# Point of Sale Test Automation

Ravi Autar
Takang Kajikaw Etta Tabe
Adam el Khalki
Ali Smesseim


TU Delft

# Point of Sale Test Automation

by

Ravi Autar
Takang Kajikaw Etta Tabe
Adam el Khalki
Ali Smesseim

to obtain the degree of Bachelor of Science
at the Delft University of Technology.

| | | |
|---|---|---|
| Project duration: | April 25, 2017 – July 3, 2017 | |
| Project committee: | Huijuan Wang, | TU Delft, coordinator |
| | Otto Visser, | TU Delft, coordinator |
| | Maurício Aniche, | TU Delft, coach |
| | Bert Wolters, | Adyen, company contact |

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

adyen

# Foreword

This thesis is the result of a graduation project of over 2 months to obtain the degree of Bachelor of Science in Computer Science and Engineering. This work represented the software engineering methods learnt during the bachelor quite well. Furthermore a lot of was learned about the payment industry, point of sale solutions, and tackling real world problems.

First of all, we want to thank Mauricio Aniche and Bert Wolters for providing us with this great opportunity to tackle a real world problem and for supervising this project. We would them for their guidance throughout this project from an academic point of view. Their weekly, if not daily feedback was very constructive and helpful for the development of our final product.

We also want to thank Maikel Lobbezoo and Lisanne van Kessel, for reaching out to us during the Delfste Bedrijvendagen, eventually introducing us to the company and making sure that our time at the company was a good and comfortable one.

Next, we would like to thank Rick Wieman, Eduard Amzand, An Pan, Alexandros Moraitis, Victor Dragan and Miguel Suarez for giving us a space in the testing team and for giving open ears to our daily cries. Thanks for giving this project the value it has and for always making time out to answer our sometimes irrelevant questions. Big thanks to the guys who gave us a space and made it for the daily pre and post lunch games at the foosball table.

Last but not the least, our thanks goes out to all the amazing ladies and gentlemen at Adyen who helped us in one way or the other during the course of this project. We appreciate their kindness abundantly.

# Contents

# 1 Introduction

Adyen is a technology company based out of Amsterdam, that offers payment solutions to merchants all over the world. Adyen's platform incorporates over 250 payment methods and 187 transaction currencies. Over 4,500 businesses use Adyen's services, which include e-commerce, in-app payments, fraud detection, and Point of Sale (POS) solutions. POS mainly concerns itself with in-store payments, where a customer performs transactions on a payment terminal. The software Adyen provides in this case are the firmware that runs on the terminal, as well as the software that communicates with the terminal and is integrated in the cash register system. All software can contain faults, which may lead to bugs. In this case, it is no different. To detect the possible faults, Adyen tests their software thoroughly. Software testing can either be done by hand, or preferably be automated by a computer.

Testing terminal firmware brings a challenge. The terminal firmware runs on a physical terminal, and testing involves physically interacting with that terminal. A human could easily press buttons and perform other types of interactions with the physical payment terminal to verify that the firmware on the terminal is working correctly. A computer on the other hand cannot do this as easily. A computer would need some sort of physical component to interact with the terminal; a robot arm. This is exactly what Adyen uses to automate their POS testing process.

Adyen has a state of the art testing arrangement. It uses several robots to test the terminal firmware. This is much more efficient than testing the terminals manually. The robots can work longer than humans, and are less error-prone. This arrangement was such a success that Adyen increased the number of robots, from a couple of robots to over a dozen. This makes it possible to test more scenarios in less time.

Adyen developed an application, *TerminalTester*, to control the robots and to verify whether there are failures in the terminal firmware. *TerminalTester* was initially designed to control a single robot; it cannot control multiple robots. To work around this limitation, Adyen runs an instance of *TerminalTester* for each robot. This approach is not scalable, as each instance requires actions from the POS testing team. Many of these actions, however, are the same for every instance. This leads to duplicated effort to manage the instances. Also, *TerminalTester* has been in development for approximately three years. In this time period, it amassed much technical debt, which makes it harder to implement new features. Furthermore, the language used to define tests is quite convoluted, which makes it hard to write new tests and to understand the written tests.

For this bachelor's end project, our goal was to improve the testing arrangement by developing a new application that tests the terminal, as a replacement for *TerminalTester*. This new application should be as robust as *TerminalTester*, but unlike *TerminalTester*, be extenisble and scalable. Also, the current test definition language should be replaced with a language that is easier to read and write.

This report describes the development of our new application. Chapter 2 gives a more elaborate background on Adyen and the current testing arrangement. Chapter 3 describes the problems of this testing set up, as well as the requirements and product vision of the new application. Chapter 4 details the implementation of the new application. Chapter 5 outlines the development process and how the code quality was upheld. Chapter 6 discusses the achieved results and describes the ethical implications of our work. Finally, Chapter 7 concludes our report.

# 2

# Background

This chapter offers some key insights into the background of this project. Adyen is the industry partner for this project, thus it is essential to learn more about them to understand their needs and their point of view. Also, the current testing arrangement is described in detail.

## 2.1. Industry Partner

Adyen is a payment service provider (PSP) that offers merchants online services for accepting electronic payments. Adyen offers multiple payment methods that are integrated into one payment gateway. This gateway is used by merchants to make themselves less dependent on financial institutions (i.e. banks). The payment landscape provided by Adyen can be divided into two main channels: E-commerce and Point of Sale (POS). E-commerce encompasses the payments that are done online (e.g. buying a Foosball table on online marketplace). POS on the other hand, encompasses the payments that are done in physical shops (e.g. buying a Foosball table in a physical store).

Despite the challenging and interesting aspects of e-commerce [16], this project only focuses on the POS payment method. Therefore, this section is dedicated to introduce and familiarize some key concepts regarding the POS payment method.

### 2.1.1. PED

A PIN entry device (PED), also referred to as (payment) terminal, is an electronic device that provides both a physical and digital interface. The physical interface can be used by a shopper to physically interact with the terminal in order to conduct a payment, whereas the digital interface provides a digital interaction with the terminal (e.g. using a cash register). Adyen does not manufacture the terminals; it only provides the firmware that is used by the terminals.

**Physical interface**

The physical interface is commonly used by the shopper to conduct a transaction issued by the merchant. This interaction can be done using the following methods:

- **Touchscreen:** some of the terminals have a touchscreen that can be used to provide the terminal with inputs, such as a signature or menu navigation.
- **Buttons:** every terminal offered by Adyen has physical buttons. These buttons can be used for various purposes, e.g. PIN code entry and menu navigation.
- **Contactless Chip Reader (CLESS_CHIP):** all the terminal have a contactless card reader, which allows the use of NFC as a payment method.
- **Integrated Circuit Card (ICC) reader:** the ICC reader is also supported on every terminal. This method refers to conducting a payment by physically inserting the payment card into the terminal.
- **Magnetic Stripe Reader (MSR):** the MSR is used to conduct a payment by swiping the card on the terminal, and is a widely-used payment method.

- **Receipt printer:** some of the terminals have a physical printer, that is used to print receipts. Other terminals send instructions to the cash register application to print the receipt using an external printer.

**Digital interface**
The terminal can be run in two modes: standalone or integrated. A transaction can be started on the standalone terminal using the physical interface mentioned earlier. The merchant enters the amount that has to be paid and hands over the terminal to the shopper, who can then perform the transaction. The integrated terminal on the other hand utilizes the digital interface of the terminal to communicate with the cash register using Adyen's POS libraries. The cash register software can use the POS library to start or cancel a transaction. Some interactions that are possible using the digital interface on the terminals are:

- creating a transaction using a number of attributes (e.g. amount and currency);
- cancelling an ongoing transaction;
- adding a cash balance to a gift card, or using the balance on the gift card for a payment;
- checking the balance on a gift card.

### 2.1.2. POS libraries
Adyen provides POS libraries to allow the cash register systems and the terminals to communicate. There can be many different cash register environments. To support a variety of these environments, Adyen provides six libraries. The Adyen libraries expose the terminal's digital interface to the POS software. The library also exposes some additional functionalities, such as registering both the terminal and the POS system to Adyen backend.

### 2.1.3. Adyen backend
The POS system sends payments that need to be processed to the Adyen backend using the POS library. On the backend, the money is transferred from one account to another (this is typically from the shopper's account to the merchant's account). An overview of information such as payments and refunds can thus be found in the backend. The backend is also used for scheduling software updates on the terminals.

### 2.1.4. Continuous integration
Subversion (SVN) is used by Adyen to maintain current and historical versions of their code base. Every four weeks, a new version of the terminal software is released. Two weeks after every terminal software release, a new version of the POS libraries is released. Adyen uses Jenkins to automate some of their building and testing.

### 2.1.5. POS testing
As the payment industry is a very secure world, testing is essential. Therefore, Adyen has a separate department for testing software which is referred to as the POS testing team. This team is separate from the POS development team, which focuses on the development of software for the POS domain. The testing is currently done in an end-to-end fashion. The terminal software as well as the different libraries are tested using robots. The next section will further elaborate on the current testing arrangement used by Adyen for these end-to-end tests, as testing the terminals is the main focus of this project.

## 2.2. Current testing arrangement
Adyen provides the firmware for the terminals, as well as the POS libraries to provide the digital interface. Both of these components have to be tested by the POS testing team. At the beginning of this project, Adyen already had its own arrangement for testing terminal firmware and libraries. This section dives further into the specifics of the technology used by Adyen, and also discusses the current workflow with which Adyen operates.

### 2.2.1. Testing the libraries
The first vector which has to be tested by the POS testing team, is the POS libraries which provide the digital interface for the terminal. Adyen provides six different POS library types. These libraries are then integrated

into the cash register system by the merchant. The merchants may have different environments for their cash registers (e.g. different operating systems). Therefore, Adyen needs to provide the different library types in order to support these environments. Adyen also supports running the terminal firmware without a POS library; this is called the standalone mode which is already discussed in Section 2.1.1.

### 2.2.2. Testing the firmware

Adyen provides terminal firmware for a variety of terminal models. All of these terminals are manufactured by Verifone, and Verifone also provides the terminal's operating system. There are two operating systems used by the terminals, namely EVO (Verifone Evolution) and VOS (Verifone Operating System), of which VOS is newer. Adyen therefore has to maintain two code bases for the terminal firmware: one written for EVO terminals, and one written for VOS terminals. Although there are only two code bases for the terminal firmware, it does not suffice to run a test suite only on a certain EVO terminal and a certain VOS terminal to verify that the terminal firmware has the correct behavior for both code bases, as a specific model might show failures. The terminal models can have differing capabilities; for example, some terminal models may not have built-in printers. Other terminals may have a touchscreen, which can also expose bugs. Table 2.1 shows a matrix of the testing combinations. Not all combinations are possible however; libraries may not support all connectivities between terminal and cash register. The possible combinations, and therefore also the combinations which should be tested, are marked in this table.

### 2.2.3. Using robots

As indicated in table 2.1, the number of combination that should be tested are quite large. Therefore, Adyen has employed robots to test the combinations of terminals and libraries (hereinafter simply called "terminal testing"). As of June 2017, Adyen has fifteen robots testing the terminals. However, such an elaborate arrangement had not been the plan from the start. Automated terminal testing started with a single robot. More robots have been added gradually. Of course, robots need to be controlled in order to execute a test suite, which was the responsibility of the aptly named *TerminalTester* application. *TerminalTester* started a series of transactions, and guided the robot to execute these transactions, and afterwards verified that what the terminal had done was actually the correct behavior.

| PED | | | | | POS | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Model | OS | Printer | Touchscreen | Connectivity | Android | iOS | .NET | COM | JNI | C | Standalone |
| E315m | EVO | no | no | Serial | | x | | | | | |
| E335 with barcode scanner | EVO | no | no | WiFi | x | x | x | x | x | x | |
| | | | | Bluetooth | x | | | | | | |
| | | | | Serial | | x | | | | | |
| E355 without barcode scanner | EVO | no | no | WiFi | x | x | x | x | x | x | x |
| | | | | Bluetooth | x | | | | | | |
| | | | | Serial | | x | | | | | |
| VX675WIFIBT | EVO | yes | no | WiFi | x | x | x | x | x | x | x |
| | | | | Bluetooth | x | | | | | | |
| VX680 | EVO | yes | yes | WiFi | x | x | x | x | x | x | x |
| | | | | Bluetooth | x | | | | | | |
| VX690 | EVO | yes | yes | WiFi | x | x | x | x | x | x | x |
| | | | | Bluetooth | x | | | | | | |
| | | | | Ethernet | x | x | x | x | x | x | x |
| | | | | 3G | | | | | | | x |
| VX820 with Duet | EVO | yes | yes | Ethernet | x | x | x | x | x | x | x |
| | | | | Serial | | | x | x | x | x | |
| VX820 without Duet | EVO | no | yes | Ethernet | x | x | x | x | x | x | |
| | | | | Serial | | | x | x | x | x | |
| MX915 | VOS | no | yes | Ethernet | x | x | x | x | x | x | |
| | | | | Serial | | | x | x | x | x | |
| MX925 | VOS | no | yes | Ethernet | x | x | x | x | x | x | |
| | | | | Serial | | | x | x | x | x | |
| UX300 | VOS | no | no | Ethernet | x | x | x | x | x | x | |

Table 2.1: Matrix of the possible POS and PED arrangements. The marked cells are the combinations that Adyen's POS testing would like to test.

### 2.2.4. Workflow

A typical development workflow for a POS library is as follows:

**1.** Developers communicate the requirements of the new POS library version to the testing team.

**2.** The POS testing team devises test cases and writes test definitions if they can be expressed in *Terminal-Tester*'s test definition language.

3. Developers make adjustments to the library.

**4.** Developers push the adjustments to the SVN repository.

**5.** Developers send the build of the new POS library version to the POS testing team.

**6.** The POS testing team deploys the new POS library build to each robot computer. A robot computer is a computer that runs an instance of *TerminalTester* and is connected to a robot. Deployment consists of copying the build to a USB stick, inserting the USB stick into each robot computer, and copying the build from the USB stick to the robot computer.

**7.** The POS testing team runs the test suite by starting the *TerminalTester* application on each individual robot computer.

8. The POS testing team communicates the test results back to the developers after the test suite has been completed, and guides them with reproducing the bug locally on their machines. If the bug is severe enough (and during a two-week test cycle, it generally it is), then the developers must fix the bug, and this cycle starts again at step 3.

The steps in bold are shown as a sequence diagram in Figure 2.1. This diagram shows clearly that running the tests and the deployment are repetitive tasks that take up a significant amount of time. A diagram showing the relationships between entities is shown in Figure 2.2. Figure 2.2 also shows clearly that, while Adyen makes use of a build automation server (Jenkins), the POS builds are not automatically tested. In fact, as of June 2017, not all POS libraries are even built on Jenkins. However, Adyen is working on building every type of library on Jenkins. Due to the current testing arrangement, it is difficult to test the builds automatically via Jenkins, as Jenkins and the robot computers are not linked. Jenkins is therefore not able to supply the builds to the robot computers.

The other vector, terminal firmware, is tested in a similar fashion. The same steps are followed, with the main difference between deploying POS libraries and terminal firmware is that terminal firmware is deployed to the terminals, and POS libraries are deployed to the robot computers.

The POS testing team has to manually deploy the terminal's firmware on the terminal if a different build was to be tested, using a serial port. This of course was no hindrance if there is only one terminal used for automatic testing. As the number of robots increased, using the robots optimally became harder. This has to do with the manual interactions required to run a test suite.

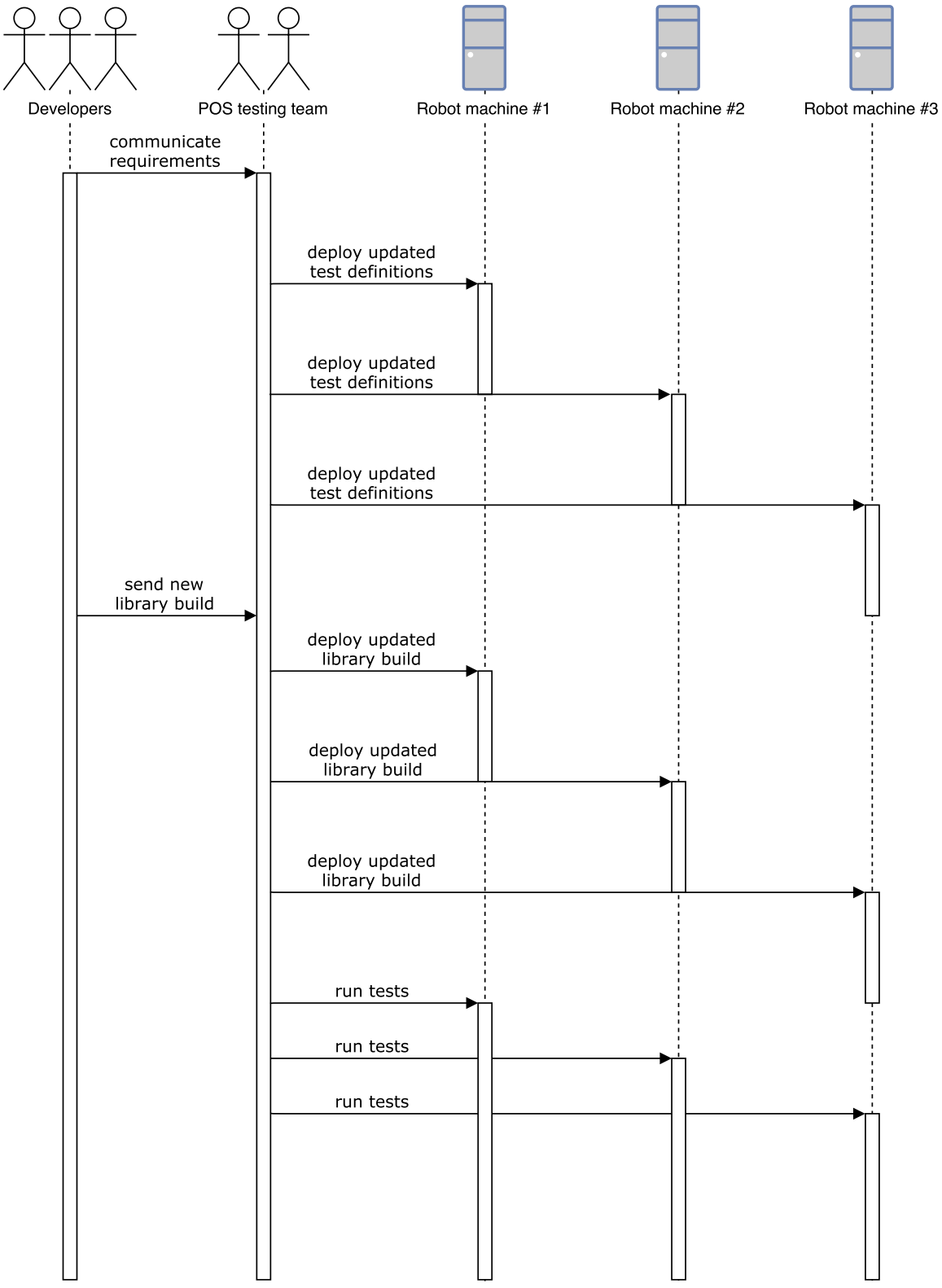


Figure 2.1: Sequence diagram of the deployment of a new POS library version. This diagram assumes that the new POS library version differs in behavior from the older versions, thus the new requirements have to be communicated. Three robot computers are used in this diagram, but Adyen already employs fifteen robots as of June 2017.

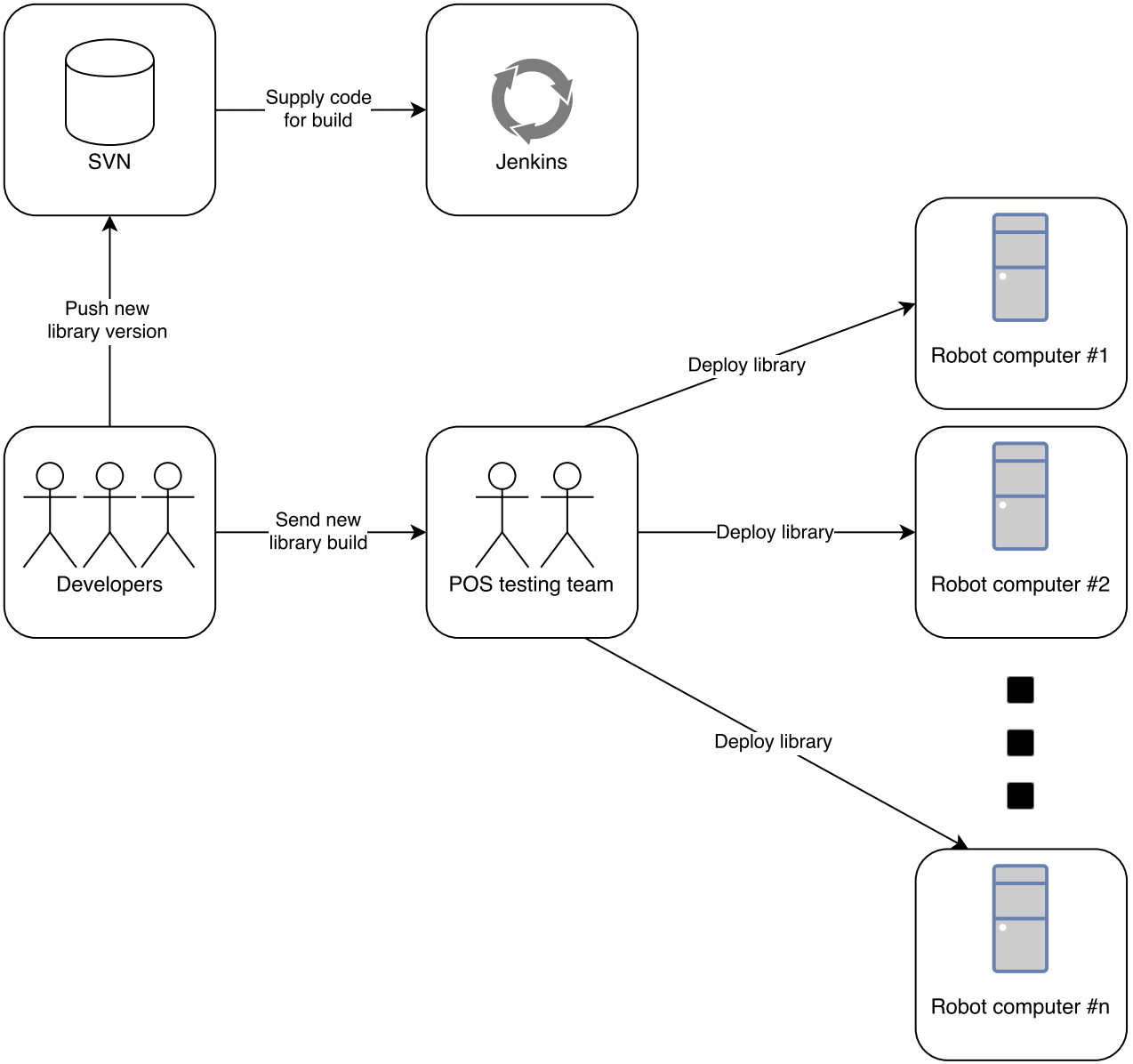


Figure 2.2: Schematic diagram of the deployment of a new POS library version. Arrows stand for the interaction between entities. The entity from which the arrow departs initiates the interaction.

# 3

# Problem Definition and Requirements

This chapter defines the main problems encountered by Adyen, and also provides an in-depth analysis of these problems. This chapter also discusses the goals of this project as well as the product vision.

## 3.1. Problem definition and analysis

As discussed in the previous chapter, Adyen offers an in-store payment solution which includes the use of a terminal. The terminals run custom-made Adyen firmware and communicate using a digital interface made possible by POS libraries. The combinations of terminal firmware and POS libraries are tested using robots which are controlled by *TerminalTester*. Adyen supports many different device models which all behave slightly differently. In order to test the these different models simultaneously, Adyen has increased the number of robots used for testing. However, *TerminalTester* was not designed to support multiple robots, which gives rise to certain limitations in the testing environment. The main focus of this project lies in addressing and resolving most of the limitations encountered by the POS testing team. This section is therefore dedicated to addressing all these limitations and possible solutions.

### 3.1.1. Problems and possible solutions

***TerminalTester* platform dependency and technical debt**

*TerminalTester* is written in .NET, and this leads the application to be platform-dependent. As a result, all robot computers are Windows machines. This is problematic as Adyen's security team views the usage of Windows machines as a security risk compared to the usage of Linux machines. Furthermore, *TerminalTester* has a lot of technical debt, which makes it hard to develop new features for it. For example, tests that engage with the network (i.e. connecting and disconnecting the terminal from the Internet) are difficult to implement due to the technical debt. In order to resolve this problem, the POS testing team had already refactored the *TerminalTester*, but this was not enough to make it scalable. To solve these problem, the POS testing team requested *TerminalTester* to be rewritten, with all the modern needs in mind. This solution needs to be platform-independent. For this reason, Java is used to write the new testing application, as this is a platform-independent language with which the team members have had a significant amount of experience.

**Test definition language**

The scenarios which are to be tested are currently specified in a large XML file. This makes it difficult for the POS testing team to read and understand test scenarios defined in the XML document. This is particular to XML, as the main aim of XML is to structure documents and information, but it gives very little or no interpretation of the information or data contained in the document [10]. This is caused by the very nature of XML which contains duplicate tags (one opening tag and one closing tag) for every element. Furthermore, many test scenarios contain lists of parameters which have to be expressed in XML notation. This is especially difficult, because XML does not provide a simple intuitive notation of lists and thus every element of the list has to be added as a normal element which introduces even more duplicate tags.

Defining new test scenarios is just as difficult as reading and interpreting the test cases due to the complexity of test cases. This is, again, caused by the redundant information which is embedded into the XML

notation, the inefficient representation of data types and the fact that an XML document gives no interpretation of the data it contains. These problems can be solved by defining a new language in which the test scenarios can be defined. The new format should allow for easy reading and interpretations of the test scenarios, as well as to provide a simple way of defining new test scenarios. The solution to this problem is thoroughly researched in Section 4.2.

**Repetitive deployment POS library and firmware**
In order to deploy POS libraries, repetitive steps are to be undertaken by the POS testing team. To test a single library build on every terminal model, it would take approximately forty minutes to complete the deployment of the library build. The deployment time could be reduced if the repetitive steps are eliminated. This can be achieved by deploying the library build in parallel to every robot computer. Jenkins would need to supply the build to ARTS, which can then deploy the build in parallel, as is seen in Figure 3.1.

Repetitive deployment is also present with deploying terminal firmware. The problem may be similar to deploying POS library builds, but the solution is vastly different. While a new application could connect to the POS computers directly to deploy the builds, there can be no such connection to the terminals. Instead, the terminals continuously poll if there is a maintenance job scheduled at the backend. A maintenance job can, among others, be updating the terminal firmware to a specific version. A new terminal testing application could schedule such a maintenance call to deploy the firmware to the terminals. A schematic diagram of this deployment can be seen in Figure 3.2.

**Decentralized testing**
Each robot currently needs its own instance of *TerminalTester* on a dedicated (Windows) machine. This does not scale easily. Each instance requires actions from the POS testing team to run a test suite. Many of these actions, however, are the same for every instance. This leads to duplicated effort to manage the instances. For example, it is required to update *TerminalTester* manually to all the machines. This is also the case when one has to add new test scenarios to the testing application in the form of an XML test definition. Not only does this decentralized arrangement make deployment time-consuming, it also prevents any type of automatic test scheduling. During the day, the POS testing team tests specific library versions on specific terminal models, and provide feedback to the developers with the found bugs. During the night, the POS testing team starts a test suite whose main objective is load testing the terminal. As this test suite is run every day at the end of the working day, it makes sense to automatically schedule this job.

**Developers running tests directly & test suite scheduling** Developers will also make use of the new testing application, with two main goals: to run tests directly and to view test results. The new testing application should provide an interface where developers can choose test suites to run. This is useful in the case where developers are unsure whether some feature still works after the changes of the new library or firmware version are applied. More extensive testing is still left to the POS testing team, who will also need to interact with the new testing application to run the tests.
A summary of the limitations and their solutions are given in Table 3.1.

| Limitation | Solution |
|---|---|
| *TerminalTester* has a lot of technical debt and therefore cannot be extended with new features. | Write new testing application from scratch. |
| *TerminalTester* is written in .NET and is therefore platform-dependent on Windows. | Write new testing application from scratch in Java. |
| *TerminalTester*'s test definition language is convoluted and difficult to read. | Design a new test definition language for new testing application. |
| Deploying a POS library takes a lot of time and requires repetitive actions. | Deploy the POS library automatically via the new testing application. The build is supplied by Jenkins. |
| Deploying a terminal firmware takes a lot of time and requires repetitive actions. | Deploy the terminal firmware automatically via the new testing application. The build is supplied by Jenkins. |
| Only the POS testing team can initiate running the tests. Developers cannot run the tests themselves. | Expose an interface where developers can schedule testing jobs. |
| Test suites cannot be automatically scheduled. | Expose an interface where the POS testing team can schedule automatic testing jobs. |

Table 3.1: Summary of the limitations in the current POS testing arrangement, with each of their proposed solutions.

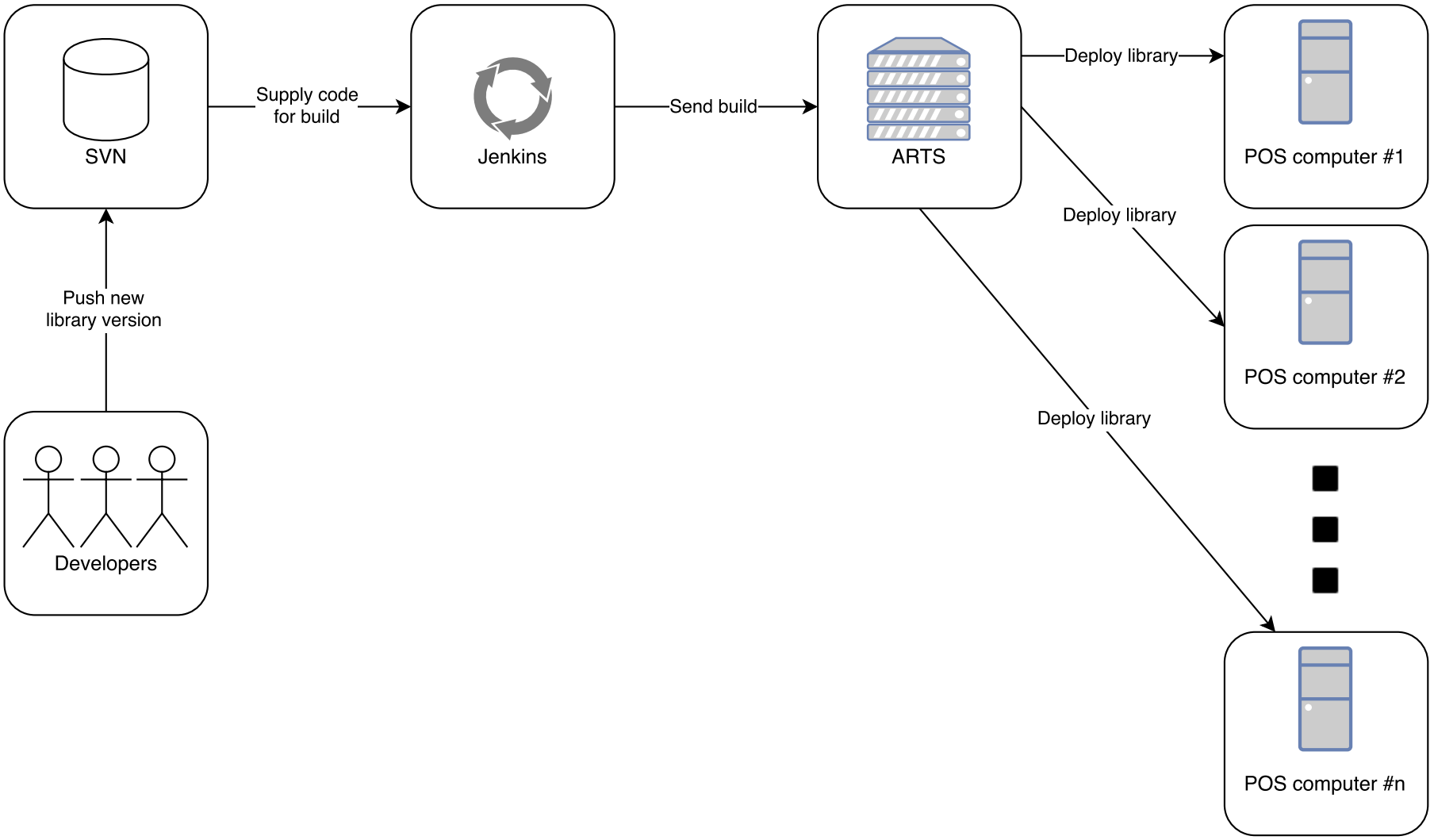


Figure 3.1: Schematic diagram of the deployment of a new POS library version under ARTS.

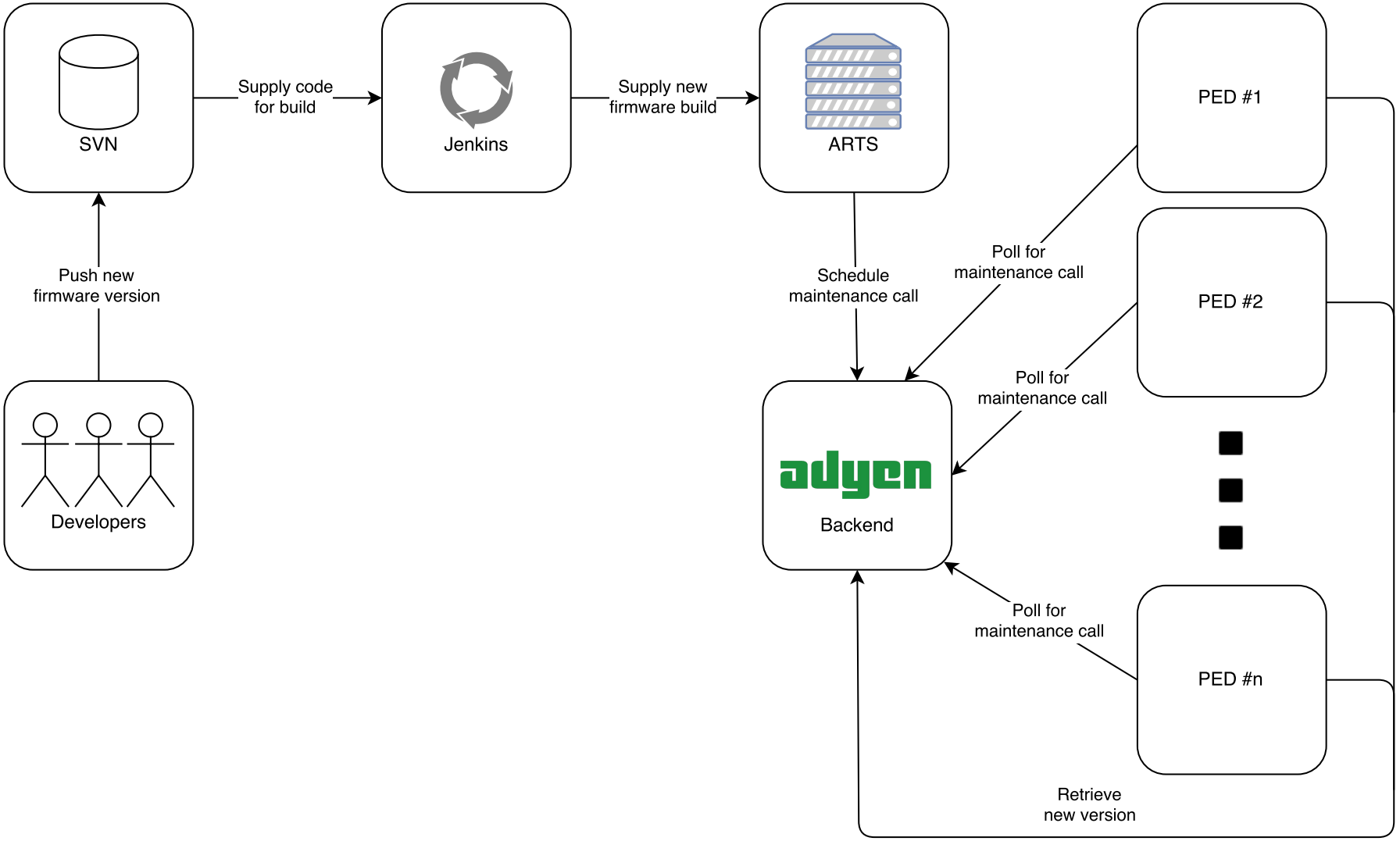


Figure 3.2: Schematic diagram of the deployment of a new terminal firmware version under ARTS.

## 3.2. Requirements

The project in itself was open ended, which means that Adyen gave us some main objectives to develop. We were in agreement with Adyen that the project could not be completed within 10 weeks due to its large scope. After considering our ambitions and the limited time for this project, a list of requirements is discussed in terms of functional requirement and non-functional requirements which are in the scope of this project. This is then followed by a list of requirement which indicate the project vision. The requirements discussed in the project vision should be taking into account while developing the new testing application.

### 3.2.1. In scope

**Functional requirements**

1. The user must be able to write new test cases that are both easy to understand as easier to write than *TerminalTester*'s test definitions.
2. The user should be able to design and add new test scenarios. These test scenarios can be (but are not limited to) controlled network tests (e.g. network outages), power outage tests and user interface tests.
3. The system must easily be extended to support testing new combinations of terminal firmware and POS library.
4. The system must easily be extended to support new types of tests (refer to 4.2).
5. The system must read and execute a test suite defined in a document.
6. The system must evaluate and present the outcome of the each test suite.

**Non-functional requirements**

1. The system will have less technical debt compared *TerminalTester*.
2. The system must be platform-independent.
3. The SCRUM methodology shall be applied for the project.
4. Git shall be used as version control tool.
5. The system should have at least 80% test coverage.

### 3.2.2. Product vision

1. The user shall be able to execute test from a centralized work space.
2. The system shall be able to optimally divide and execute test cases on different robots simultaneously.
3. The system should be able to deploy the POS library automatically via the new testing application.
4. The system could have a user friendly dashboard from which they can define new test cases.
5. The system could have a user friendly dashboard from which they can schedule (automatic) testing jobs.
6. The system could have a user friendly dashboard from which they can view the result from the test run.
7. The system will not run in a Windows environment.

# 4

# Adyen Robot Test Server

This chapter describes the implementation of the new application: Adyen Robot Test Server (commonly shortened to ARTS). This chapter also encompasses the design choices made to implement ARTS.

## 4.1. ARTS architecture

Table 3.1 briefly outlined the approaches ARTS should take in order to tackle all the limitations. The architecture that ARTS employs is described in this section. A requirement of ARTS is that it has to be extensible to allow the addition of new components that are not in the scope of this bachelor's end project.

### 4.1.1. Components

ARTS' structure is divided into four main components, namely:

1. a component responsible for communicating with the terminal(s);
2. a component responsible for communicating with the robot(s);
3. a component responsible for parsing the new test definition language;
4. a component responsible for composing and executing test cases.

Actually, ARTS will eventually contain more components than these four components. More components will eventually be implemented, but this chapter concerns itself solely with the implemented components.

All current test definitions involve starting a transaction (also called a tender), sending commands based on the responses from the POS library, and then verifying the result received from the POS library is equal to the expected result. To emphasize, aside from starting the tender, ARTS performs actions *only* based on messages from the POS library. Verifying the results will be covered later. To gain a better oversight of what messages are sent between ARTS, a POS computer and a robot, a typical test scenario is drawn up. This test case execution can be seen in a sequence diagram format in Figure 4.1, and is further elaborated on below. Message types in italics are responses to the message directly above.

1. **tenderCreate** is sent to the POS library to start the tender. The payment currency, amount, and other options are sent with this request, and whether or not ARTS would like to receive additional data (see step 8).
2. ***tenderCreateResponse*** is received from the POS library, which signifies that the request is successfully received. If the request was valid, the tender will be started on the terminal.
3. **tenderProgress** [**TENDER_CREATED**] is received from the POS library. The tender is started on the terminal and a customer/robot can start with performing the transaction.
4. ***insertCard*** is sent to the robot, so that the robot will insert the card.

5. **tenderProgress [CARD_INSERTED]** is received from the POS library, which signifies that a card has been inserted in the terminal. Note that ARTS receives this feedback from the POS library, and not from the robot. This is convenient; ARTS only has to handle messages from one entity. No action is required now, so this message is ignored.

6. **tenderProgress [WAIT_FOR_APP_SELECTION]** is received from the POS library, which signifies that the terminal has presented an app selection screen, and is waiting for an input from the customer/robot. Some cards support multiple payment methods, and the terminal may ask which of them to select. In this testing environment, Adyen has set up the cards such that there are always four options presented if a card is inserted, and each option has different behaviour (e.g. some of the apps will require the customer/robot to enter a signature later, while the others require a pin code to be entered).

7. ***pressKey(1)*** is sent to the robot so that the robot will select the first app by pressing the key "1" on the terminal.

8. **tenderConfirmAdditionalData** is received from the POS library. The message content is not relevant for this example, other than that this message must by confirmed with a ***tenderConfirmAdditionalDataResponse*** request.

9. ***tenderConfirmAdditionalDataResponse*** is sent to the POS library to confirm the previous message.

10. **tenderProgress [WAIT_FOR_PIN]** is received from the POS library. The terminal displays a PIN code entry screen.

11. ***pressKeySequence(1234)*** is sent to the robot so that it enters the PIN code "1234" on the terminal.

12. **tenderFinal** is received from the POS library. This message signifies that the tender is finished. This message also contains the results of this tender, which can be used by ARTS to verify that the terminal has processed the tender correctly.

13. ***tenderFinalResponse*** is sent to the POS library to confirm that ARTS has received this message.

14. ***goToNeutralPosition*** is sent to the robot to make the arm return to a neutral position above the keypad. This is done to make the robot arm return to a neutral state, after which the next tender can start.

15. ***removeCard*** is sent to the robot. Before this step, the card was still inserted in the terminal. This step removes the card so that the tender is properly finalized.

Using this insight, we can now look further into the individual components.

## 4.2. Adyen Test Definition Language

In order to perform a particular test on the payment terminals, the robots first need the inputs and expected outputs for that particular test. A particular test scenario is called a test case and multiple test cases are contained in what is called a test definition (which may contain the inputs and outputs for multiple tests cases). Such a test definition is dedicated to test a particular feature or scenario on the terminals (e.g. payments that require the customer/robot to enter a signature instead of a PIN code). Furthermore, the entire collection of test definitions is encapsulated in what is called a test suite. The entire test suite is defined in an XML document which is parsed by *TerminalTester* in order to execute tests on the robot. An abbreviated version of an actual test definition used by Adyen, defined in XML, is presented in Appendix B. The semantics of the contents of the test definition will be clarified later on.

During the first iteration of the development phase, the main focus was to devise an alternative way to represent the test suite which was defined in an XML format. This choice stems from the fact that the current XML file used is too large and complex to read and interpret, which would slow down the testing process. This conversion is done by first choosing a new format for the representation of the test suite and subsequently designing the structure of the test suite such that it would comply to Adyen's needs. The new format in which the test suite will be represented is named Adyen Test Definition Language (ATDL).

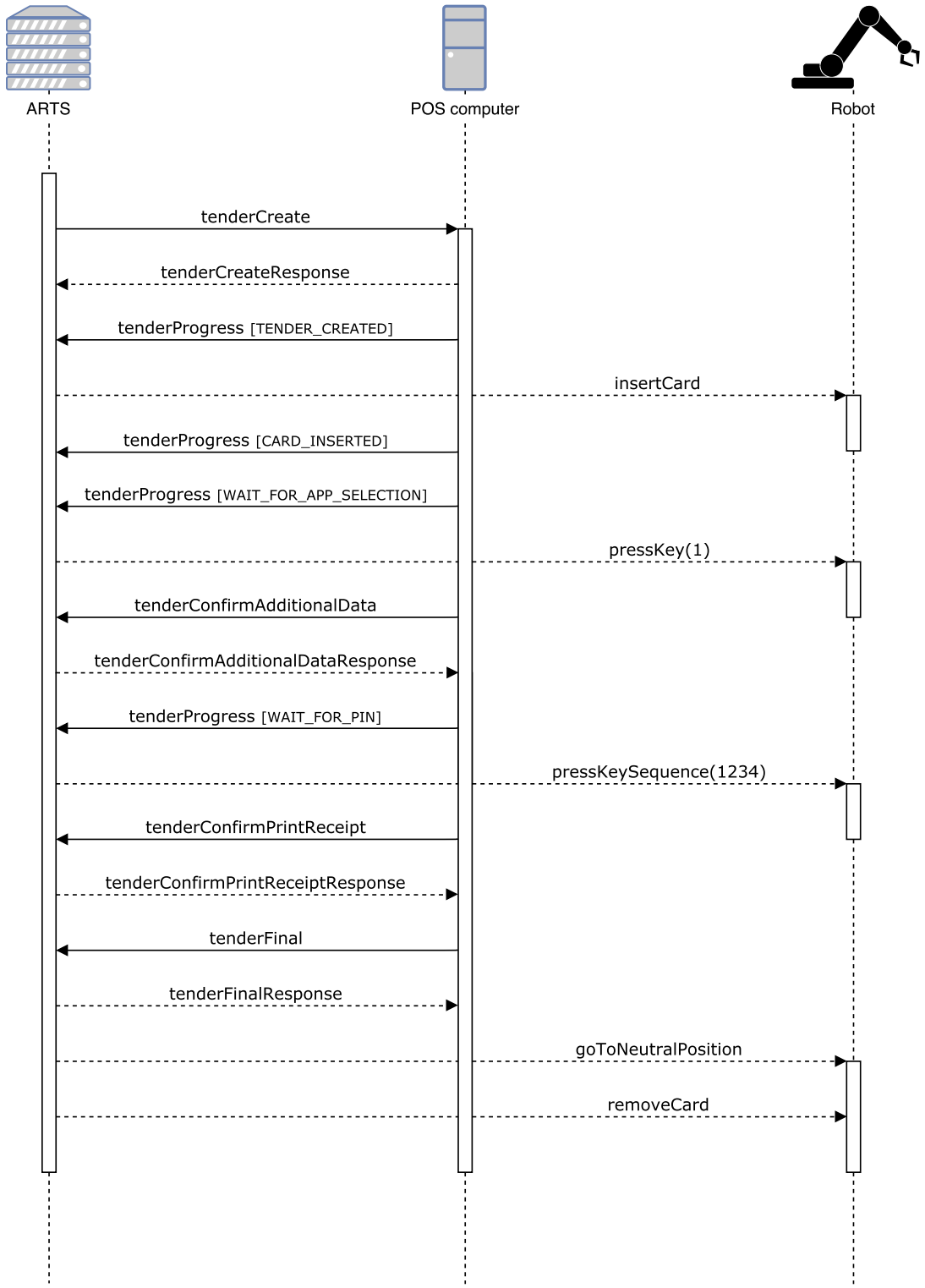


Figure 4.1: An example of a typical test case. The interactions between the POS computer and terminal are omitted. Dashed lines are reactions to the solid-line action directly above.

### 4.2.1. Format Selection

Some quality factors were taken into account during the format-selection process. These factors have to do with the features ATDL has to comply with.

**Readability:** this factor has been the main reason to implement a new representation for test definitions. The size and complexity of the XML document makes it difficult to read the test suite. The following aspects were identified which significantly decreased the readability of the test suite:

**Information redundancy:** the XML solution has a lot of useless information present in the document due to repeated name tags, which in turn decreases the overall readability of the tests. ATDL should therefore have minimal or no redundant tags.

**List representation:** another major drawback of using XML is that there is no efficient way to represent lists. Thus, for large lists, the presence of high amounts of redundant tags is unavoidable. Therefore, ATDL should be able to represent lists in a compact and intuitive way.

**Extensibility:** Adyen would like to add new types of tests (e.g. tests that involve changing network conditions). The XML documents are difficult to read already, although they are homogenous with respect to the test definition type. ATDL should therefore allow the user to define new types of test cases without sacrificing readability.

**Parsing:** a format is preferred for which existing libraries exist to parse it. This would cut down on development time as it would not be necessary to implement a parser from scratch. The external parser would also have to comply to Adyen's security requirements (e.g. the parser library must not have known vulnerabilities).

**Possible formats**

After the research conducted during the format selection process, the following formats were considered as possible alternatives to the XML document.

**XML** is already used to define tests at Adyen. However, by restructuring the XML document, it could be made more compact, readable and extensible. It is also possible to shorten name tags in order to increase the space efficiency [17]. Finding a reliable parser would not be difficult due to the high popularity of the XML format. Using XML also has the added benefit that Adyen would not need to do a lot of adjustments with regards to a new format for the representation of the test suite. The disadvantage to this alternative is that it does not solve the list readability and extensibility problems described earlier. Furthermore, shortening name tags comes at the price of human readability. Therefore, Lawrence [17] proposes JSON as an alternative to XML in terms of space saving.

**JSON** is a lightweight data-interchange format [1]. JSON has a concise way of representing data thanks to its name–value based approach, as apposed to the rather verbose approach of XML [3] [15]. Using JSON would therefore get rid of the problem of redundant information described earlier. JSON also provides an intuitive and space efficient way for defining lists, thereby overcoming the list representation problem. It provides a good solution to the scalability and extensibility problems by defining a correct structure for the test definitions. Also, a variety of libraries are available which can parse data represented in the JSON format. A minor drawback with this format is that it encapsulates inner attributes with brackets. Large test cases in particular would therefore become inconvenient to read because of numerous inner attributes.

**YAML** is a data serialization language designed to be human-friendly and work well with modern programming languages for common everyday tasks [6]. YAML has the design goal of high readability along with support for additional complex features and is mostly used for files meant to be manipulated by humans, such as configuration files [12]. Lists defined in this format are simple and intuitive, and test definitions defined in this format could easily be interpreted due to practically no redundant information present added. By using the YAML format correctly, one can ensure both scalability as well as extensibility. Finally, there are various existing libraries which are capable of parsing documents written in YAML format.

There are several related works which compare these formats in terms of run-time or resource utilization [12] [15]. However, these factors were not taken into account during the selection process. This is because

run-time or resource utilization were not of great importance for this project, as there is no frequent access to the document containing the test definitions. The document is only accessed when ARTS is started. Furthermore, the document containing the test definitions is relatively small for run-time or resource utilization to be a bottleneck.

**Selection**
XML was no longer considered a possibility due to the overhead it introduces by means of duplicate name tags and inconvenient representation of lists. After evaluating the advantages and disadvantages of the remaining alternatives, YAML was chosen as the format to represent the test suite as it provides much better human readability compared to JSON [12]. It is also noted that JSON greatly outperforms YAML in terms of run-time [12], but run-time was not considered to be a major factor which had to be considered in the selection process of the new format. The use of indentations in YAML would make it easier to interpret inner attributes compared to the use of brackets in the JSON format. Furthermore, new types of tests can be easily defined in YAML without having to change the overall structure of the document.

### 4.2.2. Structure design

After selecting a new format for the ATDL, the structure had to be designed in order to meet all the requirements discussed in the previous section. After several deliberations and validations with Adyen, a final structure for the ATDL was determined. This section breaks down the structure of ATDL and discusses the main design choices. The general structure of a test definition in ATDL is given in Example 4.1.

Example 4.1: General structure of ATDL

```
testsuite:
  - name: SomeTestName1 #  1
      [test type]: #  2
        input:
          [input type 1]: #  3
            [input name]: value
          [input type2]: #  3
            [input name]: value
        output: #  4
          - [expected output]
  - name: SomeTestName2 #  1
      [test type]: #  2
        input:
          [input type 1]: #  3
            [input name]: value
          [input type2]: #  3
            [input name]: value
        output: #  4
          - [expected output]
```

The main aspects of ATDL are described below (these points match the numbers in Example 4.1) :

1. **name:** name of the test definition. This indicates the beginning of a new test definition with its inputs and outputs. This name is unique to every test definition and it generally describes the test case.

2. **test type:** type of test definition. The type specified in this field corresponds to the type of the test definition which dictates the format of the input and output for the particular test definition. This attribute allows ARTS to handle different input and output types accordingly, thereby providing a simple interface for introducing new types of tests (e.g network tests).

3. **input type:** the type of input. Test definitions may contain multiple test cases. Therefore, it is necessary to specify different types of inputs that serve different purposes in extracting test cases from such a test definition. This point is further explained in Example 4.2.

4. **output:** the expected output(s) of the test definition. This field contains all the expected outputs for all test cases contained in the test definition and have to be matched accordingly. This will be further clarified in Example 4.2.

Currently, Adyen only uses the parameterized test type for defining test definitions (Example 4.2). However, a new test type may be expressed in ATDL, hence ARTS should support the extension of ATDL with new test types.

**ATDL example**
Following the general structure depicted in Example 4.1, Example 4.2 is constructed which defines a parameterized test definition.

Example 4.2: Parameterized test example

```
---
- name: NonDccTests
  parameterizedtest:
    input:
      standardparameters:
        Currency: EUR
        StartValue: 0
      multivalueparameters:
        TenderOptions: [ReceiptHandler, GetAdditionalData]
      parametersets:
        AmountIndex: [Signature, OfflinePin]
        App: [1, 2]
      dynamicparameters:
        - {toMatch: {AmountIndex: Signature},
          toSet: {StartValue: 251}}
        - {toMatch: {AmountIndex: OfflinePin},
          toSet: {StartValue: 10001}}
    output:
      - {toMatch: {AmountIndex: Signature},
        toVerify: [{Result: Approved}]}
      - {toMatch: {AmountIndex: Signature},
        toVerify: [{Result: Declined}]}
---
```

For Example 4.2, the inputs and outputs follow the semantics below:

- **standardparameters** are parameters that are used for all test cases in this test definition.
- **multiparameters** are parameters with a list value, and are also applicable for all test cases.
- **parametersets** is the input that actually dictates the final number of test cases generated from this test definition. Here, the cartesian product is taken of its individual parameters. The elements of each unique combination is then used as inputs to a test case.
- **dynamicparameters** defines the inputs to test definitions depending on other inputs. The inputs are matched against the "toMatch" argument. If they match, then the parameter(s) in "toSet" are set.
- **output** is divided into parameters that should be matched, and parameters that should be verified (asserted). Similar to the dynamic parameters, the "toMatch" parameters are compared with the input parameters. If they match, then the "toVerify" parameter is compared to the response given by the POS library in order to assert whether the expected response is the same as the actual response.

The test definition above contains 4 unique test cases. These 4 test cases are:

1. **input**: [Currency: EUR, StartAmount: 251, AmountIndex: Signature, App: 1, TenderOption: [ReceiptHandler, GetAdditionalData]]
   **expected output**: [Result: Approved]
2. **input**: [Currency: EUR, StartAmount: 10001, AmountIndex: Offline, App: 1, TenderOption: [ReceiptHandler, GetAdditionalData]]
   **expected output**: [Result: Declined]

3. **input**: [Currency: EUR, StartAmount: 251, AmountIndex: Signature, App: 2, TenderOption: [ReceiptHandler, GetAdditionalData]]
   **expected output**: [Result: Approved]

4. **input**: [Currency: EUR, StartAmount: 10001, AmountIndex: Offline, App: 2, TenderOption: [ReceiptHandler, GetAdditionalData]]
   **expected output**: [Result: Declined]

### 4.2.3. Conclusion

After converting a single test definition from XML to ATDL, the POS testing team agreed that ATDL seemed to fit all their requirements. Using ATDL significantly improves readability compared to the XML format Adyen used previously. After converting the entire test suite from XML to YAML, the total number of lines needed to define the test suite went from 5300 lines to 1100 lines, leading to a total line reduction of approximately 80%.

## 4.3. Adyen Robot Test Server

The previous section defined ATDL and explained its benefits. These benefits would only come to fruition if the test definitions were used to execute tests. This is where ARTS comes into play. Section 4.1 specified the four components which were implemented in ARTS so far, namely: communicating with the terminal and POS library, communicating with the robot, parsing the test definitions, and executing the test definitions. Parsing the test definitions has been covered in the previous section; this section will go through the remaining components. Extensibility is a main theme that will be referred to often in this section. All components have been implemented with extensibility in mind, and the extensibility of the components will also be demonstrated in this section.

### 4.3.1. POS library & terminal connection

All of Adyen's current test definitions involve starting a tender on the terminal, and afterward verifying if the response of the terminal is correct. All of the future test cases will also involve interacting with the terminal, though the forms of these interactions may be manifold. The POS library would therefore need to connect with the terminal and communicate with it reliably. ARTS in turn would connect with the POS library. This way, the combination of POS library and terminal firmware is tested.

ARTS would first need to set up a connection with the POS library. To find out about the possible ways of connecting with the library, it is useful to draw an example of how actual merchants interact with the POS library. A merchant that uses a Windows environment for their cash registers would integrate an applicable POS library. In this case, that would be the COM POS library. The cash register would then programatically call methods in the POS library. The POS library would then form an integral part of the cash register. Replacing the POS library by another applicable one (e.g. C) is not an easy endeavour.

It is also useful to look into how the current *TerminalTester* application communicates with the libraries. *TerminalTester* supports testing all POS libraries, and it makes use of integrating several POS libraries into the application, however not all of them. This is not possible. *TerminalTester* is a .NET application running on Windows, and not all libraries can be run in a Windows environment. Table 4.1 shows how *TerminalTester* interacts with the different types of libraries, and more specifically:

**.NET** and **COM** are libraries specifically made for Windows/.NET, so their APIs can be called directly.

**JNI** is a POS library for the JVM, which *TerminalTester* does not use. To interact with this library, Adyen built a WebSocket server in Java that exposes the JNI API. *TerminalTester* sends WebSocket requests to interact with the library.

**C** is a library that serves as the basis for both the JNI library as the COM library. In fact, these libraries only map JVM and .NET calls respectively to the C library. *TerminalTester* can interact with the C library by means of a REST interface.

**Android** and **iOS** are libraries that run on an Android device and iOS device respectively. These libraries expose their API via a REST server, to which *TerminalTester* sends calls.

| Library | Method of communication | Library instance location |
|---|---|---|
| Android | REST | Android device |
| iOS | REST | iOS device |
| .NET | native calls | robot computer |
| COM | native calls | robot computer |
| JNI | WebSocket | robot computer |
| C | REST | robot computer |

Table 4.1: *TerminalTester*'s methods of communication with all libraries, and the location where the library instance is running.

Hence it could be said that *TerminalTester* makes use of a *hybrid integration* approach. Some libraries are integrated, while the others are connected through REST or WebSocket. From this information, three approaches can be distinguished:

A **pure integration** approach, where every library is integrated into ARTS. This approach is closest to the approach merchants take, hence closest to reality. There can be no inaccuracies/bugs introduced by a WebSocket or REST connection, because there is no such connection. This approach would also lead to the tests being executed faster, as there is no WebSocket or REST overhead. That said, it brings several disadvantages. If one of these libraries is updated, then that update has to be integrated into ARTS, and ARTS in its entirety has to be deployed again to a server. These advantages and disadvantages would not matter any way, as it is impossible because each library cannot run on a single platform.

A **hybrid integration** approach, where only the libraries that can be integrated into ARTS without compromising ARTS' platform independency is integrated. As ARTS is written in Java, this would mean that only the JNI library would be integrated. The libraries that can not be integrated due to the environment ARTS is running in, have to be run on a WebSocket or RESTful server. This introduces some overhead and the testing environments are not consisted, as some libraries are integrated and some are run on different systems.

A **pure WebSocket or REST** approach, where there are no library calls anywhere in ARTS. Instead, ARTS will send calls to a single API to interact with the terminals. There are two popular ways to set up connections of such client-server models within Adyen, namely WebSocket and REST. Both are already used by *TerminalTester*. An important property that distinguishes WebSocket from REST is bidirectional communication. As the execution of test definitons involve acting on responses from the server, it is natural that the server could freely send messages to ARTS, which is only possible in ARTS. Using REST would require ARTS to poll the POS library servers, and REST offers no advantages that offset the lack of bidirectional communication. In any case, the server only maps the server API calls to POS library calls, without introducing any logic itself. A huge advantage of this pure WebSocket implementation is that a POS library can be run on different platforms (e.g. JNI on Windows x86 and x64), while keeping ARTS centralized.

Out of these considerations, a pure WebSocket approach seems to be the best approach. Indeed, ARTS has taken this approach initially. *TerminalTester* already used a WebSocket server to communicate with the JNI POS library. Using this WebSocket server was the easiest option at the start of this project, because it required no knowledge of the internals of the POS library. This WebSocket server proved to be unstable, in the sense that it would frequently stop with sending callbacks. The only remedy was restarting that server. Reasons for this unstability was not the use of WebSocket, but rather the integration of the library. If ARTS were to be a stable system, it had to find an alternative for this WebSocket server.

The decision was then made to integrate the JNI POS library into ARTS. Virtually all of ARTS' assets have been designed with extensibility in mind, and this time it was no different. By the time this decision was made, there was a better understanding of how the POS libraries were used. Integrating the JNI POS library brought more stability to ARTS. Due to the class design, other libraries were still supported through WebSocket (see Figure 4.2).

There was one limitation of integrating the JNI POS library that was not overseen when it was integrated. Each of *TerminalTester*'s test cases involved scenarios where *TerminalTester* communicates with a single terminal through a single POS library. This scenario is consistent with how most merchants use Adyen's POS

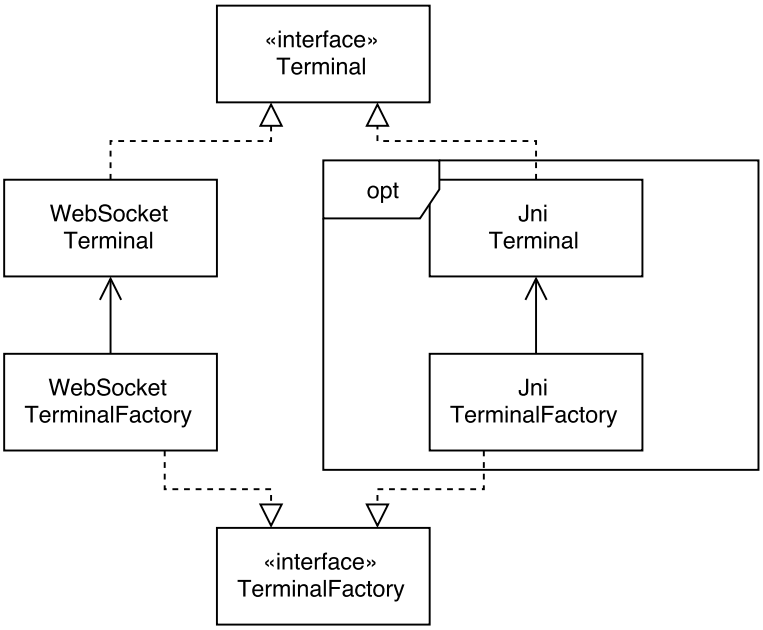


Figure 4.2: Class diagram of the Terminal-related types. JniTerminal was implemented during the course of this project, and later refactored out of ARTS to be a standalone WebSocket server.

services. Each cash register has its own payment terminal. However, Adyen also supports that one POS library that is connected to multiple terminals, or that one payment terminal is connected to multiple POS libraries (many-to-many POS–PED scenario).

Supporting the many-to-many POS–PED scenario would involve ARTS starting multiple instances of the POS library. This was not possible however. The JNI POS library is a wrapper around the C library, and there can be only one instance of the C library on a single machine. The JNI POS library integration thus had to be refactored out of ARTS, so it could run stand-alone on multiple machines. ARTS communicates with the newly created POS library through, unsurprisingly, a WebSocket. Considering that executing a test case requires a callback model (see 4.1), it should be emphasized that the decision to use WebSocket for the communication between ARTS and the POS computer is a great decision, as it allows the server to send messages at any time.

It is important that this communication is implemented in an expandable manner. Refer to 4.3 for a visualization of the implementation. Firstly, it is important to separate ARTS's actions into atomic steps. Each of these TestSteps will form a class. Completely new and vastly different kinds of actions can be created easily. TestSteps only contain an action, but they do not contain the conditions on which their action should be executed. That responsibility is left to the ResponseCallbacks. Whenever ARTS receives a message, this message is passed to each of the ResponseCallbacks, who can then decide to act upon them. Since ARTS currently only receives messages from the POS library and terminal, the only realization of ResponseCallback is TerminalResponseCallback, and this class' responsibility is to deserialize the JSON message received from the POS library and extract its message type (e.g. tenderFinal or tenderProgress). TenderProgressCallback extracts the specific progress type of a tender (e.g. CARD_INSERTED)). The concrete TerminalResponseCallbacks execute their contained TestStep based on a specific message type or progress type. Currently the TestSteps are either requests to the POS library or requests to robot, which are realized by the TerminalTestStep and RobotTestStep respectively.

After such a test case has been executed, the result has to be verified. The results are sent in the tenderFinal message from the POS computer, in the form of key-value pairs in the JSON message received from the server. For each of the relevant key–value pairs (there is more information sent in tenderFinal messages than could be useful), a ResponseCallback is used that adds the specific key–value pair to a map. Verifying test results is simply asserting that each key–value pair in the "toVerify" parameter in ATDL is present in this output map.

Also, there are limitations in the callbacks the terminal can send back to ARTS. Previously, the terminal firmware sent its states to *TerminalTester*. However, there are some test cases where the terminal firmware does not send a state at all (not a bug), whereafter no action can be triggered. Adyen is developing image recognition software that recognizes states using a camera mounted above the terminal.

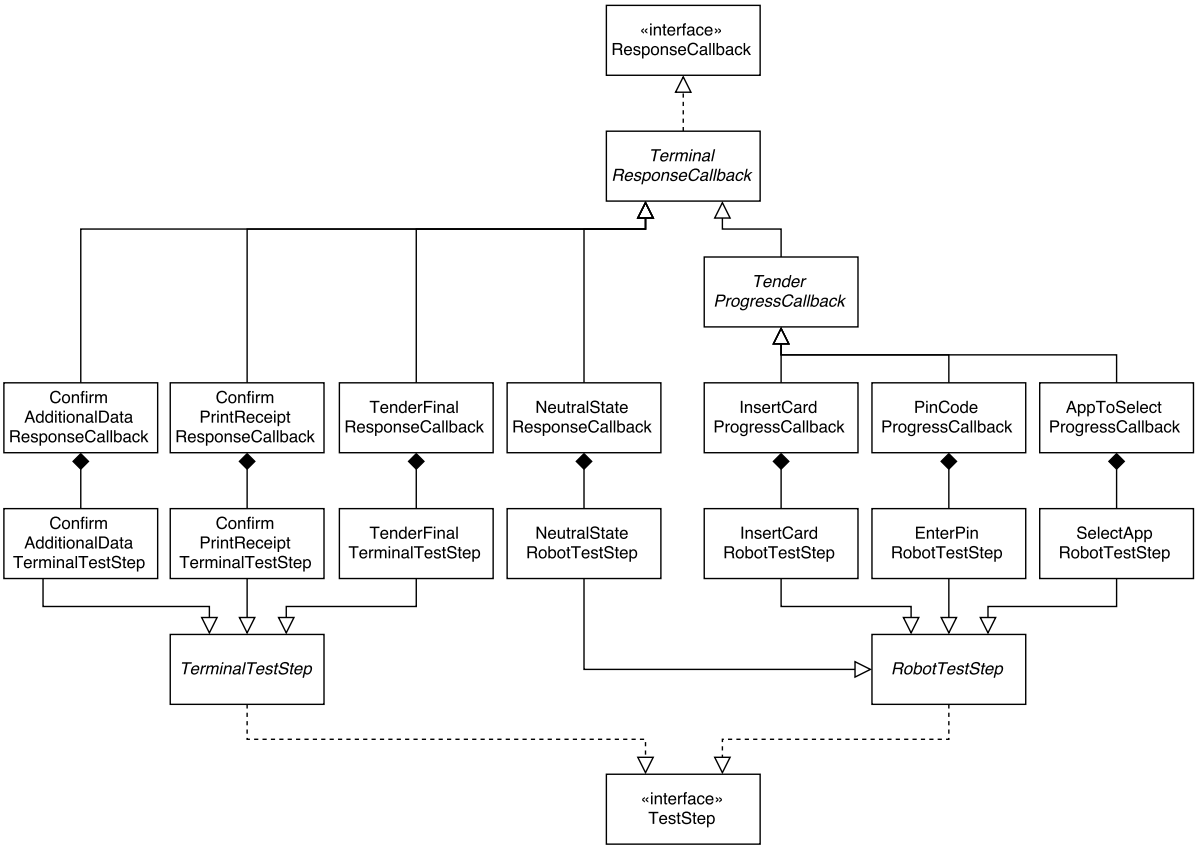


Figure 4.3: An UML diagram with classes responsible for properly communicating with the library in Figure 4.1.

### 4.3.2. Robot connection

One of the huge limitations of the previous *TerminalTester* is the Windows dependency that exists. The robot is controlled by a dynamic-link library (DLL) assembled in .NET and must therefore be called from a .NET application. A third party has ownership over this DLL. Due to time constraints it was not possible to rewrite this code to a shared library for UNIX. As the requirements dictate ARTS must be platform independent, thus a solution had to be found for this problem.

previous
textitTerminalTester calls the .NET DLL that controls the robot, named *TerminalTestRobotApi.dll*. *TerminalTestRobotApi.dll* depends on a COM DLL, named *libdxl_x64_c.dll*. The source code of *libdxl_x64_c.dll* is open-source and could be compiled into different types of libraries [11].

To instruct the robot from ARTS, the first solution tried was building a RESTful server in Windows on top of the DLL. ARTS would then send requests to the Robot Rest Server (RRS). As Windows machines are very rare inside Adyen, the solution was tried on a Windows virtual machine. The solution worked. However, as the notorious WannaCry ransomware was introduced, security disliked having a server running in Windows that could be accessed from any system. This made it impossible to open ports on the Windows virtual machine for the RESTful server.

Mono is a software platform designed to allow developers to easily create cross platform applications part of the .NET Foundation. It allows running of Windows executables in both MacOS as well as Linux. The solution was tried in MacOS using Mono. However in order for the application to run on MacOS, a new *libdxl_x64_c* had to be compiled into a *.dylib*. This is a dynamic library for MacOS. This succeeded and the RRS ran on MacOS. After some testing a huge delay arised between the application and the robot. The DLL used a custom baudrate to communicate over serial with the robot. MacOS does not support custom baudrates.

To by-pass this, the same setup was tried on a Linux virtual machine. The library was compiled into a *.so*. This seemed to work better than the MacOS solution. Nonetheless, the connection was not seamless. The last and final solution was to run the application on a computer that had Linux installed as operating system. It ran smoothly and no problems were found. The robot could now be instructed via a RESTful server from any system.

### 4.3.3. Test definitions

The covered components do not yet form one harmonious whole. ARTS' purpose is to execute test definitions, and there is a process to go through before this purpose can actually be achieved.

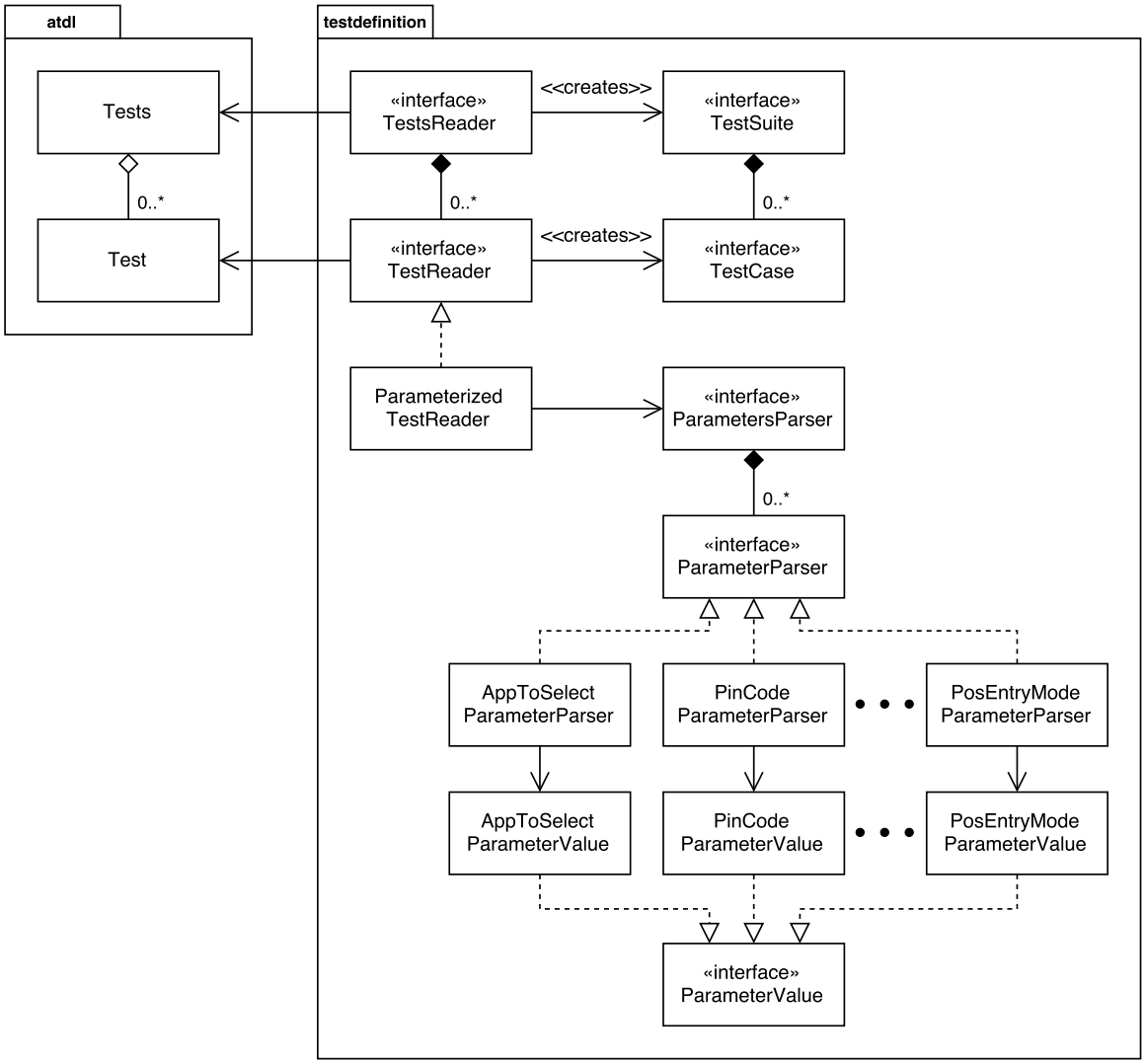


Figure 4.4: Class diagram of test reading.

1. ARTS parses the ATDL test definitions. Here, parsing means serializing ATDL to Java objects, which can then be used to create Java TestCases.
2. The parsed ATDL test definitions could have various types (e.g. parameterizedtest). Each of the test types are read by their own specialized TestReader (e.g. ParameterizedTestReader).
3. As there can be multiple test definition types, the parsed ATDL test definition must be passed to all TestReaders. This is the responsibility of TestsReader. TestsReader contains a multiple TestReaders. TestsReader takes a parsed ATDL test suite, and passes each test to each of the contained TestReaders, and collects their generated TestCases.
4. The TestCases form a TestSuite. This TestSuite is executes every TestCase in order and verifies if the result is correct.

Refer to 4.4 for the classes related to these steps in a UML class diagram. How is expandibility achieved here? The old *TerminalTester* only supports parameterized tests. New types of tests could not be added easily. To address this issue, each type of test has its own TestReader. Each TestReader defines how it turns ATDL test definitions into TestCases. New TestReaders (and therefore also new types of tests) could easily be added.

It is not only important that new types of tests could be added; existing test types must also be expandable. We shall only concern ourselves with the parameterized test type, as that is the only test type currently in use. A distinguishable feature of a parameterized test is that in its expanded form, the test definition only consists of key–value pairs. Each of the key–values may result in a different TestCase behaviour. Take the parameter "PosEntryMode" as an example. "PosEntryMode" determines how the robot should insert the card in the terminal. If "PosEntryMode" would be "ICC", then the robot should insert the card in the terminal slot. If however "PosEntryMode" would be "CLESS_CHIP", then the robot should perform a different action, namely moving the card above the terminal, such that a contactless payment would be initiated. The difference lies in the callback that is added to the terminal.

To effectively encode this behaviour programmatically, it is chosen to encode each value type as its own ParameterValue type. To reuse the example of "PosEntryMode", there is a class PosEntryModeParameterValue that is responsible for adding the correct ResponseCallback to Terminal.

Executing a TestCase means executing a series of TestSteps. These TestSteps cannot simply be executed separated by a fixed time interval, as almost all of these steps are a response to a message from the POS library. Figure 4.3 shows how callbacks can be used to trigger the correct action from ARTS (i.e. sending a message) at the correct time (i.e. after receiving the corresponding message from the POS computer). tenderCreate is the only message that is not sent based on a response from the POS computer. In fact, the tenderCreated message initiates the sequence (as shown in Figure 4.1). Executing a TestCase therefore only means sending tenderCreate to the POS computer. After this step, ARTS receives messages from the POS computer. The callbacks will then make sure that an appropriate action is performed based on the received message.

# 5

# Quality Assurance

When developing an application from scratch, it may be hard to comply with all design principles and to use proper design patterns immediately. The laws of software evolution by Belady and Lehman state that the mess in a software system will increase over time, unless specific actions are performed to combat it [18]. To minimize the mess, proper code reviewing and pair programming should be done. The code should also often be refactored to reduce the mess.

This project has employed several measures to ensure code quality. The extensibility of ARTS was one of its requirements, so upholding the code quality does not only aid the development process, but it also obligatory. This chapter describes the measures we have taken.

## 5.1. Software engineering methods

Good code quality starts of course with writing good code. This section outlines the software engineering methods we have used to write good code.

### SOLID principles

The SOLID principles, as described by Robert C. Martin [19], were used extensively during ARTS' development, to write extensible code.

#### Single Responsibility Principle

The design of ARTS incorporates the single responsibility principle by dividing the responsibility among different classes so that each class has responsibility over a single, atomic part of a functionality. A specific example of this is shown in Figure 4.4. Every class has responsibility over parsing its specific parameter. More information about the behaviour of these classes can be found in Chapter 4.

#### Open Closed Principle

As ARTS will be extended to fulfill the product vision as described in Chapter 1, the Open Closed Principle was a fundamental aspect during development. Classes are designed with the consideration that their behaviour would not be changed, even if the requirements did. If a different behaviour was desireable, then the class should not be changed, but rather a new class should encapsulate the new behaviour. This principle is seamlessly embedded into ARTS. Figure 4.3 shows an example of this principle. The TestStep can be easily extended with a new class and thus new behaviour, however the existing classes can not be modified as their behaviour is atomic.

#### Liskov Substitution Principle

The Liskov Substitution Principle ensures semantic interoperability of types in a hierarchy. An example of this can also be found in Figure 4.3. InsertCardCallback extends TerminalResponseCallback, however InsertCardCallback can be substituted by any other TerminalResponseCallback. Furthermore, the preconditions of InsertCardCallback are not stronger than the preconditions of TerminalResponseCallback and the postconditions of InsertCardCallback are not weaker than those of TerminalResponseCallback.

#### Interface Segregation Principle

All of ARTS' interfaces only have a single responsibility, and are therefore segregated. This is not immediately clear from the UML diagrams in this report, as they do not list the interfaces' methods.

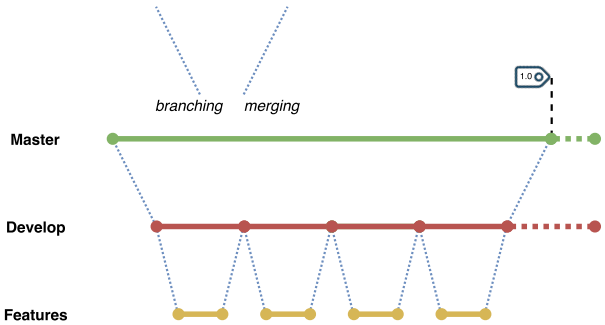


Figure 5.1: Branching model used during development of ARTS.

**Dependency Inversion Principle**
Virtuall all of ARTS' classes depend only on interfaces, and never on implementations. Accordingly, all classes implement interfaces. This makes substituting classes to change ARTS' behaviour trivial. Examples of the use of this principle can be seen in Figure 4.3.

### Developing iteratively
As the requirements of ARTS changed weekly and some desired features were deemed unfeasible, an agile way of developing was used in this project. Sprints lasted one week and contained planning, analysing, designing, developing, testing and delivering the software.

### Controlling changes
As mentioned before, Adyen uses SVN to manage their code changes. Adyen also hosts an internal Gogs Git server, which is similar to GitHub in features and interface. ARTS has been developed on Git for a couple of reasons. The main reason is that all the code projects at TU Delft that we have followed all use Git, and thus we were experienced with its use. Also, the decentralized aspect of Git is very useful for a company like Adyen with over a hundred developers.

ARTS was developed using the branching model shown in figure 5.1. Every feature has its own branch, in which both implementation and testing is done. This is not merely a good practice, this also eases code reviewing as each feature can be reviewed separately. The code reviews would also not involve a massive amount of code.

### Managing requirements
Adyen's Git server supports milestones and issues that were used for managing requirements. During the development of ARTS, the stakeholders were contacted almost daily to discuss the desired features. Every sprint, a milestone was set with main goals. Based on these goals, new tasks (issues) were created to help achieve these goals. On top of that, a physical scrum board was used (see Figure 5.2). The board was an overview of the issues of the current sprint. The notes contain a number which stands for the issue number on the Git server. Color coding was used for both categorizing the issues into ARTS' components: RRS (green), ARTS (yellow), terminal (orange), and ATDL (pink), and for assigning someone to an issue.

### Modeling visually
While designing and implementing ARTS, it was hard to keep an overview of all the different components. To have a better understanding of the component that was being worked on, many drawings were made of the architecture at many times. After many revisions the drawings became the figures shown in Chapter 4.

## 5.2. Verifying quality
Verifying software quality is a good software engineering method, but as mentioned in Chapter 2 security is very important to Adyen. It was not allowed that ARTS' code base would be placed on external machines. This withheld submission of our code to SIG, whose goal it was to measure the code quality of ARTS' code base. The solution was to dedicate a section in our report to the quality of the code and how it is guaranteed.

Verifying code quality can be hard, especially when different developers have different opinions about what defines high quality. However, using the software engineering methods as explained above, design patterns, and refactoring principles that were explained in both the course Software Engineering Methods at the TU Delft and in the book *Refactoring: Improving the Design of Existing Code* by Martin Fowler [13] to design ARTS, the code quality is still of a high level.


Figure 5.2: Physical scrum board used in ARTS' development.

## Code review and pair programming

Nevertheless minor and sometimes major flaws can still exist whilst using the proper methods, patterns, and principles. The cause of this can be lack of experience, or by tunnel vision. During implementation of a feature, the developer can justify bad choices by accepting that what is built is working fine. This can lead to design flaws. As aforementioned the mess in a software system will increase over time, unless there is something done to combat it. To counter the flaws mentioned above, code review and pair programming are key.

According to Alberto Bacchelli,"Peer code review, a manual inspection of source code by developers other than the author, is recognized as a valuable tool for reducing software defects and improving the quality of software projects" [5]. This research shows that code improvements is the most frequent outcome of code reviews. Further more it also concluded that knowledge transfer among developers while code reviewing is definitely one of the outcomes. The code reviews done during the development of ARTS can only support the conclusions of Bacchelli, as the knowledge transfer between team members was definitely aided by the code reviews. Moreover, the flaws were found easily because of the insight of the reviewer.

Pair programming played a huge role for preventing these flaws as well. As Alistair Cockburn defines, "Pair or collaborative programming is where two programmers develop software side by side at one compute" [8]. This has been put into practice during this project. Developers teamed up almost every day to do some pair programming. This has been very useful as experience and specialisms of developers differs. Many decisions have been made whilst pair programming. During these sessions, most of the time is spent on argumenting a proposition. If both developers agree, the decision is made. If they disagree, a third developer or a member of the POS testing team will be consulted to help in making decisions. This ensures that good decisions have been made during the development of ARTS. Both code review and pair programming contribute to a higher code quality.

## Static analysis

As code changes can differ in size and complexity, code reviews can sometimes be error-prone. Some tools can be very helpful for static code analyses. To adhere to the Java coding standard, CheckStyle was used to guarantee proper formatting and contributed to readable code in ARTS. To discover design faults and bugs, both PMD and FindBugs were used. At the end of every week, it was made sure that these static analysis tools reported zero warnings. This improved the code readability as well as the code quality.

After all development work for ARTS was completed, we came across SonarQube, which is another static analysis tool that provides a more comprehensive analysis. After executing the analysis on ARTS' code base, the tool reported very positive results, which means that the measures we have taken to assure code quality were indeed effective. The test results can be seen in Figure 5.3.

## Prototyping and testing

ARTS is an application that is designed to replace an existing system, namely *TerminalTester*. *TerminalTester* was developed over the course of three years, and during this time period, *TerminalTester* amassed a lot of various features and requirements. The scope was so large that it would not be possible to understand every requirement before ARTS' development started. To still build ARTS in a quality manner, evolutionary prototyping is used. An evolutionary prototype

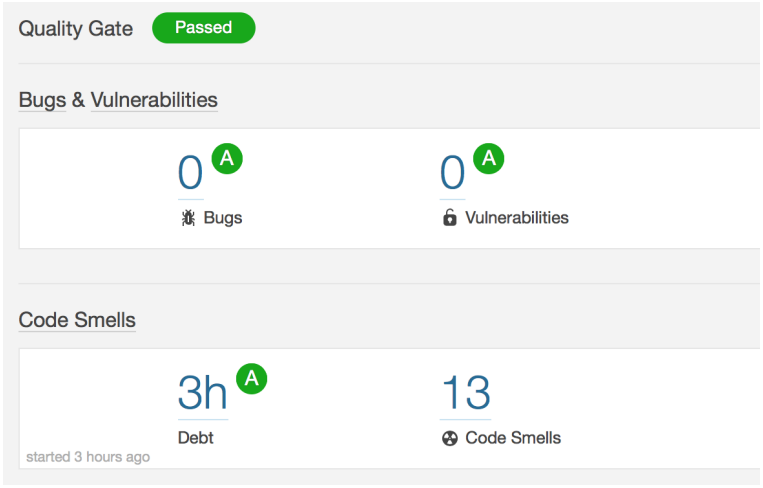


Figure 5.3: SonarQube's analysis results on ARTS' codebase.

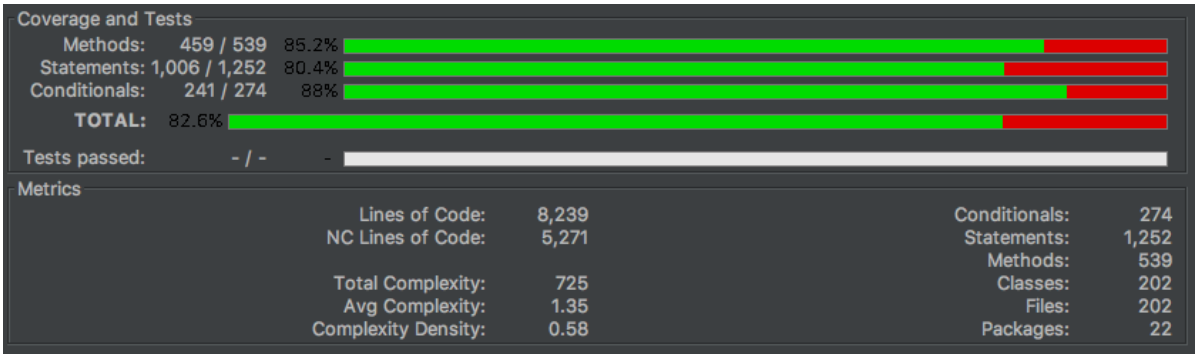


Figure 5.4: The final test coverage of ARTS, generated by the OpenClover tool [2].

implements the understood and confirmed requirements only [9]. After developing a prototype and requesting feedback from the users (the POS testing team), other requirements would be better understood. These requirements can then be incorporated into the prototype. It is key to understand that the prototype was developed rigorously with good coding practices, as this prototype would become the base for the following prototype. Using this approach meant that all the requirements and aspects of development could gradually be understood, which is useful as there are many of them.

In our opinion, testing forms a major part in rigorous software development. However, in the first half of this project, the software was not tested as thoroughly as we would like. Despite making use of evolutionary prototyping, we were still afraid that the interfaces and behaviour would change based on our possibly limited understanding of the requirements. The error of having little tests was rectified halfway through the project; an entire sprint of one week was dedicated to reducing technical debt and increasing test coverage. After this point, each new feature was comprehensively tested. The test result and coverage of the final version of ARTS is in Figure 5.4.

# 6

# Discussion, Future Work And Ethical Implication

This section gives and discusses multiple aspects of the project and the final product and also outlines the recommendations for future work.

## 6.1. Discussion

The multiple aspects of the project and the final product are presented in this section. For each aspect of the process, a brief reflection is given and also the lessons learnt. This is then to be used as a helping tool while carrying out future work.

### 6.1.1. Reflection on requirements

The requirements as defined in Chapter 3.2 were used to define the scope of the project. These served as guidelines to give us a clearer overview of the problem. While defining the requirements for the project, the aspect of achieving feature parity with the software currently used at Adyen for testing was not considered. The main focus was centered on the missing features of the present testing software. Consequently, a lot of time was spent on first of all achieving feature parity and fine-tuning the existing features already present in the testing software deployed at Adyen.

Midway in the project, communication needed to be established with the robot (made available to us by Adyen) for testing terminal application. A REST (Representational State Transfer) server had to be implemented by an Adyen employee to accomplish the aforementioned task, which took longer than expected. This time was not taken into account during formulation of the weekly sprint for that week. To compensate for this, a REST client interface was implemented to mock the behavior of the robot and avoid lags in the workflow.

Also, some features were delayed such as setting up and running the RRS (RESTful Robot Server) on Windows. The main task was to get this RRS set up and running on any machine running Windows. However, the RRS could no longer be run in a Windows environment because it involves enabling IIS( Internet Information Services) on Windows, which is not allowed by the security team at Adyen. This is due to the fact that the security team at Adyen is not fond of having a cross-platform REST connection. This issue was solved by in a later sprint by running RRS on mono in a UNIX environment. Uttermost priority was set on implementing the requirements in 3.2.1 while those in 3.2.2 were postponed. This decision to prioritize the requirements in 3.2.1 and postpone those in 3.2.2 to future work was taken by us after regular meetings with our Adyen supervisor.

#### Lessons learnt

Some requirements may not be clear while setting up the initial requirements. This is very evident especially when requirements arise from the user itself during the course of the project. For future work, it will be important to include dynamic requirements which will cover important features or the requirements from client. Also, it would be important to prioritize requirements. This is to handle the problem of limited time. Features up on the priority list should be implemented first, while those right at the bottom should be low priority features. When caught short of time, the low priority features should be omitted or postponed for further work.

### 6.1.2. Reflection on the entire development process

The entire project was split into two main phases. The first phase, which was carried out during the first two weeks of the entire project, was the research phase. After the research phase, the remainder of the weeks was the development phase. This phase consisted entirely of the development of the final product.

## Research Phase

As mentioned above, this phase was during the first two weeks of the project. The main goal of this phase was to gain an insight on the present workflow of the POS testing team at Adyen and gather information about available solutions to the specific problem described in chapter 3. Trying to gain an insight of the internal structure of the software used at the Adyen by the POS testing team and the present software architecture led us to several researches on the various libraries and APIs being utilized currently. Research was done on what test definition language was best to replace the previous XML being used, and also on the currently used API and libraries for the robots and terminals. Research was also done on what methods were best to create an interaction between the various components we were to work with (robots and terminals), and also the tools(such as Terminal WebSocket Server and RRS) to be utilized to carry out this task.

## Development Phase

This phase followed the research phase and lasted a total of 7 weeks starting from the third week of the development process. In this phase, the scrum methodology was used. Tasks were split from week 3 per team member for more efficiency, with each team member working on a different subsystem. Weekly sprints were made to assign different tasks to various team members and also to evaluate the milestones reached. The overall milestone is an indication of how much work from the previous sprint has been completed. It gave us a clear overview of what had been accomplished and what still had to be done. A private and internal Adyen GitHub repository was used during the development process wherein weekly issues were made and assigned to team members. Every issue also had a label (such as java, documentation, feedback, won't fix, python, regression, on hold) which described the issue e.g java refers to an issue which has to do with an implementation in java, a milestone (the specific sprint in which that issue has to be carried out) and an assignee (person responsible for the issue). A scrum board as shown on 5.2 was made available to us on which weekly task were written down and assigned. For each task, it was placed in a category on the scrum board. The categories on the scrum board were:

**Sprint Backlog:** This category contained all the tasks to be performed for that sprint, the issue or task number as it is on Github and the assignee

**In Progress:** This category contained all the tasks which were being worked on by a team member. Once a team member started working on an issue, it was then removed from the category Sprint Backlog to In progress. This includes tasks which have been started but are not completed yet.

**Pull Request Opened:** For every completed task, a pull request is opened. Task which are done are then moved from the In progress category to the Pull request open category. This is to give the other team members the possibility of reviewing the work done and approving it before it is merged.

**Absolutely Done:** This Category consists of all tasks for which the pull request has been merged. These tasks have been reviewed and are deemed as completed by the entire team.

The scrum board also gave us a clear overview of what was still to be done and how far we were for the week. The development phase started with the definition of a new test definition language (ATDL) to replace the presently used XML and that was in week 3. In week 4, the interaction between ARTS and the TWSS was established, and also a parser that parses XML to YAML was implemented and tested to make sure it is error-prone. In week 5, feedback was gotten from the TU Delft supervisor and Adyen based on what had been implemented so far. The feedback from the supervisor and Adyen was processed and implemented in week 6, alongside with implementations of the RESTful client for interaction with the robot. This implementation took longer than we expected. In the weeks after that, we worked on initiating connections with the robot and its interaction with the terminal. For the connection with the terminal, a Terminal Web Socket Server was created but it crashed frequently after a couple of tenders. Increasing the time delay between requests to the server was noticed to lower the probability of the server crashing but not good enough for the stable running of ARTS. Due to this, we ditched the websocket server and implemented POSJNI natively which removed the server layer and every server bug. In the course of time, we noticed that the websocket server was better and readopted its use instead of running POSJNI natively. About 30-32 hours were assigned for each team member per week for implementation, leaving about 8-10 hours free for weekly scrum meetings, meetings with our Adyen supervisor and various parties, minor fixes, unfinished implementations, code review, development challenges and poor time management. This time was used judiciously to implement feedback from Adyen and our TU Delft coordinator.

## Lessons learnt

One of the major lessons learnt is that the 7 weeks for which the development phase lasted proved to be too short for the implementation of all features. As explained in 6.1.1, the inclusion of feature parity as one of the requirements was not included in the requirements in Chapter 3. This, combined with time constraints for the implementation of major features led to some features not being implemented. For further projects, it will be important for us to either limit our list of functional requirements or discuss the deadline. For this project, the deadline was a strict one and could not be influenced by us. This means the best and only option is to limit the number of functional and non-functional requirements. This should be properly discussed with Adyen.

## 6.2. Future Work

During the 2 months of the project at Adyen, a lot was learned. Due to the short duration of the project, not all features we planned to implement could be implemented. This section presents the most important features which should be considered for future work.

### Running of tests with Jenkins

Jenkins is a real time, open-source continuous integration software tool which is used for testing and also for reporting on isolated changes in larger code bases. This software tool helps automate test builds and also helps developers to discover and solve defects in a code base rapidly. Jenkins is particularly of importance because of the continuous integration aspect where in developers constantly get feedback on the software. It also detects deficiencies early in the development stage, defects are usually small and easy to resolve because they mostly are not complex. For the running of tests on the robots, it will, therefore, be advantageous to use Jenkins because of the aforementioned features. In this way, bugs can quickly and easily be detected and fixed[14].

### Running of tests on the robot and terminal from a centralized workspace

One of the main goals out of the scope of this project was to be able to run tests on all robots and terminals at Adyen from a centralized workspace. A prerequisite to obtaining is first of all being able to run tests on individual robots from a personal computer running Windows. A centralized workspace for the running of tests on all the robots and terminals available would ease the workflow of the POS testing team at Adyen and it would also allow for the scheduling of specific robots for the running of particular tests. This is a feature which Adyen's testing application lacks. With the present testing application, every robot needs an instance of the testing application on a dedicated machine, which does not scale. This also does not allow the user to select specific test cases to be run on the terminal by the robot arms.

### Provision of user friendly dashboard for viewing of test results

The actions performed by the robots on the terminal are used to verify the behavior of a particular terminal. The results obtained from the actions performed by the robots are then verified against the expected results to identify mismatches and incorrect terminal behavior. For future work, a user friendly dashboard should be implemented so that users can easily view the results of a test run. This would help easily identify failed tests or incorrect terminal behavior, which can then be quickly fixed.

## 6.3. Ethical Implications

There are many ethical issues surrounding the development and management of software, and these are made more complex by the power of individuals and infrastructures. First, we will define ethics. "Ethics is defined as the study of morality and the application of reason which sheds light on rules and principle, which is called ethical theories and ascertains the right and wrong for a situation"[4]. Ethics is of course applicable in various professional fields but in this case, we will focus more on the "rights and wrong" of the software we developed during the period of this internship.

In the course of the project, to be able to understand how the present testing system works, reverse engineering was carried out. Reverse engineering has to do with revealing the source code of an application by decompiling it, and does not involve changing the present system. It is a process of examination and not change or replication[7]. To fully develop ARTS, the software for robot communication was decompiled in order to gain insight into how it worked. This process is controversial, as it can both be considered both an ethical issue and at the same time not an ethical issue in the process of software development. It is not considered an ethical issue when it is carried out for maintenance purposes as it helps you understand the system in order to be able to make changes to it[7], which is the case for ARTS. Decompiling software with a license which prohibits reverse engineering is ethically wrong, and can lead to legal ramifications for example fines and even civil suits but doing this for maintenance is ethically right as it helps developers understand the system better and also helps developers identify what components of the present system are reusable (in the case of software reusability)[7].

ARTS is developed to broaden the capabilities of the present testing application being used at Adyen. It adds to the features which the present testing application possesses, thereby, increasing its functionality, meaning less manual work to be performed by the POS testers at Adyen. This is because ARTS eliminates some functionalities which are manually carried out by the testers at Adyen. Looking at this from an ethics point of view, it is considered to be ethically right and justified by the fact that the workload on the POS testing team at Adyen is reduced and the functionality of the testing application presently being used is improved.

ARTS is developed to be quality-assured. Several static analysis tools and testing are used to produce a high quality assured application. The ethical implication in this case is that Adyen does not lose revenue in the form of lost productivity due to bugs and security flaws. Software developers and companies can lose customers and revenues due to a ruined reputation which can come as a result of software bugs.

# 7

# Conclusions

The advent of technology has allowed for the possibility of both online (e-Commerce) and in-store (point of sale) payments for goods. For this to be possible, the services of a Payment Service Provider (PSP) is needed, who offers merchants online services for accepting electronic payments from their customers. It ensures a safe, reliable, and faster transfer of funds from the shopper' s bank account to the merchant's bank account. Adyen, an omni-channel PSP is different from the others because it offers both online and in-store payment services to merchants. This project, done at Adyen focuses on point of sale, which encompasses the payments that are done physically by the shopper in a physical shop, such as buying a Foosball table at Walmart. The terminal firmware used by these merchants to accept in-store payments from their customers is provided by Adyen. To ensure a reliable and error-prone software, Adyen has to test its terminal firmware to discover bugs and fix them before getting these out to the merchants. To perform these tests, Adyen utilizes robots which are controlled in order to execute a test suite. For this purpose, Adyen possesses a state of the art application called *TerminalTester* which starts a series of transactions, instruct the robots at Adyen to carry out these transactions and afterwards, verifies the correct behavior of the payment terminal. In order to resolve the limitations of *TerminalTester*, some new features have to be implemented. At the moment, the *TerminalTester* application has a lot of technical debt, which makes the application difficult to extend. Addition of new test types such as network tests (connecting and disconnecting from the internet) or power tests to the present *TerminalTester* is difficult to implement due to this technical debt. Also, *TerminalTester* is currently written in .NET, making the application platform-dependent. As a result of this, all robot computers are windows machines. Finally, the *TerminalTester*'s test definition language is quite convoluted, making the written test definitions difficult to read and understand.

The current testing arrangement used at Adyen has some areas which require improvement. The result of this project is the Adyen Robot Test Server ARTS, which provides a solution to these areas of improvement. The areas which could be improved and lie in the scope of this project are stated below and for each area, a solution is given.

## 7.1. Technical debt of TerminalTester and platform dependency

The current TerminalTester has a lot of technical debt and is built in .NET. To reduce this debt, the POS testing team already attempted to refactor the TerminalTester to an expandable state. This refactoring allowed for the alignment of testing functionality across libraries and decouple testing logic from library logic but this was not sufficient. Also, because this is written in .NET, it is platform dependent. This is not ideal for Adyen as a platform independent application is preferable.

### Solution

The POS testing team requested TerminalTester to be rewritten, with all the modern needs in mind. This rewritten application is ARTS. ARTS is platform-independent, therefore, solving the problem of a platform dependent application. For this reason, Java is used to write ARTS, as this is a platform-independent language with which the team members have a significant amount of experience.

## 7.2. Test definition language

The language used to define tests at the moment is quite convoluted, which makes it hard to write new tests and to understand the written tests. This is due to the fact that these tests are stored in a large XML file of about 5000 lines, thus making it difficult to understand and interpret. This is particular to XML, as the main aim of XML is to structure documents and information, but it gives very little or no interpretation of the information or data contained in the document. The factors which make the XML less favorable as a test definition language are:

**Readability** This factor has been the main reason to implement a new representation for test suites. The size and complexity of the current XML file makes it difficult to read the test suite.

**Scalability** There are still numerous test definitions that can be defined, due to the variety of terminal models (and software run on it) present at Adyen. This is difficult to achieve at the moment due to the information redundancy problem explained earlier.

**Extensibility** Adyen would, in the near future, like to add new types of test definitions e.g. network, power and user interface tests. Doing this in the current XML scheme is a tedious and cumbersome task.

## Solution

A new test definition language, called Adyen Test Definition Language is implemented as solution. This language is defined using the YAML notation which makes the document less complex. It also ensures that the test suite is extensible, by allowing the user to define new types of test cases in a intuitive way.

While the product could be used as it is, we recommend further development in order to incorporate the remaining requirements.

# A

# Acronyms

**PSP** Payment Service Provider

**CLESS_CHIP** Contactless Chip Reader

**ICC** Integrated Circuit Card

**MSR** Magnetic Swipe Reader

**POS** Point of Sale

**PED** PIN Entry Device

**PIN** Personal Identification Number

**XML** Extensible Markup Language

**YAML** YAML Ain't Markup Language

**JSON** JavaScript Object Notation

**ATDL** Adyen Test Definition Language

**ARTS** Adyen Robotic Test Server

# B

# XML test case example

```
<TestCaseList name="NonDccTests" repeat="1">
   <TestInput>
      <Parameters>
         <Parameter name="Currency">EUR</Parameter>
         <Parameter name="StartValue">0</Parameter>
      </Parameters>
      <MultipleValueParameters>
         <MultipleValueParameter name="TenderOptions">
            <Value>ReceiptHandler</Value>
            <Value>GetAdditionalData</Value>
         </MultipleValueParameter>
      </MultipleValueParameters>
      <ParameterSets>
         <ParameterSet name="AmountBracketIndex">
            <Value>Signature</Value>
            <Value>OfflinePin</Value>
         </ParameterSet>
         <ParameterSet name="AppToSelect">
            <Value>1</Value>
            <Value>2</Value>
         </ParameterSet>
      </ParameterSets>
      <DynamicParameters>
        <DynamicParameter>
           <ParametersToMatch>
              <Parameter name="AmountIndex">Signature</Parameter>
           </ParametersToMatch>
           <ParametersToSet>
              <Parameter name="StartValue">251</Parameter>
           </ParametersToSet>
        </DynamicParameter>
        <DynamicParameter>
           <ParametersToMatch>
              <Parameter name="Amountndex">OfflinePin</Parameter>
           </ParametersToMatch>
           <ParametersToSet>
              <Parameter name="StartValue">10001</Parameter>
           </ParametersToSet>
        </DynamicParameter>
      </DynamicParameters>
   </TestInput>
   <TestOutput>
      <ValidCombination>
```

```
        <ParametersToMatch>
           <Parameter name="AmountBracketIndex">
              Signature
           </Parameter>
        </ParametersToMatch>
        <ValidResults>
           <Result>
              <ResultValue name="TransactionResult">
                 Approved
              </ResultValue>
           </Result>
        </ValidResults>
     </ValidCombination>
     <ValidCombination>
        <ParametersToMatch>
           <Parameter name="AmountBracketIndex">
              OfflinePin
           </Parameter>
        </ParametersToMatch>
        <ValidResults>
           <Result>
              <ResultValue name="TransactionResult">
                 Declined
              </ResultValue>
           </Result>
        </ValidResults>
     <ValidCombination>
  </TestOutput>
</TestCaseList>
```

# Original Project Description

Adyen is a so-called omni-channel payment service provider that offers solutions for merchants to accept payments online, on mobile devices, and in-store. All those channels connect to the same online platform that then communicates with credit card schemes and banks. This project focuses on the in-store solution, which is commonly referred to as point of sale (POS). The POS solution consists of a payment terminal running custom Adyen application software. For easy integration with cash registers, Adyen offers different libraries that interact with the terminal (e.g., for starting a transaction). For testing the full solution in an automated fashion, Adyen currently uses five testing robots that physically interact with the terminals (i.e., that press keys, insert a payment card, et cetera). However, Adyen supports many different device models, that all behave slightly different – for that reason, Adyen ordered ten more testing robots, so that they can test new versions simultaneously on all devices.

The current testing solution consists of a tailor-made .NET application that integrates with the different libraries Adyen offers, and the library to control the robot. Each robot requires its own instance of this test application, running on a dedicated machine. Clearly, this does not scale easily. Furthermore, all test cases are currently defined in long XML files that are difficult to maintain. The goal of the internship project is therefore to design and implement a (Java) solution that does scale, by the means of one centralised management system that controls all robots and test runs. This solution should also allow for writing test cases/scenarios in an intuitive fashion. Ultimately, the testing system needs to be integrated in Jenkins. Stretch goals include designing new test scenarios, such as controlled network tests (e.g., reliably disconnect the internet in different stages of a transaction), power outage tests (e.g., reliably introduce power loss in different stages of a transaction), and UI tests (i.e., verify whether all screens are shown correctly).

# Bibliography


[1] Introducing JSON. `http://www.json.org`.

[2] Openclover web site. `http://openclover.org`.

[3] JSON Pros and Cons. `https://myarch.com/json-pros-and-cons/`.

[4] Haslinda Abdullah and Benedict Valentine. Fundamental and ethics theories of corporate governance. *Middle Eastern Finance and Economics*, 4(4):88–96, 2009.

[5] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 2013 international conference on software engineering*, pages 712–721. IEEE Press, 2013.

[6] Oren Ben-Kiki, Clark Evans, and Brian Ingerson. YAML Ain't Markup Language (YAML™) Version 1.1. *yaml. org, Tech. Rep*, 2005.

[7] Elliot J. Chikofsky and James H Cross. Reverse engineering and design recovery: A taxonomy. *IEEE software*, 7(1): 13–17, 1990.

[8] Alistair Cockburn and Laurie Williams. The costs and benefits of pair programming. *Extreme programming examined*, pages 223–247, 2000.

[9] Alan M Davis. Operational prototyping: A new development approach. *IEEE software*, 9(5):70–78, 1992.

[10] Stefan Decker, Sergey Melnik, Frank Van Harmelen, Dieter Fensel, Michel Klein, Jeen Broekstra, Michael Erdmann, and Ian Horrocks. The semantic web: The roles of XML and RDF. *IEEE Internet computing*, 4(5):63–73, 2000.

[11] SDK Dynamixel. for pc manual, version 1.0. seoul: Robotis, 2008.

[12] Malin Eriksson and Victor Hallberg. Comparison between JSON and YAML for data serialization. *The School of Computer Science and Engineering Royal Institute of Technology*, 2011.

[13] Martin Fowler and Kent Beck. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[14] Martin Fowler and Matthew Foemmel. Continuous integration. *Thought-Works) http://www. thoughtworks. com/-Continuous Integration. pdf*, page 122, 2006.

[15] Zia Ul Haq, Gul Faraz Khan, and Tazar Hussain. A Comprehensive analysis of XML and JSON web technologies. *New Developments in Circuits, Systems, Signal Processing, Communications and Computers*, pages 102–109, 2012.

[16] Kenneth C Laudon, Carol Guercio Traver, and Alfonso Vidal Romero Elizondo. *E-commerce*, volume 29. Pearson-/Addison Wesley, 2007.

[17] Ramon Lawrence. The space efficiency of XML. *Information and Software Technology*, 46(11):753–759, 2004.

[18] Manny M Lehman and Laszlo A Belady. *Program evolution: processes of software change*. Academic Press Professional, Inc., 1985.

[19] Robert C Martin. Principles of OOD. *URl: http://butunc lebob. com/ArticleS. UncleBob. PrinciplesOfOod (Last accessed: 22nd June 2017)*, 1995.