of tests can be applied on all levels of testing. Testing can be focused on functionality, performance or security (Hooda & Chhillar, 2015). In verification, implementation is compared to the specification, whereas in validation users' expectations are compared to system performance (Baresi & Pezzè, 2014). Both can be achieved automatically via testing; however, verification implies a more structured, mathematical way of working which lends itself to automation. Validation is more generally achieved via manual testing by end users or quality personnel. For a system to be verifiable, its properties need to be described quantitatively (Baresi & Pezzè, 2014).

Testing can be split into three categories based on their approach: Black, White, and Grey box testing. In Black box testing, the tester has no access to the source code and is instead forced to analyze the inputs and outputs. The main advantage of this method is that the tester and programmer are independent of each other. The problem with this approach is that the test specification must be exhaustive and even then, the lack of knowledge can lead to inefficiencies. This type of testing is usually applied on the system level. White box testing, which allows examining the source code, is done in addition to black box testing. This type of testing requires additional skills from the tester as a programmer but results in better test coverage. Grey box testing is a newer method testing, in which the tester is not given full access to the source of the SUT but is instead given information on the interface level. While this method requires less time than White box testing, it compromises in inheriting the lesser test coverage of Black box testing. Each of these methods is most applicable to different levels of software testing. (Gaur et al, 2016)

Functional testing generally precedes performance and security testing (Hooda et al, 2015). Activities in this category can be split into several levels: module, integration, system and acceptance testing. Module testing is usually done by developers locally, in isolation from other components. Integration testing is done at a later stage to monitor the behavior of software modules in concert, to avoid unexpected interactions between components. Integration tests can be created and executed by developers locally, or by quality assurance personnel (Hooda et al, 2016). These are generally verification-oriented tests. Module and integration testing raise confidence for a component's function in isolation from end-users through verification, but do not necessarily give information of their function in practice. (Baresi & Pezzè, 2014.)

3.2.3 Regression testing

Regression testing is a way of ensuring that recent changes do not break old features. This is done by frequently rerunning old tests and creating new tests to be run when fixing bugs. In a test, the functionality of the Unit Under Test (UUT) is checked by asserting relationships between input and output items. Generally functional, regression testing can also focus on security or performance. The common approach to regression testing is to execute the entire test suite, but this is a needlessly expensive approach since not all tests are affected by the changes made. To this end, several different approaches to test case selection exist. One method is to focus on tests that have failed frequently or fail often. Alternatively, the tests can be focused on areas not tested recently, or ones which test most effectively. It is also possible, albeit inefficient, to do test case selection manually. Regression testing is mostly focused on the unit and integration levels, as these are more easily automated than system level tests. (Baresi & Pezzè, 2014.)

An important requirement for selecting regression tests is to shorten the test time. To this end, one possible approach is basing the selection on file dependencies. If dependencies are known for a test, the test can be executed only when some of those dependencies change. While generating the dependencies in the first place is technically challenging, this approach provides clear benefits with respect to saving time in regression test suites. Gligoric's approach to dependencies is to monitor which files get accessed by a test. These files get automatically marked as a dependency for the UUT and changes to the UUT can then be used to trigger other affected modules. (Gligoric, Eloussi & Marinov, 2015.)

Hybrid methods can also be utilized. For example, a technique has been used at Google, where tests that are executed before being submitted to source control are chosen manually by a developer by submitting a change list. On the other hand, post-submit tests are automatically executed based on the change list and, similarly to Gligoric's work, dependency graphs that are automatically generated. (Elbaum, Rothermel, & Penix, 2014.)

Test selection systems can be measured in terms of safety and precision. A safe system selects all tests affected by changes. A precise system picks less tests unaffected by tests. For example, executing all tests is a safe but very imprecise approach. An effective automatic regression test system selection is both safer and more precise than the manual equivalent. In addition to saving developers from doing manual work, the quality of testing increases. Manual test selection results in unnecessary tests being run, while affected tests are often missed. (Gligoric, Eloussi & Marinov, 2015.)

3.2.4 Test design

Software tests, especially automated ones, require careful design. The tests need to validate system behavior, be maintainable and execute quickly enough not to slow down development work. Since test design is a well-researched topic, some common guidelines exist for development.

A test generally consists of retrieving or generating a pre-state if needed, executing the module under test, cleaning up the results and reporting results. This process is illustrated in Figure 4. For simple modules, test data can be saved as a part of the test instead of a separate file location. In the execution phase, the SUT is either run through in a monitored fashion. Test cleanup may not be necessary either, if the system state has not changed. The reporting phase should present results in such a way that the developer can assess the error. This includes both any logs created during the run and any errors created by asserting the functionality of the module. (Baresi & Pezzè, 2014.)

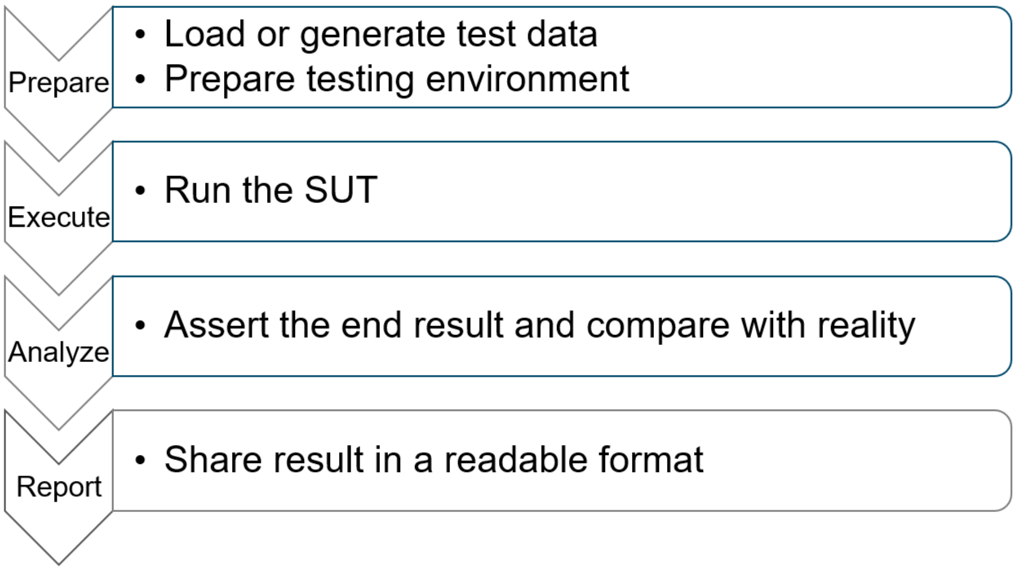

Prepare
• Load or generate test data
• Prepare testing environment
Execute
• Run the SUT
Analyze
• Assert the end result and compare with reality
Report
• Share result in a readable format

Figure 4. The test execution process. Test data is generated through code or read from storage, the test is executed, and the results are evaluated against conditions decided at design time. Finally, the results are shared with the developer.

On the unit testing level, one of the key aspects is speed. A slow test is difficult to execute continuously without disrupting the developer’s workflow. A unit test should also avoid outside dependencies. In the cases where these are unavoidable, their state should be strictly controlled. Redundant and eager tests should also be avoided. Redundant tests cover the same area as other tests, whereas an eager test attempts to test several system behaviors at once, making it difficult to maintain and the results difficult to interpret. (Garousi, Kucuk, & Felderer, 2018.)

On the integration level, the slowness of tests is more difficult to avoid, especially when GUI components are involved. Integration test suites will easily fill any time slot they are given, so test cases should be prioritized. In addition, outside dependencies become difficult to avoid, but their state should at least be recorded. Flaky tests, which fail even when the SUT has not been changed, also become more common with integration tests. Flaky tests are often simply called unreliable. Finally, a test should not leave behind any persistent data. (Garousi, Kucuk, & Felderer, 2018.)

Tests should be written to be reliable. A reliable test, upon failure will indicate a fault in the code. These tests can pass and fail without changes being made to the unit under test. The reliability rate is simply measured by re-executing a test and measuring how often the test failure corresponds to an actual fault. Flakiness often appears after the unit testing stage at the integration layer, when the code is subjected to a less controlled environment. However, since the integration phase is a common source of bugs a trade-off between test reliability and bug discovery has to be made. (Roseberry, 2016.)

Some practises are considered unfavourable on both levels. Test naming should be logically related to both the module under test and the test type, in order to make the test easily discoverable. More difficult to identify test smells include tautological tests, which follow the structure of the SUT very closely reimplementing the same functionality. Test quality can be improved via static code analysis on both levels. Finally, pre- and post-checks for system status should be implemented to both scan for errant data creation and persistent processes. (Garousi, Kucuk, & Felderer, 2018.)

Automated test design could be considered a method for creating quality tests every time. Several methods for this exist. Tests can be directly created from specification, even supporting some levels of natural language description. Alternatively, test generation can be based on static analysis of code structure. Test generation can result in pre-generated stubs to help developing proper tests, or even full test suites. Unfortunately, using an automatically generated test suite can make debugging test results more difficult, as the developer will be less sure if the test failure is due to a break in the test code or the program itself. (Gligoric, Negara & Legunsen, 2014**)**

However, flakiness can be avoided, since some testing patterns are more prone to flakiness than others. Using sleep or delay commands to wait for UI elements to appear should be avoided. In addition, test preparation is often a cause for flakiness and should be minimized. For example, instead of testing a web page by navigating to the right page via UI and then executing the test, the navigation should be done via a direct link to minimize chances for failure at this stage. Testing as much as possible at the unit level is also recommended. (Roseberry, 2016.)

In some cases, flakiness cannot be entirely removed from the test suite. Roseberry recommends that these tests should not be used as gating mechanisms for deployment. Since an unreliable test may succeed on a rerun, a developer working with such a test can simply rerun the test to get past a gating mechanism, wasting time. If unreliable tests are kept separate from reliable ones, they can be used for bug discovery without breaking builds and requiring re-execution. A flaky test may reflect a periodic problem with the infrastructure, that requires developer attention to prevent the same occasional errors from appearing in a production environment. (Roseberry, 2016.)

Test results also need to be reported back to the tester. For automated testing this can be done for example with JUnit. A JUnit results file is saved in XML format. It contains properties for several test suites, all of which can contain multiple tests. The suite can also include information on the total runtime of a test suite. Each test is named and contains a pass/fail status, with an exception message for failed tests. (IBM, 2019)

3.3 Agile software development

In modern software development, agile methods such as Continuous Integration and Deployment (CICD) are widely used (Elbaum et ak, 2014). The benefits of these methods are thought to be both reducing latency in bringing new products and features to market and increasing quality through decreased time to feedback (Elbaum et al, 2014**)**. Since the purpose of the testing system under development is to support a CICD system, this chapter includes an overview of Continuous Integration, Continuous Deployment and other relevant DevOps methods currently in use.

3.3.1 Continuous integration

In continuous integration, software build automation is combined with automatic testing. Changes made by developers in a codebase are prone to cause failures, but integrating minor changes often is thought to help avoid errors.

Using CI is thought to bring several benefits. Automatic testing, a central requirement of CI, often helps with finding errors early. Building automatically on a server instead of developers' computers helps enforce a common build environment, easing the discovery of dependency differences on development environments. CI is also thought to help developers iterate faster and deploy more often. Additionally, via automation builds and tests can be directed at several platforms in parallel. Executing tests in parallel helps reduce the issue of test suites taking up all the resources available. Finally, CI is thought to increase developer confidence in their work. (Hilton, Tunnell, Huang, Marinov, & Dig, 2016.)

Increased test quality is also considered a benefit of CI, since badly written tests are quickly exposed by broken builds (Pinto et al, 2017). Likewise, developer investment in testing success is thought to increase due to the increased visibility of results (Hilton et al, 2016).

Even though CI is widely considered useful, there are several potential problems with using it, as reported by developers. Since the tests happen automatically in the background, the developer does not see real time information. This leads to problems with identifying the cause of errors. Alternatively, an inadequate test suite can lead to some builds falsely passing (Pinto et al, 2017). Additional problems stem from the long build times that executing several tests can lead to. This is especially visible with testing heavily GUI dependant applications. Lack of compatible tooling for a specific workflow is also considered a problem. Finally, security of automated build processes can be considered problematic. (Hilton et al, 2016.)

The volume of tests that take place can be considered both a cost and a benefit. While test suites tend to occupy all the space they're given, repeated test execution can help identify components that fail sometimes due to instability in the infrastructure (Roseberry, 2014). This type of error is typically difficult to spot in limited manual testing runs, as the developer does not have the patience to repeatedly run one seemingly successful test. However, with this benefit comes a key challenge for the test automation system to determine when a component is working in a stable manner. (Roseberry, 2014)

One of the core goals of CI is to increase the frequency of sharing results with other project contributors and there is some evidence to support that this goal is being achieved (Hilton et al, 2016). However, another core goal of decreasing the size of individual commits and increasing their frequency does not appear to be fulfilled. (Baltes, Knack, Anastasiou, Tymann, & Diehl, (2018.)

3.3.2 Other continuous methods

Continuous integration is often extended further with other agile methods, such as continuous deployment, delivery and testing. These methods are thought to add on to CI, further increasing quality and decreasing development time.

Sometimes considered as a part of CI, Continuous Deployment (CD) is the act of automatically deploying a product to an environment. Doing this allows for a higher frequency of new features arriving to the customer but requires care as to avoid introducing instability to the software. This can be extended further to Continuous Delivery, which is the act of frequently releasing features. (Hilton et al, 2016.)

While testing automatic testing is a key part of any CICD workflow, Continuous Testing (CT) takes the concept a step further. In continuous testing, the changes being made get tested automatically as they happen in the background, to alert the developer of breakage. CT is generally aimed at unit testing, since these are usually fast to execute. (Moe, Cruzes, Dybå & Mikkelsen, 2015.)

Testing this way is well suited for test driven development, where tests evaluating system performance are written before the program code. Since the code is executed constantly in the background, changing the test state from failing to passing is seen immediately. Tests that can be run continuously should be very short, taking less than a second to execute in total. This is in part made possible by the tests happening on the unit testing level, since one changing a file will only result in a small subset of tests executing and the unit tests are over quickly as background tasks. (Moe et al, 2015.)

4 RPA AT THE CASE COMPANY

There are currently two main categories of development related to RPA: the automated processes themselves, created by MVP developers, and reusable components and background methods created by the RPA team.

The current state of the infrastructure, processes for product creation and the implementation of the test system are described as general in nature unconnected to the specific RPA provider. However, the actual implementation work has been done using the product stack provided by UiPath and some of the problems the system is created to solve may be platform specific rather than problems inherent to RPA. While the issues that extensive testing can uncover with other RPA products may be different, the methods for uncovering those issues should be the same.

4.1 Infrastructure

An RPA at the case company described as a software process that carries out a process generally on a Virtual Desktop Interface (VDI) using the UiPath software. The processes in question are varied. They can last either a few minutes, or span over several hours. They can also be based on schedules or be executed on demand based on user input.

The RPAs are both created and run using UiPath. Both development for QA purposes and production jobs are run on virtual desktop machines. The scheduling and triggering of RPA jobs is done through UiPath’s Orchestrator, a tool built for this purpose. Orchestrator is hosted on a separate server from the virtual desktop machines. UiPath’s Orchestrator contains the existing RPAs, schedules and jobs to be run. Additionally, some settings for the virtual machines are hosted on Orchestrator. This structure is roughly outlined in Figure 5. Logging information on RPA runs is also available on Orchestrator. It can be interfaced with either via a web interface or through an API provided by UiPath.

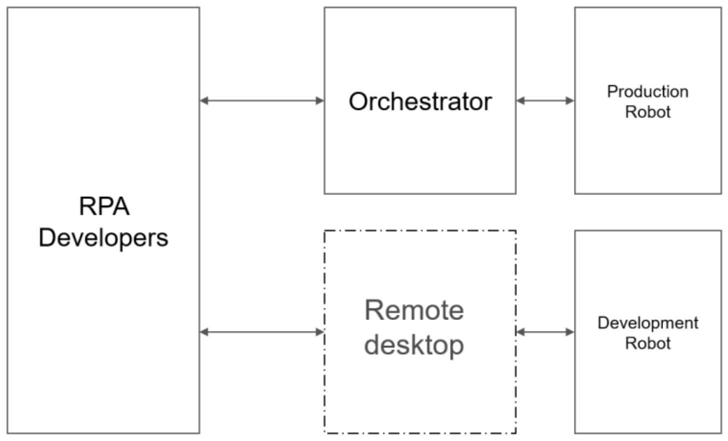


Figure 5. Depiction of development and production infrastructure. RPA developers interface with production robots through Orchestrator. On the other hand, they directly connect to development robots as a remote desktop.

In UiPath, the RPA processes are defined using the following terminology, as further described in Figure 6. A process is a deployed RPA, which generally interacts with an instance of Windows running on a Virtual Machine. Processes are created using UiPath studio and packaged for deployment using tools provided by UiPath. The packages are NuGet package files, containing information on dependencies and Microsoft's Workflow Foundation `.xaml` files. A "job" is a specific instance of a process, triggered through the Orchestratror. A schedule can be used to trigger jobs on demand. Inside an RPA, there are modules designed for carrying out business logic and reusable components. Reusable components are either provided by the RPA provider or are created in-house. Both can either interact with GUI level tasks or execute in the background like a normal software process. (UiPath, 2019.)

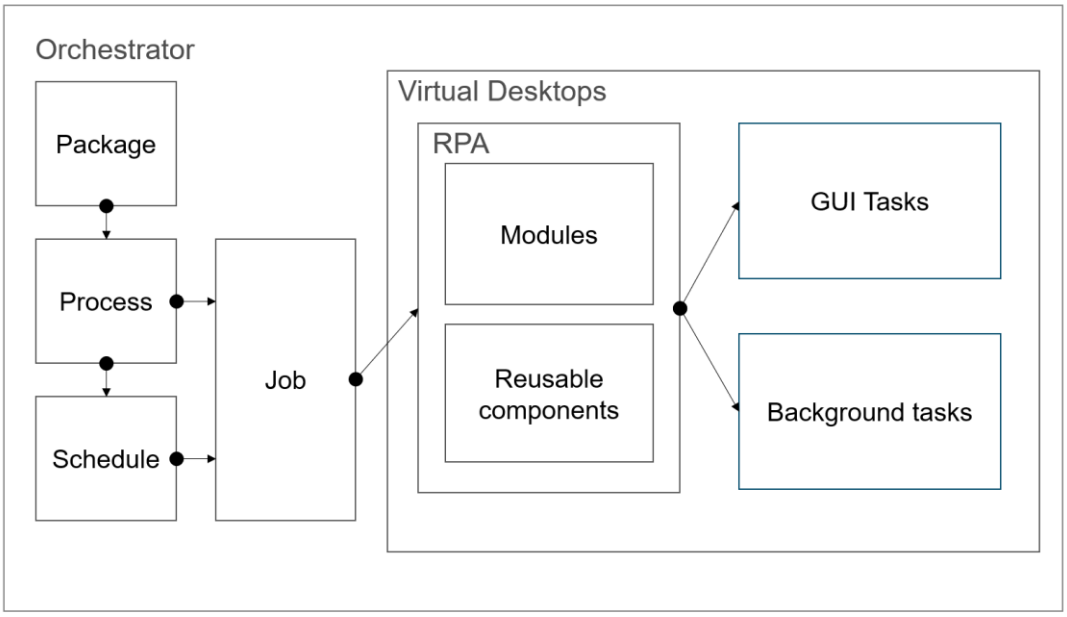


Figure 6. Structure of the RPA infrastructure. Orchestrator hosts packages, which can spawn several processes. A job is a specific runtime instance of a process, which can also be started via schedule. Internally on the virtual desktop, the RPA consists of reusable components which can be shared between RPAs and RPA code. Both types of activities can do either GUI or background tasks.

In-house reusable component creation is a part of the RPA process. The components can be implemented inside UiPath or written using a .NET language. .NET components can be created by implementing a function in an interface made available by WF, which makes it visible in the UiPath GUI. Alternatively, the component can be implemented as a simple method, compiled and included in a project via code. A reusable component can be imagined to be a function or a method in object-oriented programming, created once and used several times. Parts of the design process are abstracted away as components inside an RPA tools lack concepts such as inheritance and interfaces. Instead, the interaction of a component to the rest of the RPA is set up with a simple implementation of input and output arguments.

4.2 RPA development

Development of RPAs can be split into three main phases, MVP (Minimum Viable Product) development, Quality Assurance (QA) and maintenance. Most of the development work happens in the first two phases, while maintenance is a constantly ongoing process executed on demand. In case of errors, the RPA is handed back to either QA personnel or the MVP developer, based on the type of problem encountered. This process is pictured in Figure 7.

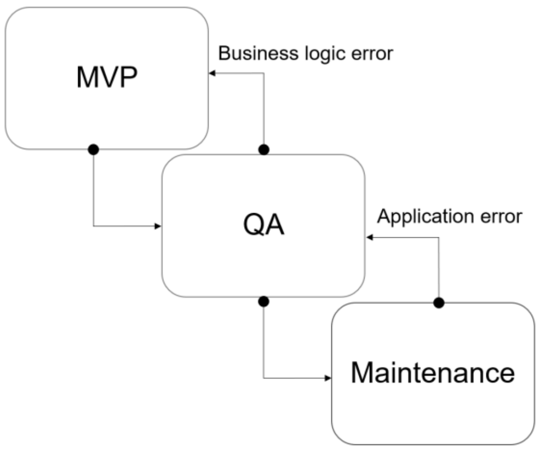


Figure 7. Stages of RPA development. The original developer creates an MVP in their own environment. The integration developers then handle the QA phase and the RPA is put to production. Maintenance is also done by integration developers.

The structure of a complete RPA product is shown in Figure 8. The `modules` folder contains separate parts of the workflow being automated, while the `tests` folder contains separate tests for each of those, where possible. Currently there is generally one test per module, but this can vary on a per module basis. `Main.xaml` is the file used to begin the execution of the RPA. Finally, the `project.json` file is automatically generated by UiPath and holds information on the version, name and dependencies of the RPA.

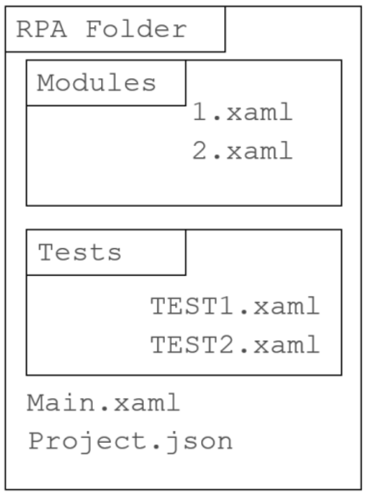


Figure 8. Structure of an RPA. `Project.json` contains version and dependency information for the RPA, while the `Modules` folder holds the business logic. Ideally, there will be at least one test per module.

RPAs are mostly developed by MVP (Minimum Viable Product) developers, who are experts in their own fields and spend some of their time automating processes. The MVP developers generally do not have knowledge of software development. They automate processes they want to with assistance from the RPA team as needed. The MVP development process is illustrated in Figure 9. First, the MVP developer uses their own methods and process knowledge for creating a functioning example of the process. Second, the RPA team prepares for receiving the MVP by preparing the necessary shared folders for hosting the RPA. Third, the IT department is required for granting access to the developer to this folder. Fourth, the original developer shares the project in the agreed place. Fifth and finally, the project moves on to QA. This stage of the whole RPA development process results in varying levels of initial quality and is the only stage where the assistance of the IT department is needed.

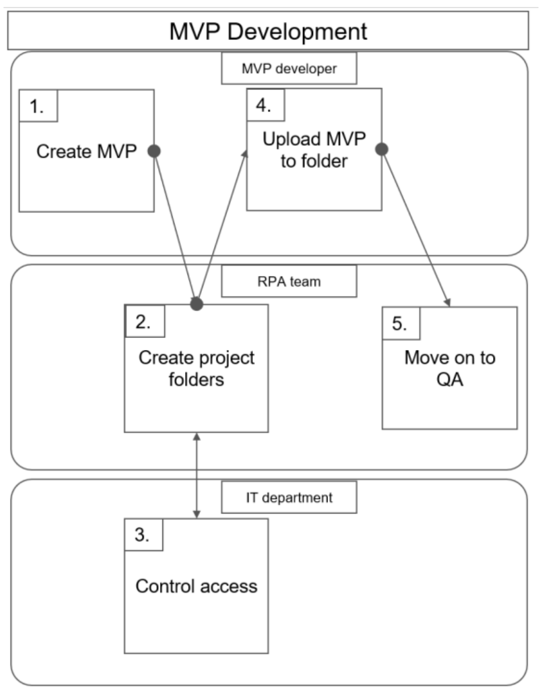


Figure 9. The MVP development process. The process consists of a very small amount of work for the RPA team, but unfortunately requires assistance from the IT department.

The quality assurance process consists of several steps, which are illustrated in Figure 10. The RPA is split into submodules as needed and tests are created for these modules. In addition, the production environment is prepared for the RPA by requesting necessary system access rights, for example in Enterprise Resource Planning (ERP) software, and by doing software installation. During this stage, the original MVP developer supports in case of business specific information requirements. In addition, exception handling is added to the modules, unstable parts of the RPAs are improved, and logging is added. Often, testing is done on a virtual desktop machine to avoid having to request access rights and install additional software for the RPA team member. Finally, the RPA is made to save results in a standard location, defined by the runtime environment and timestamps. Once the initial QA process is finished the RPA is packaged using a specific build computer (currently a computer belonging to a developer) and the RPA is uploaded to Orchestrator for User Validation Testing (UVT) with quality data in cooperation with the MVP developer and key users. Once the UVT is satisfactorily completed, the RPA is set to use production data, usually by changing a variable in Orchestrator, sometimes by driving a patch to the RPA, and is piloted. After the piloting phase, the RPA is deemed complete and monitoring begins for maintenance.

Currently version control is done informally by creating new folders in shared disc for major versions. Each new folder requires a step from IT to grant access rights to MVP developers.

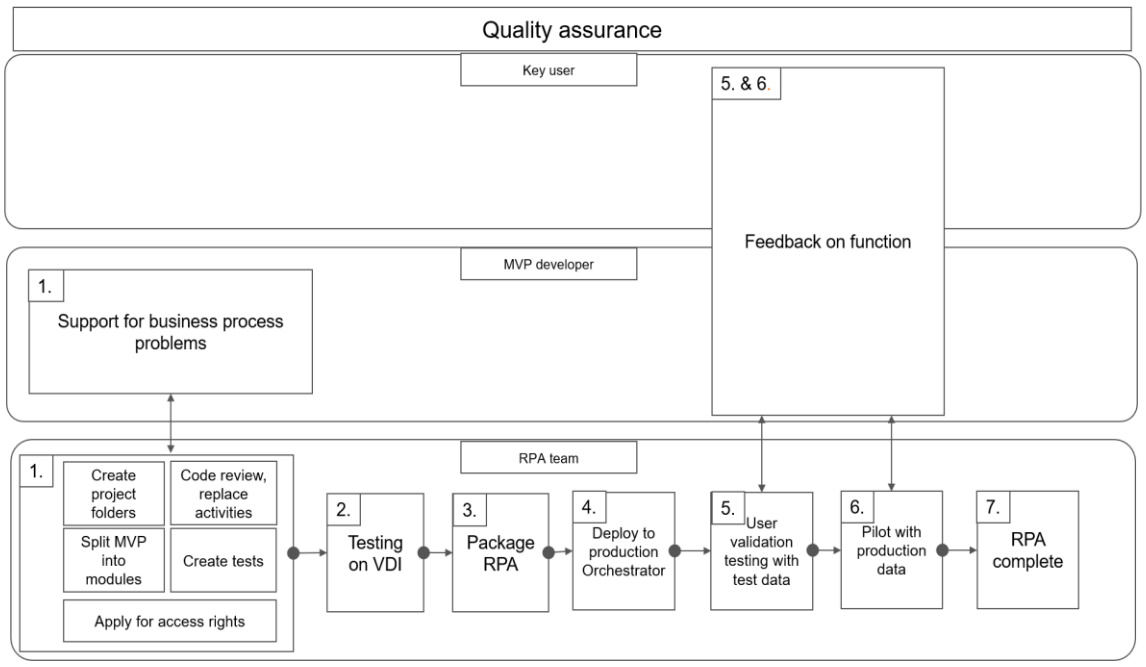


Figure 10. The figure details how various stakeholders participate in the quality assurance process. Most of the work is done by the RPA team, with the MVP developer acting as support and the key users giving final feedback.

The result of the QA process is an RPA with logically separate modules and tests for those modules. At this point, the RPA should run on a production machine in a stable manner.

4.3 Component development

The component development workflow is less developed, as it is currently not as common as development of RPAs. However, with future better support of reusable components in

UiPath Studio, the current process should still be documented so that it can be improved. The workflow as it stands is shown in Figure 11.

First, a request for a component is received. Requests for component development are currently informal, but the main driver is a need identified by the RPA team. Most of the development of a component is done by a single member of the team, including initial development and testing. Upon completion of the component, it is uploaded to a shared file system so that it is available to other RPAs. The component is then piloted as a part of an RPA and taken to production with it. Monitoring occurs as a part of the RPAs which use the component.

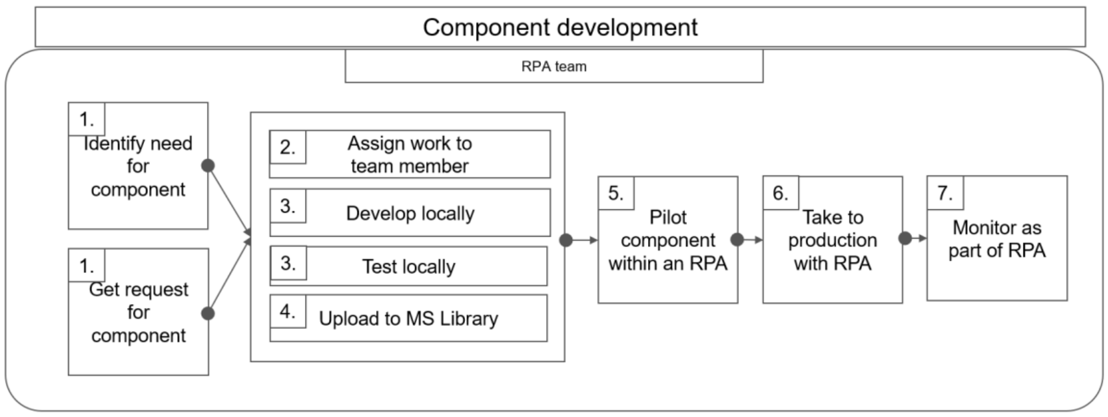

Component development
RPA team
1. Identify need for component
1. Get request for component
2. Assign work to team member
3. Develop locally
3. Test locally
4. Upload to MS Library
5. Pilot component within an RPA
6. Take to production with RPA
7. Monitor as part of RPA

Figure 11. The component development workflow is handled almost entirely by the RPA team. However, the end users can contribute by requesting specific components.

4.4 State of testing

Testing of RPAs is currently done manually. As a part of the beginning stages of QA, individual tests are created for parts of an RPA. These tests are executed manually, and their results are likewise manually validated. The test execution at this stage is usually done on VDI closely approximating the production environment. The environments are not quite identical, due to the test environment containing out of date data and driver differences in remote desktop sessions when run from a developer's computer. Later, when the development work is deemed complete, further testing is done in cooperation with the end-users, in UVT and piloting. The UVT stage more closely resembles the production environment, since the process is triggered through UiPath’s Orchestrator, instead of manually on a visible virtual desktop. The RPA being triggered through Orchestrator has been a cause of some problems in this stage. The piloting phase is generally done in the production environment.

Testing of RPA components is also achieved manually. While there are currently no separate tests for the components, most of them can be run independently with their default parameters for manual validation. Further validation is currently achieved by running tests associated with RPAs using the component. Test success is evaluated manually by the person running the test, either by looking at output data or the ending state of the UUT.

4.5 Evaluation of the current state

In terms of testing, there are a few problems in execution, validation and environment. Currently, tests are executed manually based on a judgement call made by the developer. This may lead to problems, due to unforeseen impacts of changes. Furthermore, it is too easy to forget to run all affected test cases.

In addition, further effort is required from the developers in the form of manual validation of testing. While automatic validation of test results could already be carried out, it is not required, and a developer is unlikely to create validation on their own. In an automated system, some test result validation must be created for the test to work at all, removing the need for this sort of discipline. A test case should define both the inputs and expected outputs for itself, so that the system can immediately tell if the process has successfully executed. Normally, in a test this is done by using an `assert` command (Garousi et al, 2018). Since UiPath does not natively include support for .NET testing tools, a similar command can be created in a custom component.

Finally, there are several issues with the testing environments currently in use. The tests are usually executed on the development machine, leading to unintentional inclusion of certain system settings and dependencies, which are discovered when moving to production. Adding a separate testing environment should help discover these issues before testing that includes an end user. In addition, this should help with defining the requirements for a process, which should help with setting up future production environments. The development environment is kept equivalent to the production environment manually, which is an issue that should be remedied with automation. This automation solution should also be carried over to the new testing environment. In addition, some software is automatically updated or is in other ways dependent on external constantly changing information. These unmanaged changes should be disabled where possible and carried out intentionally via managed installations. Fixing some of these issues can be achieved via automation and good test design, whereas other infrastructural problems need to be fixed manually.

Manual testing is typically done while connected to a virtual desktop environment. Ideally, this would be identical to the production environment, but small differences between the two accumulate over time. Additionally, starting processes through Orchestrator sometimes results in unpredictable issues not reproducible during manual testing. Currently, testing a robot on a close approximation of a production environment requires packaging the project and uploading it to Orchestrator, where it can be run. The limitations of this approach are that it is a multi-stage manual process which provides limited results, since the package is limited to running the entirety of the process and the developer cannot directly observe what is going on. This could be worked around, by creating specific test packages and uploading these separately, but this would require a significant amount of manual work. An attempt to work around this will be made with a special test runner RPA, which can choose specific modules and tests out of other RPAs, to be tested automatically. Executing tests often through Orchestrator should be able to highlight problems with the RPA before UVT, saving time for both developers and the end-users.

However, this test automation is still limited by several factors. First, since the RPAs mostly interact with GUIs, the tests are slow to execute. This can be especially problematic, when the tests require long processes for pre-state generation and post-state cleanup. Pre- and post-state generation are both also a source of flakiness so they should be minimized where possible. Second, some of the problems identified by the tests have less to do with errors in the RPA and more with issues with an unstable test/production environment. However, identifying unstable behavior can be helpful, since RPAs should be resilient enough to work through slight problems with system availability. This instability has to be in the module under test for this to be useful, instead of being in the actual test code. Third, the test data is not universally available. An example of this is enterprise resource planning software, where the quality data is refreshed periodically, deleting changes previously made to the environment. Because of this, not all tests can be executed automatically. However, applying automation where possible should help make manual test execution less labor intensive and the test environment more like the production environment.

5 IMPLEMENTATION

This chapter details the planning and development of the system that was built to alleviate some of the problems identified in the current state analysis.

5.1 Requirements for the testing system

The automated test system to be created should adhere to certain constraints elicited from the literature review. At its' most basic, the testing system should be able to take premade tests and execute them based on some sort of input test suite. However, this can be made much more useful by adding some features that have been generated based on the literature review, which are further enumerated in Table 2.

Table 2. Requirements for the automated test system based on the literature review. The system should support several levels and types of tests. Additionally, creation of high quality, reliable tests should be supported. These tests should also be connected to a CI pipeline, which requires certain considerations.

Feature	Source papers
Avoiding flaky tests	Roseberry, 2016
Design of functional tests	Baresi & Pezzè, 2014
Designing tests for CI	Pinto et al, 2017
Regression testing	Baresi & Pezzè, 2006
Test quality	Garousi, Kucuk & Felderer, 2018
Testing level separation	Baresi, 2006 & Garousi, 2018

Avoiding flaky tests is best done by implementing straightforward ways for generating pre-states. This is done by having support for specific repeatable pre-state generation workflows. This is expanded further by supporting test cleanup workflows that are reliably executed, whether a test fails or succeeds. This also ties into designing functional test, which is further supported with both a written guideline and purpose-made assert type components. Separation between testing levels is achieved via time constraints, as low-level tests typically take significantly less time to execute. A limit on the execution time on tests that are run often will encourage keeping tests on their specific levels. Tests that are run nightly can execute with loose time constraints.

5.2 Platform limitations

UiPath does not have built in support for generic testing frameworks. In addition, existing code coverage tools do not function. Because of this, the approach to testing RPAs created with UiPath will be made within UiPath's own framework. Reports on the test results must be created in a format that generic CICD software can parse e.g. in the XML format.

In addition, our control over our test environment is not as thorough as required. This creates some problems with both test execution and maintenance. Since our production RPAs are running on a virtual desktop, with a standard Windows 10 + Software Center setup, some automatic updating of processes is currently inevitable. This means that the versions of software being depended on are not in full control of the developers.

The RPAs are also extensively dependent of external systems such as ERP software, which are prone to having intermittent outages. While this causes problems in production, it is also inconvenient for testing since a test failure does not necessarily reflect a failure in code, but a failure in infrastructure. The flakiness caused by these factors can be mitigated by rerunning tests to identify stable problems and by programming the RPAs to be more robust in handling small infrastructural problems. Test flakiness, while inconvenient, can be used to identify unstable parts of the infrastructure used and through that can lead to developing more resilient processes in the future.

Finally, some systems are missing a stable testing environment, which can cause extensive test breakage every time if the test data is not hosted externally. Often repairing the test data requires business knowledge which the RPA team does not have, requiring input from MVP developers. In some cases, the test data can be created at the start of a test, but this is usually not the case. Because of this, some components are classed as not automatically testable, until a more stable testing environment can be created.

5.3 Implementing testing levels and types for RPA

RPA development is very similar to normal software development. Due to this, it should be testable using the same methods as software. Due to RPA working on a higher level of abstraction, there are some marked differences.

The concept of the different testing levels is applied as follows. Testing on the unit level focuses on the smallest available unit of RPA, a module. Mostly functional in nature, these tests are run locally as a part of development. The key differences to software development are the size of the individual units and the method of execution. Functional testing for a self-contained module may take several minutes, while unit level testing times for software are normally counted in seconds. It should be noted that this type of testing can also be done in relation to RPAs, but it is contained to the testing done by the RPA provider and possibly made software components. The method of execution for unit testing software is easily automated and large testing suites are routinely executed as part of a built action. For RPAs, however, the unit is fundamentally tied to the environment in which it was created, and test execution is done locally via manual execution. Functionality of the RPA product in the development environment is determined at this stage.

Integration testing is more similar than unit testing in software and RPA development, there are however some key differences. With the basic functionality of the RPA established on the unit level, integration testing focuses on the interaction of the developed modules together and with the runtime environment. At this stage, the stability of the interaction between developed modules can be tested. In addition, the interaction between different exception handling schemes implemented inside modules is tested. Additionally, this level contains testing the RPA for problems stemming from the almost inevitable differences between a development and a testing environment. This holds similarly for classic software development, with the key difference being the test execution times. For software, integration tests can still mostly happen in the background possibly escalating to the GUI level towards the end of a successful test suite. However, for RPAs, most of the testing at this stage is focused on the GUI side and as a direct consequence will take much longer.

Regression testing can be carried out on some of the integration tests. For background focused tasks, a fast to execute regression testing suite can be created. This test suite should include both basic logical tests created during development and contain specific tests checking for bugs that have been fixed. Testing for regression should help avoid bugs from reappearing when changes are made. For larger GUI based modules, the size of the regression test suite must be actively maintained to ensure fast feedback. A method for determining files affected by changes made should be implemented for regression testing, since executing all the regression tests for a project is cost prohibitive.

System level testing is carried out very similarly between classic software and RPA. This stage generally involves end users and is the most difficult to automate. The users get the opportunity to interact with the software, via whatever interfaces that are implemented. At this stage, the implementation of the software or RPA product is evaluated against its specification. Being the highest level of testing, issues found here are the most difficult to fix. Due to this, careful design rather than automatic testing should be applied for this level.

5.4 Tool selection

Since UiPath’s RPA offering is based on Microsoft’s Workflow Foundation (WF), it is technically possible to create and execute RPAs via .NET code. Testing could also be handled using this method. Instead of doing this, the test runner was implemented directly in UiPath.

The main reason for using UiPath for creating and running tests is dependency management. While it is possible to manually include all the required software dependencies for running RPAs, there is no tooling for doing this. In addition, creating the tests using the same software which is used for creating the RPA process makes the development process more consistent. Using RPA software for creating the test allows for using RPA for pre-state generation, and post-state cleanup as well. RPA software natively contains a significant number of modules meant for interfacing with GUIs, which can be leveraged for test development. Additionally, analyzing test results is easily done as the software can easily check for the existence of for example specific popups. With a normal software development language, external testing libraries have been developed to do the same, but this would require adding even more dependencies to the project.

If the issue of loading automatically UiPath's own packages into a development environment such as Visual Studio could be solved, test development via code might be a viable option. The greatest benefit of creating tests using a .NET language would be the inclusion of pre-existing testing frameworks such as NUnit. Additionally, this would help with leveraging existing .NET testing skills in the workforce. On the other hand, developers specializing in RPA would have to be re-educated to create tests. Some reverse engineering could also be done to use the other RPA components through code, but this presents a significant body of work to be done. In conclusion, unless an RPA provider provides new built-in tools for interfacing with existing testing frameworks, using the RPA software is preferred.

5.5 Plan for implementation

There are several steps that need to be taken to facilitate useful test automation. These steps are version control, test orchestration, execution, reporting and creation.

First, version control needs to be applied to the RPA project. This will be done with Git. This is needed to connect to the build system, which in turn is used for triggering tests. It is also considered good software development practice to avoid lost work.

Second, a system for orchestrating building and test execution needs to be set up. Test orchestration is the process for determining that tests need to be run and starting their execution. For this, a CI tool called Azure DevOps is used, due to its ability to later accommodate solutions for CICD. The same software can be used for presenting test results created by the test runner.

Third, a custom test runner needs to be created, which can trigger useful test runs. The focus of this will be integration testing, both since integration is currently the highest risk area identified in the RPA development process and because it is the simplest form of automatic testing to implement, due to lacking test frameworks. Reporting of test results needs to be created in such a way that Azure DevOps can display them.

Fourth, guidelines for testing need to be created, so that the organization can achieve consistency in results. Creating necessary templates for quickly generating pre-states and validating post-states will also need to be created where possible. Where applicable, software components for testing also need to be created. Developers also need to be taught the basics of using the new system.

Fifth and finally, the previous steps need to be tested. This testing can first be done with small scale dummy projects made for this purpose alone. The next stage, however, will require applying the testing process on actual RPAs to evaluate existing features and elicit required future features.

The proposed pipeline for facilitating test automation is as follows. The version control system connects to the CI tool, which uses a set of scripts to trigger tests through the Orchestrator API. The test runner executes the tests specified either as a list by developers or automatically generated based on changes made and records the results. The test results are then returned to the CI tool to be viewed by developers. The entirety of the proposed development system is shown in Figure 12.

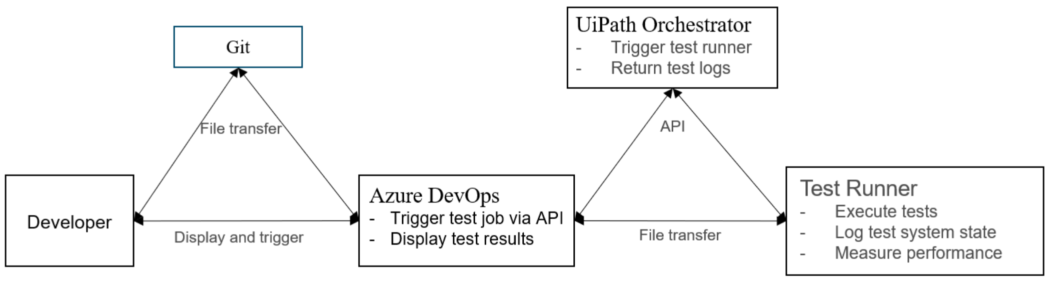


Figure 12. Depiction of the agents required for triggering tests. The developer directly interacts with Git and Azure DevOps. Azure DevOps then uses data retrieved from Git to trigger tests via UiPath Orchestrator and both gives test data and retrieves test results directly from the test runner. The connections between these tools are either custom created by utilizing their respective APIs, or already exist.

5.6 Tools

While this specific set of tools is used, in principle any system offering similar features will do. Version control of RPAs will be done with Git in a single repository. The repository is hosted on Azure DevOps (previously known as Visual Studio Team Services). A specific branching strategy will be utilized to help with test selection per project. Automated tests can be triggered either by changes on a branch or a certain file. These changes are automatically monitored by Azure DevOps with their own built-in system.

Azure DevOps is used to trigger automatic tests and display their results. To trigger tests from Azure Devops, a combination of MSBuild, Powershell and the UiPath Orchestrator API is used. Generally, MSBuild is used to move files to locations reachable by the test runner and to trigger Powershell scripts. Powershell scripts are then used to access the Orchestrator API, which is required for triggering the test runner on a desired test machine. The integration between Azure DevOps and UiPath Orchestrator was created for this purpose applying their respective APIs, since it was not previously available.

The test runner was created from scratch using UiPath Studio and deployed to a separate test Orchestrator instance. The test runner was built based on the literature review, requiring few dependencies and allowing for easy test preparation, cleanup and validation processes. Additionally, special care was taken to maintain the type of execution, a remote connection to a Windows Virtual Desktop triggered through UiPath's robot service, the same as it is in the production environment. The test Orchestrator is kept separate from production, both to avoid faults and to separate the test results from production runs. A test environment gives additional benefits for testing changes in the infrastructure itself. However, maintaining a second environment as a close copy of the production environment is an investment in time. The test machines are currently manually created to be close copies of the same machines in production. The system could be further developed by automatically creating the test environments as direct copies of the production environment. However, this is outside the scope of this work.

Using the described tooling requires some changes to the RPA infrastructure, which are shown in Figure 13. Previously development machines were used for testing during development, while in the new working method new machines that are dedicated for testing are required. In principle, these could still be unused development machines e.g. during evenings. However, separating the test environment from the development environment would help with discovering environment changes erroneously made during the development process, which could adversely affect bringing the project to production. In addition, a separate Orchestrator instance is required. While using a separate instance for testing does increase the maintenance cost of the Orchestrator environment, it separates production runs from possibly high-risk actions. With a separate environment, developers can confidently test functionality without the test results appearing in production logs and monitoring as failures.

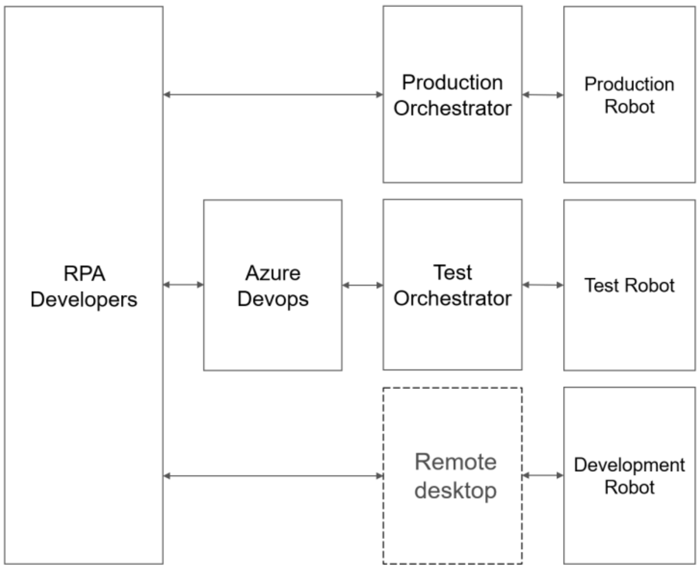


Figure 13. Depiction of changed infrastructure from Figure 5. The main difference is the addition of separate test robots that are used after the development stage to validate functionality and to run regression tests.

5.7 Build process

The build process that was created involves four stages: package, test, publish and clean. Packaging is achieved by calling the UiPath build process via specially made scripts, which is then continued by passing the results to the test system. The testing phase moves the RPA under test to a shared file system and uses an integration made to UiPath's Orchestrator API to trigger the test runner robot to carry out the tests. The publish phase uploads the results generated by the test executor robot to the CICD tool. Finally, the cleaning phase attempts to remove temporary files created during the build process. The last two are achieved by built in publish and clean tools of the CICD tool. The testing and publish phases are optional and can be executed depending on project demands if needed.

5.8 Use cases for test execution

Tests are executed as a part of a build action through Continuous Integration software. The benefit of executing tests as a part of the build action is that the package created at this stage is more likely to be functional. The same package can then be deployed to either production or further testing as needed, based on the test results.

The RPAs can be tested in several ways. Basic functional tests on a local computer should be run before submitting to the shared repository. Full integration however needs to be tested on a close copy of the production environment. There are different cases for integrating with production systems on the current infrastructure setup.

One case is testing a changed RPA. This is the simplest case, as the tests related to that RPA can be executed. However, during the initial testing of the test runner, it was discovered that it is easy to create workflows that result in test suites lasting well over an hour. Due to this, a simple system of including and excluding files from test and detecting changes in files was created.

While automatically detecting dependencies through file accesses was thought to be useful, it was not deemed necessary. Inside an RPA, the dependencies are generally limited in number and can be managed manually. The dependencies RPAs have to outside packages are on the other hand already automatically managed by the RPA software. In other words, when a module is changed it is most often enough to execute one well designed test. Some changes are notable exceptions to this, however. For example, when input or output arguments are added or removed from a module, all the dependencies involved need to be updated to reflect this change. Failing to do this will result in failures in dependencies. This scenario is quite rare, so manual handling of dependencies can be done. An additional solution to this problem is to run smoke testing, which execute critical RPA workflows to reveal faults. This can be done as a part of a nightly build action for cases that are not time sensitive. Implementing a more automated test dependency management system would be a useful improvement.

Second case is testing a changed subcomponent which is used by several RPAs. To validate the integration of the change in all RPAs, a lot of tests need to be run, which might be difficult in terms of the time taken. Some tests are also significantly simpler to execute than others. For example, functional testing for a module for logging into software is simple. However, checking that the module integrates into an RPA properly, sometimes whole RPAs need to be run. Fortunately, a component that passes functional tests will also usually pass integration tests. Best efforts should still be made for executing integration tests for components.

The proposed solution to the problem of long test times is splitting builds into four categories: continuous, component, under development and nightly. Continuous builds include no testing, but simply produce a package. These builds take little time, so they can be executed on any change in the repository. Component builds cover RPA components, which have relatively short tests and can cause breakage in several RPAs. In the future, as the component list becomes longer, this category may have to be broken into subcategories. These builds should include a significant amount of testing. Test execution for builds targeted on RPAs that are under active development can be based on changes in the repository. Initially, the list of under development RPAs is manually curated. However, it may be possible to automate this process in the future based on internal signals for RPA completion. Finally, nightly builds can involve both the testing of changes and a constantly updated list of explicitly included tests, which are not as limited by time.

In terms of time taken the priorities are as follows, Continuous builds should take less than 10 minutes, so that the created package can be rapidly used if required. Component builds should be similarly fast, in order to facilitate the rapid deployment of fixes to critical components. The time taken for development builds is project dependent and can be set by developers. However, generally these should be kept short enough to run during a break, which is to say 15-30 minutes. Finally, nightly builds can last the entire nights and should include all the tests that could not be accommodated during the day. These can be used both to discover faults in the changes made during the day, to discover errors through flaky tests and to test for problems in infrastructure. The different types of testing run are enumerated in Table 3.

Table 3. Types and lengths of different testing runs. Continuous tests are short and low level, whereas tests targeting components try to achieve a compromise between speed and code coverage. Nightly tests can test the full system integration.

Test type	Length	Level
Continuous	<10 min	Unit
Component	10-14 min	Integration
Nightly	>30 min	System

Unfortunately, the integration of all components is not automatically testable due to the previously outlined limitations in the test environment. Integration tests should be limited when problems with the testing environment or the time constraints are insurmountable. These tests will be mainly executed while the RPA is actively under development. During development, the most important tests, i.e. the ones prone to breaking, can be identified and added to the nightly testing list. After being assigned to the nightly build, their breaking can be detected and the value of repairing the test data can be evaluated on a case by case basis.

5.9 Test Executor Robot Features

The test executor robot is an RPA deployed to Orchestrator whose job is to test other RPAs. The robot requires inputs in two forms, Orchestrator queue items and test files under a file path that can be accessed by the test runner RPA. This location can be either hosted directly on the test runner machine, or on a network file system. Using a network file system for spreading test files is the simplest way of having the files available for use. If the test machine is also the Azure DevOps build machine, however, the test files are already locally available. Both the queue items and test files are produced by the build process.

All test phases produce test result files in the JUnit test result format. While JUnit is a testing framework for Java programs and is in no way involved here, it has an easily implemented result format. Due to its popularity, the format is also widely supported by DevOps tools. In the case of Azure DevOps, the program can read and display test results that are created complying with the JUnit format.

The implemented test result file includes the name of each test that was executed, the result of the test and its' execution time. In addition, exceptions result in screenshots being taken of the current desktop state. A visual comparison is made between the beginning and end states of tests and if a discrepancy is detected, screenshots of both are taken. Screenshots are taken to help with debugging, as one large issue with testing on a remote machine is the lack of visibility for whatever goes wrong during the tests. Execution logs for the robot are also retrieved through the Orchestrator API and saved for every set of tests.

The flow of test execution is as follows, as depicted in Figure 15. First, the test runner RPA is invoked, usually via the Orchestrator API. Second, the test runner robot reads a queue item from Orchestrator with the name of the test to be executed. Third, the robot copies the necessary test files from shared file system to a local folder, to match the execution conditions in production. Fourth, the robot iterates through the tests that are set to be executed, executing each one and depositing results locally. Fifth and finally, the robot uploads the result files back to the shared file system to be retrieved by the build system.

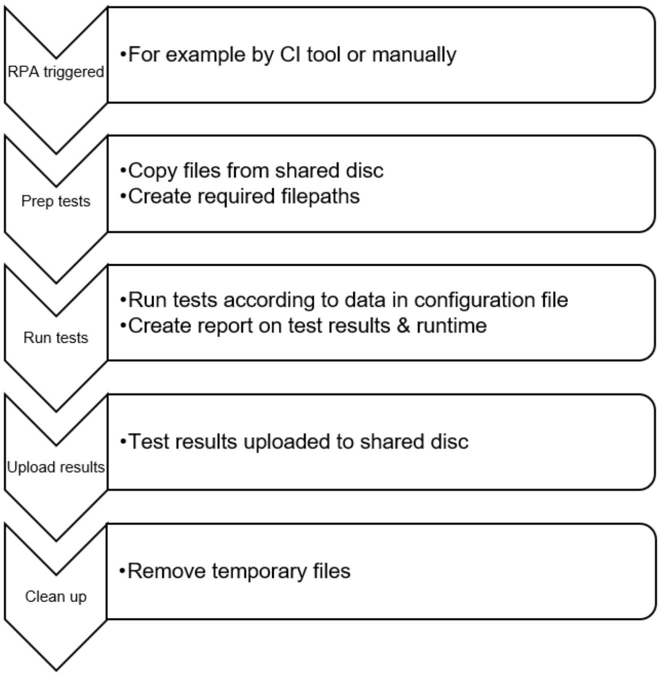

RPA triggered
•For example by CI tool or manually
Prep tests
•Copy files from shared disc
•Create required filepaths
Run tests
•Run tests according to data in configuration file
•Create report on test results & runtime
Upload results
•Test results uploaded to shared disc
Clean up
•Remove temporary files

Figure 14. Workflow of the test executor RPA. The trigger mechanism can be automatic. Test preparation requires moving files to a location accessible by the robot. Running tests is straightforward, although the test runner has to be robustly built for failing tests not to cause problems. Finally, results are shared back to be visible with the developers.

In addition to running the tests inside the RPA under test, the test executor carries out checking of dependencies, system resources and system variables. Packages installed on the test machine are retrieved from the installation location on the test machine. This list is then compared to the list included inside the `project.json` file of the RPA under test, which can be manually updated by a developer or automatically updated by using UiPath's tools. System resources are logged under a monitoring initiative, which seeks to connect faults to specific types of resource usage. The types of resource usage currently logged are the percentage of memory used, free storage space in MB, the amount it takes for the robot to successfully log in to Windows and system uptime. Finally, the values of some system variables are logged to trace the name of the computer executing the tests and the versions of Windows and .NET framework which are running on said machine both to help debug faults and create a trace for the future of the environment the tests were executed in.

5.10 Test schemes

As expected due to the results of the literature review, executing all the tests all the time takes too long to be useful for development (Elbaum et al, 2014)(Gligoric et al, 2015). During the initial pilot project for the test executor, executing all the tests resulted in a runtime of over 1 hour for the build process, for a relatively small project that takes about 20 minutes to run by itself. Because of this, the following test schemes were evaluated in practice.

- Only the tests connected to changed files are executed.
- All tests run initially but transition to only executing the main integration test as the test times become prohibitive.
- Tests chosen manually by developer are executed on every change to project.
- Triggering all tests for changed projects, but only during the night. Daytime would use shorter test scheme.
- Combinations of the four.

Combining the test schemes on a per-project basis was identified as the most useful approach, since both the features and system requirements of RPAs vary significantly per project. Shorter RPAs can be executed completely, whereas for longer RPAs only critical tests chosen by developers should be run during the day and longer tests should be delayed until the evening. In the case of executing only changed files, a list of safe tests should be included inside the project folder, as making permanent changes to the shared test environment is impractical.

5.11 Test design

The modules under development can be tested in a multitude of ways. The test evaluation can be simply exception based, or more preferably based on some comparison on known-good results. A good test is also capable of creating its own pre-state and cleaning up any temporary files after. The flow of a test mirrors that of normal software testing, consisting of generating or retrieving a pre-state, running the module in question, evaluating the result, cleaning up after the test and reporting results. The test system should be robust enough to be capable of carrying out cleanup even after a failed test.

The most basic way of testing a component is to simply see if it throws exceptions. RPA modules emulating a human's workflow for example, will generally automatically throw exceptions due to missing UI elements, rather than executing forward blindly. Creating these tests requires little time, however they also do not catch all the potential problems with a module. This is also the most common type of test that currently gets created outside the test automation system, since validation and often preparation and cleanup are done manually.

In a more advanced test, the beginning and end states are somehow validated. In a data processing module, this will consist of comparing the output data to known good values. Workflow based modules on the other hand can check if the module under test ends in the right GUI window by either comparing to a known good screenshot or checking if a UI element exists with UiPath's built in tools. Typically, verification testing relies on asserting system status.

As UiPath does not directly support an assert command, a small library of different assert commands was made as custom components. In its first iteration this only included modules AssertTrue and AssertFalse. In the future, this library can be extended when other commonly used test verification methods are identified. Using specific assert commands help both with reducing the effort required for making a test and with standardizing the reporting of results.

Reporting results is done in two different ways. If the test faults due to an unhandled exception, the message and stack trace are recorded. The assert methods also have their own default messages, which can be overridden with more descriptive error messages on demand. For logic errors, having the test result automatically state which result is wrong should speed up debugging.

Generating the pre-state and cleaning up after tests should also be done. Each test should both create its own pre-state and clean up after itself. Unfortunately doing this also has a time cost, for example in the case of running a series of modules that run with a certain piece of software, each of them needs to start it up separately. It is possible to do pre-state generation and post-state cleanup only once in a test suite, but this can be considered a test smell and should be avoided as the previous test result was noticed to affect the results. Both pre-state generation and cleanup should however be minimized, both to save on execution time and to avoid test flakiness. One method for doing this when testing web pages is to navigate directly to the URL under test. Stable test data should also be saved in a separate configuration file, instead of being programmatically generated inside the test, in order to decrease the lines of code taken for testing.

5.12 New workflow

In the new workflow, shown in Figure 16, some elements of CI already exist to facilitate execution of tests, since using CI tools to trigger test suites was deemed the most straightforward solution. Any time a developer pushes local work to the remote repository, the CI tool evaluates the need for a build. If this results in a build with tests involved, the tests are executed and reported together with the build results.

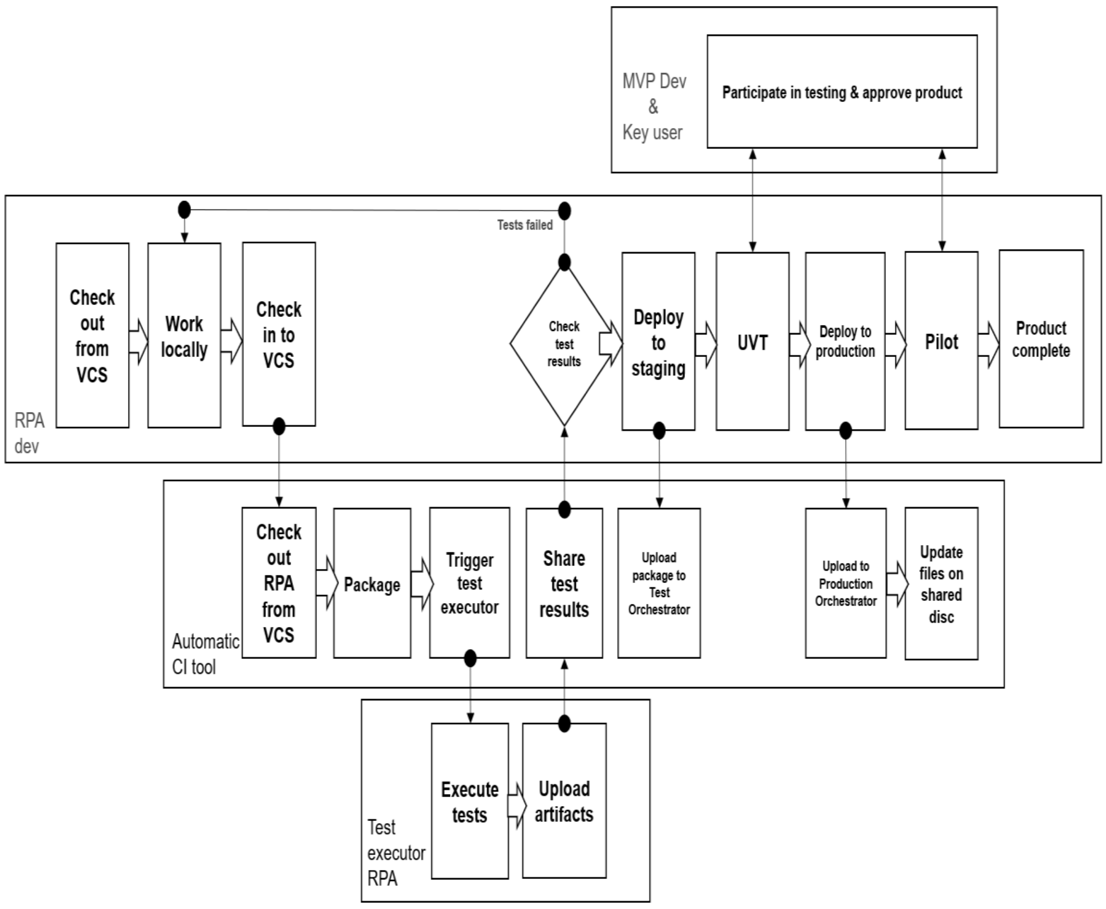


Figure 15. Workflow of the new process. Initially the RPA developer creates a functioning solution on their own workstation. These changes are then checked into version control. It is then the task of the automated CI tool to build the code and task its' testing on the RPA side. The RPA executes the assigned tests and uploads the results back to the CI tool, which is used for sharing the results. If the results are not satisfactory, the process is iterated. If the tests are successful, the RPA is tested with the end user and eventually released to production again using the CI tool.

The improved workflow includes the use of a proper version control tool to facilitate the connection to a CI system. The CI system is then used as the main mechanism for triggering the tests. The designed test runner serves only to take in test files and put out test results while avoiding changes to local files. The CI system is further utilized to more easily share the test results with developers. Further phases of testing, such as the UVT and pilot, should be based on packages that are produced through a testing workflow and are known to be functional.

An alternate way of triggering the local tests is achieved by using MSBuild. Since most of the build process is based on scripts in the repository, any user with the proper access rights can act as an ad-hoc build server. Since one of the main benefits of CI is the usage of a standardized building environment, this should not be used to create production packages (Hilton et al, 2016). However, this can be used to offload some testing tasks directly to test machines without having to rely on the CI tool. This should generally be avoided, since the test results are most difficult to read without the visualization aids given by the CI tool and since using a centralized tool increases traceability for the entire process. This capability should however be preserved as much as practically possible, both to preserve testing capability in cases of outage for the cloud-based CI tool and to avoid platform lock into a single tool.

The tests could in principle be used as a gating mechanism for deployment, where only builds with 100% test success rates can be deployed to production. However, since a significant portion of the tests are flaky, this would create the need for re-executing test suites with no changes made to create a successful build. Like Roseberry's recommendation, test results should be split into two categories: reliable and unreliable. Reliable tests can be safely used as a gating mechanism. The reliability of tests can be evaluated during the active development of an RPA, when the tests are naturally executed more often than in monitoring. Instead of using unreliable tests as an automatic gating mechanism, the tests results should be checked manually to determine if they correspond to deterministic faults. The exception to this rule comes from critical projects, such as with RPA components, which carry a significant amount of risk, where even unreliable tests can be used as gating mechanisms. The tests for critical modules can be considered as gating mechanisms if the extra effort is put in to eliminate flakiness from them.

The structure of the RPA project also changes slightly, with the addition of specific test scheme files to the tests folder, as shown in Figure 16. Additionally, a MSBuild script file to direct the build process specific for the RPA exists elsewhere in the repository.

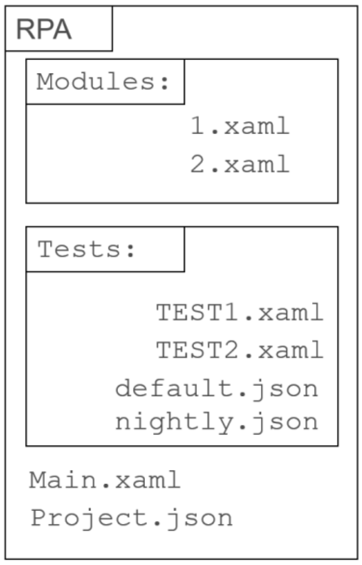


Figure 16. Modified RPA project structure split into the folders `Modules` and `Tests`. The main difference for developers is the test suites defined in `.json` files. The test suite files contain information on what tests to execute during different test runs. In this example, `default.json` represents the test suite that is run when no other suite is specified and `nightly.json` is used for a specific test run only running during the night due to its length.

The infrastructure which the developers use is likewise slightly changed. They interact with development and production robots as before, however the new category of dedicated test robot is interacted with through Azure DevOps and used exclusively for testing. Separating the test and development environments should result in discovery of changes made to system settings on the development side before going to production. This could in turn shorten the time spent on system testing, if these discoveries are then applied to make changes to the production environment.

5.13 A practical case for the new system

One practical use-case of the testing system is the data entry robot described in the introduction. The robot has both a need for regression testing for its data scraping logic and automated integration testing in the interaction with the web-form used for uploading data.

Over time, the data scraping logic has required some updating. This has been both due to slightly changing requirements and missed special cases in the initial development phase. Automated testing has improved the process of adding new features to the data scraper, as any time a change is made to it an automated test suite is triggered. The suite functions based on a bank of known good input-output pairs. An example of a previously scraped dataset is run through the changed module and the results are compared against a previously generated dataset. This is then repeated for a few cases. When a new problem case is identified and the logic is updated to handle it, the case can be added to the regression test suite. This way when changing the parsing logic, the developer can remain confident that past cases do not suffer from a regression.

The process also provides a use-case for automated integration testing. The web-form is periodically updated. Fortunately, the updates are first done to a testing environment and kept there for a time before pushing the changes to production. By having a nightly automated integration test running for this RPA, the developer can be informed of impending difficulties on the production side by automatically monitoring the test environment. In principle the test works in a very similar way to the previous regression test. First, a previously tested dataset is loaded into memory. Second, the module for uploading it through the web-form is executed. Third, the output is monitored for both errors and a known success message.

In both cases, if any errors are detected, the build triggering the test displays errors to developers who can then react accordingly with fixes. The test suite could be expanded in the future to e.g. automatically create new test cases for the regression suite based on past error cases. Likewise, the integration suite could be improved by running the test based on changes in the environment, rather than a nightly repeated test.

6 EVALUATION

This chapter consists of the evaluation of the created testing system and some proposals for the next steps for this project. Since the system was specifically created with the UiPath RPA offering in mind, generalizability towards other RPA products is also briefly discussed.

6.1 Generalizability of results

The automated testing system was created to be used with the UiPath RPA product. However, the same design principles outlined in this thesis could be applied to any other RPA product which is somewhat similar to the UiPath offering.

The main requirement for similarity is the existence of an API to use for triggering RPA processes. Additionally, the test runner RPA would have to be recreated using the specific RPA product. For this to be possible, the program needs to be able to load test modules dynamically or have enough support for automatically generating and deploying packages to be tested.

Beyond those requirements testing any other RPA is going to require the same considerations. Any GUI based interaction is more difficult to test, since the tests take long. In addition, the GUI based modules make up most of any RPA projects. The major difficulties for recreating this project using a different RPA vendor will come from creating a proper testing environment and the long manual process of integrating the provided APIs with the build process.

6.2 Automatic test system

In this chapter the first research question "*How automated testing can be applied to robotic process automation*" is answered. While automating the testing process of RPA is somewhat possible, it is less suited for it than the more traditional programming methods are. This is mostly due to testing being above the unit level. Testing can however be applied to functionality and regression. Additionally, testing can and should be applied to custom components, which can be created using traditional software languages. Some steps can be taken at the design stage of RPA to improve testability.

In its different iterations, both the requirements and features of the testing system evolved. In its initial format, the test runner could execute tests for a project on a specific robot machine. The first iteration confirmed that test times were going to be a problem, which was resolved by giving developers more granular control over test execution. Subsequently, a need for improved debugging of failed tests was discovered, which was in turn responded to with a process for extracting logs and screenshots for tests. In its current iteration, the primary problem is executing tests with varying sets of dependencies. Dynamic dependency handling is an area for future improvement for the test runner.

A constant challenge for using the test system with individual RPAs is managing test execution time. Being aimed mostly at the integration level, these tests take more time than unit tests. Execution time is most easily managed when proper discipline is applied to designing the RPA, keeping modules as small as possible. Creating small workflow modules with the necessary inputs also helps with the reusability of the components, so this should be a high priority, next to stability in any case. Manual testing is also significantly easier with smaller modules.

The automatic testing system is not applied to either the unit level or the system level. On the unit level, the necessary support on the software side is lacking and the modules tend to consist of integration between components pre-created by the RPA provider. System side testing however mostly depends on manual validation by the end users.

A benefit from using automatic testing is evident when applied on reusable components. RPA components created directly through UiPath can be easily tested and problems with them can now be discovered before deploying to production. In addition, with an integration to a mainstream CI tool it is now possible to test and discover problems with the code-only components that are used as a part of the RPA project.

RPA modules can be designed from the very beginning with testability in mind. A well-designed module will focus on doing one thing well, while limiting the amount of inputs it receives. While creating modules, an eye should be kept out for natural starting points for tests. Good starting points are ones that can be reached quickly when a test is started and reliably recreated. For example, when testing a module that interacts with a web page, static URLs such as the home page are good starting points. Generally, the test state generation step should have as few steps as possible in order to be stable. Additionally, while a module that generates its' own data during the test may be more reliable, creating it will take more time. To avoid this, test data should be stored in test specific data files that can be repeatedly accesses. For a module to remain testable over time, it is important that this data is stored together with the code in a shared repository, rather than separately in an external database.

If an RPA has a set of well-constrained functional tests, the test system can also be used for testing changes to the runtime environment and regression testing. Changes in the runtime environment consist of updates to the software being automated. With a clear and fast functional test of the RPA, an updated software version can be automatically validated before pushing to production. When the RPA is changed to fix bugs or handle new requirements in the runtime environment, the same functional tests can be used for regression testing, to ensure that existing functionality does not break. Having functional tests available results in the capability for doing Continuous Integration.

6.3 Facilitation of Continuous Integration

The second research question "*How test automation can facilitate continuous integration*" was answered mostly through the literature review. In the review, it was determined that test automation is a perquisite for doing CI. However, using tooling meant for facilitating CI had side benefits in facilitating the automatic testing system.

One of the major challenges for the project was the orchestration of test execution. While the workflow for executing a test for a module is relatively straightforward, determining which tests should be executed is a bit more difficult. Scheduling test execution was also an issue. Integrating the automated testing system with Azure DevOps was an easy solution for both problems.

An additional benefit to integrating with an existing DevOps tool, was the possibility of doing continuous integration on the side. Since automated testing is a prerequisite for CI, planning the test system with CI in mind made sense. CI consists of committing changes to a shared repository often and running automated tests on those changes to help avoid builds breaking.

At the time of writing, the impact of using CI with the RPA project is not staggering. The largest immediate benefit is the security which is generated by using a proper version control system. The automated tests connected with changes are necessarily all run on the integration level instead of the unit level, so the feedback cycle for breaking changes is a bit too slow. Commits also do not currently happen frequently. This may also be related to the long time to feedback. However, if the possibility for doing unit testing presents itself, for example with custom components, the capability now exists as a built-in part of the CI system.

6.4 Future prospects and next steps

The testing system can be developed for further use. Some of the development is easily accomplished whereas other goals require significant efforts from both the RPA provider and the RPA developers.

Having some unit testing capability built into UiPath, to be run locally, would be useful. As discussed before, in the case of an RPA unit testing needs to be defined slightly differently to normal software development. If the RPA software had some support for defining test suites to be run for a module to be locally accepted, quality could be increased. Unfortunately, this sort of support cannot be added by the end user, as the RPA design software has no plugin support, and must be added in by the RPA provider.

As a part of providing built-in tools for unit testing, RPA providers should provide code coverage tools for said tests. While the implemented automated integration test system can catch bugs when they happen, it has no notion of the code branches that get executed. If a branch of the code never triggers and causes a bug, this can be at best caught as part of an end case condition. In normal software testing, this may already be caught as the unit testing through the usage of code coverage tools. A tool for this could be created without support from the software provider, however this would require significant work and possibly unstable solutions. This tooling should then be extended beyond the unit level for all relevant parts of testing.

In addition, the usage of a linter is currently not provided by the RPA software. While some IntelliSense support is built in by the RPA provider this helps only in repeating choices made in variable naming but does not enforce naming them correctly in the first place. In a larger team, it would be useful to have built in support for defining style guides for naming variables and modules. This is a functionality that can be added in by an end user for UiPath as a build-time activity, since the software provider uses pre-existing well understood technology for developing their product. However, support from the RPA provider is required to create style guides that could be enforced at design time.

The implementation of continuous integration can also be expanded. Creating and configuring the virtual machines used for executing robots could be improved. If the configuration required for creating robots was saved as a part of the repository, changes to it could be tested as they happen. Additional difficulties now come from sparse commit activity combined with long testing times. While the commits can be made more frequently by retraining developers, reducing testing times is a difficult optimisation problem. If these issues were to be fixed in the future, CI could also be extended with continuous deployment. For continuous deployment to work, the current release process should also be formalised, so that it too can be automated and tested.

7 CONCLUSIONS

The purpose of this thesis work was to create a test automation system for RPA to increase quality and facilitate the use of CICD in the organization.

The research question "*How automated testing can be applied to robotic process automation?*" was answered with a brief literature review on test automation and creating and evaluating a system for this purpose. The resulting system is constrained by limitations in the RPA platform used but is nevertheless thought to be of value to the development process.

The study was carried out in two parts. First, a literature review on topics of RPA, test automation and Continuous Integration (CI) was made. Second, an implementation of test automation was created based on knowledge from the literature review. The results of the implementation were then analysed for the effectiveness and possibilities of future improvements.

In the future the concept of automatic testing of RPAs can be developed in several ways. One way of improving testability for RPAs is to collaborate with the RPA service providers to better integrate testing workflows into their products. This can be done by adding continuous testing and validation to happen in the background as workflows are developed. Creation of the automated tests should also be made easier. This can be achieved by creating better templates and components for creating test pre-states, cleaning up after tests and asserting test function. Methods for automatically creating tests could also be investigated. Finally, combining the already existing testing workflow with setup and validation of robot configuration should be experimented with, as this would help ensure that the runtime environments of new robots are up to necessary standards to be used in production.

In conclusion, the implemented test automation system can execute and report the RPA under test in an environment close to that of a production environment. The system was piloted in an actual project and it was deemed capable of catching errors that would have otherwise needed manual testing to discover. The system could be improved in the future after updates to the RPA platform to include automated unit tests. In the future, the test automation system can be used to increase development speed by utilizing CI/CD methodology at the case company.

REFERENCES

Aguirre, S., & Rodriguez, A. (2017, September). Automation of a business process using robotic process automation (RPA): A case study. In *Workshop on Engineering Applications* (pp. 65-71). Springer, Cham.

Anagnoste, S. (2018). Setting up a robotic process automation center of excellence. *Management Dynamics in the Knowledge Economy*, 6(2), 307-322.

Anagnoste, S. (2018). The road to intelligent automation in the energy sector. *Management Dynamics in the Knowledge Economy*, 6(3), 489-502.

Baltes, S., Knack, J., Anastasiou, D., Tymann, R., & Diehl, S. (2018). (No) Influence of continuous integration on the commit activity in GitHub projects. *arXiv preprint arXiv:1802.08441.*

Baranauskas, G. (2018). Changing patterns in process management and improvement: using RPA and RDA in non-manufacturing organizations. *European Scientific Journal, ESJ*, 14(26), 251.

Baresi, L., & Pezze, M. (2006). An introduction to software testing. *Electronic Notes in Theoretical Computer Science*, 148(1), 89-111.

Bygstad, B. (2017). Generative innovation: a comparison of lightweight and heavyweight IT. *Journal of Information Technology*, 32(2), 180-193.

Elbaum, S., Rothermel, G., & Penix, J. (2014, November). Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (pp. 235-245). ACM.

Fung, H. P. (2014). Criteria, use cases and effects of information technology process automation (ITPA). *Adv Robot Autom*, 3(124), 2.

Gaur J., Goyal A., Choudhury T. and Sabitha S., A walk through of software testing techniques, 2016 *International Conference System Modeling & Advancement in Research Trends (SMART),* Moradabad, 2016, pp. 103-108.

Garousi, V., Kucuk, B., & Felderer, M. (2018). What we know about smells in software test code. *IEEE Software Volume: 36 , Issue: 3 , May-June 2019*

Gligoric, M., Eloussi, L., & Marinov, D. (2015, July). Practical regression test selection with dynamic file dependencies. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (pp. 211-222). ACM.

Gligoric, M., Negara, S., Legunsen, O., & Marinov, D. (2014, September). An empirical evaluation and comparison of manual and automated test selection. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering* (pp. 361-372). ACM.

Hilton, M., Tunnell, T., Huang, K., Marinov, D., & Dig, D. (2016, August). Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (pp. 426-437). ACM.

Hilton, M., Nelson, N., Dig, D., Tunnell, T., & Marinov, D. (2016). Continuous integration (CI) needs and wishes for developers of proprietary code. Corvallis, OR: Oregon State University, Dept. of Computer Science.

Hooda, I., & Chhillar, R. S. (2015). Software test process, testing types and techniques. *International Journal of Computer Applications,* 111(13).

IBM Website, JUnit XML Format, [Online, cited on 01.04.2019, available at: https://www.ibm.com/support/knowledgecenter/en/SSUFAU_1.0.0/com.ibm.rsar.analysis.codereview.cobol.doc/topics/cac_useresults_junit.html]

Issac R., R. Muni & K. Desai, Delineated analysis of robotic process automation tools, 2018 *Second International Conference on Advances in Electronics, Computers and Communications (ICAECC)*, Bangalore, 2018, pp. 1-5.

Lacity, M., Willcocks, L. P., & Craig, A. (2015). Robotic process automation at Telefónica O2. London School of Economics and Political Science, LSE Library.

Moe, N. B., Cruzes, D., Dybå, T., & Mikkelsen, E. (2015, July). Continuous software testing in a globally distributed project. In *2015 IEEE 10th International Conference on Global Software Engineering* (pp. 130-134). IEEE.

Moffitt, K. C., Rozario, A. M., & Vasarhelyi, M. A. (2018). Robotic process automation for auditing. *Journal of Emerging Technologies in Accounting*, 15(1), 1-10.

Peffers, K., Tuunanen, T., Rothenberger, M. A., & Chatterjee, S. (2007). A design science research methodology for information systems research. *Journal of Management Information Systems*, 24(3), 45-77.

Penttinen, E., Asatiani A. & Kasslin, H. (2018). How to choose between robotic process automation and back-end system automation, Conference: *European Conference on Information Systems*, At Portsmouth, United Kingdom. 14 p.

Pinto, G., Rebouças, M., & Castor, F. (2017, May). Inadequate testing, time pressure, and (over) confidence: a tale of continuous integration users. In *Proceedings of the 10th International Workshop on Cooperative and Human Aspects of Software Engineering* (pp. 74-77). IEEE Press.

Rafi, D. M., Moses, K. R. K., Petersen, K., & Mäntylä, M. V. (2012, June). Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. In *Proceedings of the 7th International Workshop on Automation of Software Test* (pp. 36-42). IEEE Press

Roseberry, W. M. (2016). Winning with Flaky Test Automation. [Online, cited 03.10.208, available at: http://uploads.pnsqc.org/2016/papers/12.WinningWithFlakyTestAutomation.pdf]

Stople, A., Steinsund, H., Iden, J. & Bygstad, B.: Lightweight IT and the IT function: Experiences from robotic process automation in a Norwegian bank (2017). Paper presented at NOKOBIT 2017, Oslo, 27-29 Nov. NOKOBIT, vol. 25, no. 1, Bibsys Open Journal Systems, ISSN 1894-7719.

UiPath website, www.uipath.com, cited on 20.11.2018

Vaishnavi, V., Kuechler, W., and Petter, S. (Eds.) (2004/17). Design Science Research in Information Systems, January 20, 2004 (created in 2004 and updated until 2015 by Vaishnavi, V. and Kuechler, W.); last updated (by Vaishnavi, V. and Petter, S.), December 20, 2017. [Online, cited 05.10.2018, available at: http://www.desrist.org/design-research-in-information-systems/.]

Wiklund, K., Eldh, S., Sundmark, D., & Lundqvist, K. (2017). Impediments for software test automation: A systematic literature review. *Software Testing, Verification and Reliability*, 27(8), e1639.