Joachim Lesser, BSc

NFC extension for Catrobat

Master's Thesis

Graz University of Technology

Institute for Softwaretechnology
Head: Univ.-Prof. Dipl-Ing. Dr.techn. Wolfgang Slany

Supervisor: Dipl.-Ing. Dr.techn. Christian Schindler

Graz, May 2018

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Graz, _____ _____

Date Signature

Eidesstattliche Erklärung[1]

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am _____ _____

Datum Unterschrift

[1] Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008; Genehmigung des Senates am 1.12.2008

Abstract

The need for software developers and people with programming skills is increasing continuously. It is often demanding and difficult, especially for children, to start learning programming. The Catrobat project was launched with the aim to bring children in contact with software development and show them that programming is enjoyable. Catrobat is a visual programming language particularly developed for smartphones. It shows young people the capabilities of everyday technology. Further various built-in hardware sensors and features in smartphones are supported.

The goal of this thesis is to illustrate the capabilities of Near Field Communication (NFC) to young people, being implemented as a Catrobat extension, which should enable Catrobat users to learn something about NFC and its usage. With the help of the implemented NFC features, user should be able to read the content and the ID of NFC tags as well as to write new content to tags. The content and the ID of NFC tags can also be used to handle conditions in the program logic.

To guarantee a good user experience, an application has to be stable and flawless. Catrobat meets this requirement through obligating the developers of the application to use Test-Driven Development (TDD). Since emulated Android devices are not able to work with NFC hardware simulation, an external testing tool is necessary to test the implemented NFC features. This testing tool is an Arduino based server with several sensors, a NFC Shield for NFC tag emulation and an Ethernet shield. It is connected to the Jenkins network and therefore suitable for every Catrobat developer.

Contents

List of Figures

Glossary

ADB	Android Debug Bridge
ADC	Analog to Digital Converter
API	Application Programming Interface
AUT	Application under test
AVD	Android Virtual Device
CI	Continuous Integration
CNC	Computerized Numerical Control
IDE	Integrated Development Environment
IEC	International Electrotechnical Commission
IMU	Internal Measurement Unit
IoT	Internet of Things
ISO	International Organisation for Standardization
kB	Kilobyte
NDEF	NFC Data Exchange Format
NFC	Near Field Communication
P2P	peer-to-peer
PCD	Proximity Coupling Device
PICC	Proximity Integrated Circuit Card
PWM	Pulse-width modulation
REST	Representational State Transfer
RFID	Radio Frequency Identification
Rx	Receiver
SD	Solid Disk
SS	Slave Select
SUT	System Under Test
TNF	Type Name Format
Tx	Transmitter
URI	Universal Resource Identifier
UUID	Universally Unique Identifier
UX	User Experience

1 Introduction

The number of smartphone users worldwide is increasing year per year[1]. This is comprehensible since smartphones bring a lot of advantages in the modern world. No other smart device is as versatile and portable and as well integrated into the daily life as the smartphone. Even laptops and tablets are not that mobile to be used during a short bus ride or in the tramway.

Especially for young people good smartphones are becoming more and more important and indispensable. At this point it is necessary to show young people that smartphones can not only be used for gaming or sharing pictures but also for acquiring new skills and extending one's knowledge.

This is the reason why the Catrobat project was founded by Professor Slany at the Graz University of Technology. Catrobat is a visual programming language for teenagers who are new to the field of software development. They do not need a laptop or desktop computer for programming, only a functioning smartphone. Nonetheless Catrobat has that many features implemented and supports a wide range of programmable hardware, that even experienced junior developers can use it. Some of these supported features are face detection, the gyroscope of the phone which measures the tilt, or code blocks for the Lego robot NXT.

The major difference between Catrobat and common programming languages is the code design. In Catrobat it is not necessary to memorize the programming language specific commands, since all commands are available as drag and drop able bricks. This means that programs are not coded the usual way, but instead they are a combination of various bricks.

[1] https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/

In the course of this thesis a new brick is added which extends the hardware features by Near Field Communication. Further an automated testing system for hardware features is integrated.

1.1 Motivation

Related to the increasing number of smartphone users the amount of Android phones shipments is rising [2]. This is relevant since the integrated development environment (IDE) for Catrobat, by the name of Pocket Code, is available for Android. Catrobat is in the same way as Android an open source project. This means that the source code is free available and can be viewed and copied by everyone. With a market share of about 85% Android is the most widespread operation system for smartphones worldwide.

The philosophy of Catrobat is to delight children and youth for programming. They should also be encouraged to express themselves in a creative way with the new media. This promotes the skills of connecting logical processes, creating creative solutions and solving problems with a programming-approach.
Learning programming by playing a game includes a fun factor that could keep the motivation higher than simply studying conventional programming languages.

Through this thesis it is possible to bring these young programmers in contact with the widely used Near Field Communication (NFC) technology. NFC is already integrated into the daily life of many people. It is used for payments in the supermarkets, for entry controls, to transmit small amounts of data like contact information or website links and it can also be used for fast pairing of Bluetooth devices.
The possibility to write and read NFC tags in Pocket Code introduces this technique and its capabilities to young people. Writing to tags is made possible through a preselected range of common used NFC message types

[2] https://www.statista.com/statistics/272307/market-share-forecast-for-smartphone-operating-systems/

like `HTTPS`, `MailTo` or `text`. This enables Pocket Code users to exchange information with the help of NFC tags.

The second part of this thesis aims to help the developers of Pocket Code. It is quite time consuming to test features of a mobile application manually with every pull request, software update or release. Unfortunately not all hardware features of Pocket Code can be tested on an emulated device. It is necessary to install the application on a physical device to test the hardware input and output.
Some of this manual testing can be automatized by using external hardware sensors. In combination with an easy to use interface this is suitable for all developers that work on Pocket Code.

1.2 Problem Statement

This thesis deals with two fundamental questions.
The first issue is how to enable children and young people to explore the Near Field Communication (NFC) technology with its advantages and potential but also its restrictions.
For this question the relevant and usable features of NFC are elaborated relating to Pocket Code. Several extensions for Pocket Code are necessary.

The second question is how to enable automated testing for Pocket Code hardware related features.
All Catrobat developers are committed to write tests before the productive code to fulfil the Test-Driven Development (TDD) guidelines. Since not all available features in Pocket Code can be tested meaningfully on an emulated device a solution with external hardware is needed to enable TDD.

These two issues are connected with each other, since TDD is a way to improve the quality of a software. It is necessary to offer a stable and less error prone application to enhance the user experience. Children and young adults are quickly disinterested in buggy software which stands in contrast to the Catrobat philosophy of delighting them for programming. Because of this a special testing tool for automated hardware testing is indispensable.

1.3 Thesis Outline

This thesis is structured as follows.
In Chapter 2 all essential background information is provided, starting with a short look into Catrobat and following with the Test-Driven Development (TDD). Since TDD is crucial in Catrobat it is explained in more detail. Therefore the three relevant testing frameworks for Catrobat are explained. Further Jenkins is presented because it is the used continuous integration tool. The next section shows how an Arduino works, because this microcontroller will be the basis of the external testing server. As main topic of this thesis Near Field Communication (NFC) is explained in detail. Starting with the NFC device types and operating modes, followed by the structure of the NFC Data Exchange Format (NDEF) and the NFC ISO norm 14443. Finally the radio controller PN532 as special part of the hardware testing box is presented.

Chapter 3 deals with Catrobat in detail. It starts with a clarification why TDD is that important for the project. Subsequently the different types of tests and testing frameworks are discussed with a focus on a comparison of Robotium and Espresso. Furthermore the pros and cons of the emulated device and the real device are listed. To understand in which development states the hardware testing box tests have to executed, the ticket workflow of Catrobat is shown. As last section in this chapter the usage of Arduino in Pocket Code is presented.

In Chapter 4 all implemented NFC features are expounded. Starting with the NFC sensor variables which are newly available device variables that contain information about the last read NFC tag. Followed by the `SetNfcTag` brick that allows to write to tags during program runtime. Subsequently the implementation of a NFC automated Bluetooth pairing is presented. Next the Arduino hardware testing box is explained in detail, starting with the construction of the container and followed by a listing of each component. At first the hardware and software implementation of the audio shield is explained, followed by the evaluated Ethernet shield and NFC shield. As last components the light and vibration sensor hardware and software implementations are explained. The final section deals with the

debugging of the hardware testing box which provides an instruction for future developers.

Finally in Chapter 5 a conclusion of this thesis with the implemented features and the new findings are presented. At last some ideas for possible future work are propagated.

2 Background

This chapter provides information about techniques and hardware used in this master thesis. It describes a.) the free open source software (FOSS) project Catrobat (C. Contributors, 2017), b.) the Extreme Programming (XP) (Beck, 2000) practice Test-Driven Development (TDD) (Langr, 2004) and the use of three different testing frameworks, c.) the role of Continuous Integration (CI) using Jenkins (J. Contributors, 2017a), d.) use of Arduino (Arduino Contributors, 2017b) and finally e.) Near Field Communication (NFC).

2.1 Catrobat

"Catrobat is a visual programming language and set of creativity tools for smartphones, tablets, and mobile browsers"[1].
It is specifically designed for teenagers aged between thirteen and eighteen. The integrated development environment (IDE) Pocket Code enables the user to directly create programs on Android and iOS devices. Catrobat programs can be executed in the IDE on the mobile device or on any HTML5 capable (mobile) browsers. Such programs can be animations, music videos games, or other apps using the internal device sensors such as GPS location, surrounding volume, inclination and so forth. With Pocket Code programs for external hardware like Arduino, Raspberry Pi, Lego NXT or remote control flying drones, and wheel driven robots can be created. Catrobat is a block based visual programming language where each block represents a separate functionality. Blocks are ordered in lists which are part of graphical objects and can be arranged to form a certain program.

[1] https://www.catrobat.org/

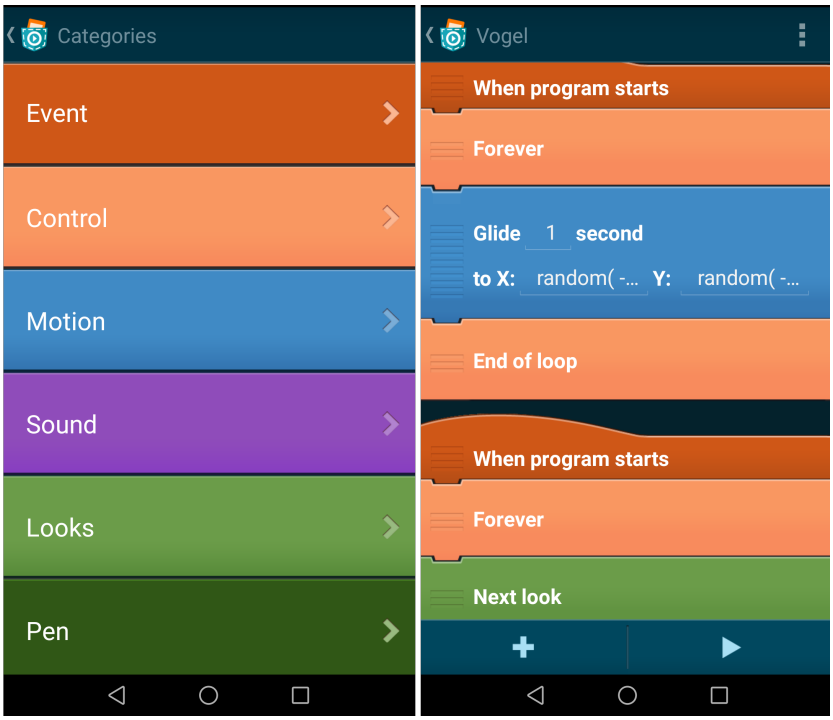

Figure 2.1: Screenshots of Pocket Code in the 'add brick' menu and the object 'Vogel'

A main focus for the development of Catrobat code is on using Extreme Programming practices according to Beck (2000), which is a combination of Test-Driven Development (Langr, 2004) with Collective Code Ownership along with other agile methods, like Continuous Integration and Ping-Pong-Pair-Programming. The code styling is regulated via Clean Code methods, which are described by Martin (2009), and several other standards like Simple Design to ensure a high quality code as well as an always up to date documentation.
To efficiently coordinate the work of the project members all tasks regarding development of new features, bug fixes and all other adaptations are stored as ticket. As ticketing system Jira[2] is used in combination with a Kanban board[3] to visualize the work in progress, what has to be done and what is already done.

[2]https://www.atlassian.com/software/jira
[3]https://www.atlassian.com/agile/kanban

2.2 Test-Driven Development

This master thesis is about hardware features and hardware testing. Therefore Test-Driven Development as specified by Langr (2004) plays an important role.

TDD can be seen as a circular workflow. The developer starts with writing a test for a feature which has to be implemented. Since the feature is not implemented yet the test must fail. The next step is to add functionality to the previously written test so the test runs successfully. Therefore only the actual needed parts should be implemented. Once the test passes it is time to refactor the written code of the feature and the test. When this is done, the next test has to be implemented regardless of whether the existing feature is extended or a new one is created. By following these simple three rules a circular workflow accrues which can provide a lot of benefits for the developer and the whole project such as tested code and a simple and testable architecture.
By continuously practising the radical refactoring the quality of the code can be improved significantly (Nagappan et al., 2008). Code duplication and other code smells like hard coded values are prevented since refactoring is done continuously in small chunks. Another good aspect is the focus on the core feature which is under development. It gets harder to drift off and fall into the trap of feature creep when applying TDD and complying to Clean Code.
Clean Code according to Martin (2009) includes to use no abbreviations, no comments and carefully considered and appropriate names for variables, methods and so on. Further no duplicated code and a well-wrought design for the system is an essential part. As a result of this the code has to be self-explained and clear especially for foreign developers.

In probably every big software project with various Developers, several challenges and problems occur which must be handled to be longtime successful. TDD can be a key to provide a solution for these challenges. Catrobat has a high turnover rate because it is solely developed by volunteers. Most of them are students and hence work on the project for their bachelor or master thesis. So it is common, that members only work for a limited amount of time on the project before leaving again. Another challenge is the knowledge

transfer before they leave since they often are hard to contact after finishing their work on the project.
Because of this circumstances TDD is so important to run the project successfully. The high staff turnover rate is easier manageable if no or at least less knowledge gets lost about the written code. If features need to be changed or get superfluous, it is easier to find the affected classes via changing the test to the new requirements and follow the normal TDD circle, namely fixing the test via fixing the functional code.

There are three challenges using Test-Driven Development. As already mentioned a simple design is necessary to be able to test single components. When the software is test-driven developed from scratch, the developers are forced to use a good design.
Second, tests must be automated. They must be executed periodically on a test system and they have to run on the client of the developer that they can profit therefrom. To achieve test runs on a regular basis, they should run on a continuous integration system which can build the software and execute the test on a defined schedule. Such a continuous integration system is used in the Catrobat project. The Jenkins server is not only configured to build and execute the tests on regular basis but is configured to start all tests when a pull request is issued by a developer. A pull request is a request for merging a feature where its developer thinks it should be integrated into the code base.
Finally the developers need a lot of practice to write good and useful tests in adequate time. This can be the biggest obstacle when starting with TDD, because it takes time to master writing tests especially before the functional implementation. In Android to test the functionality of a feature the recommended framework is JUnit[4]. If tests are related to the user interface, frameworks like Robotium[5] or Espresso[6] are required.

[4] http://junit.org/junit4/index.html
[5] https://github.com/RobotiumTech/robotiumv
[6] https://developer.android.com/training/testing/espresso/index.html

2.2.1 JUnit

Writing tests is alleviated by testing frameworks and by default the xUnit framework is a framework which shares features regardless of Programming language. For Android Development Java is the language of choice, therefore tests are written using the JUnit framework. JUnit[7] is as a simple testing framework developed by Erich Gamma and Kent Beck (2000) as a tool for Extreme Programming.

According to Krümmel and Ried (2011) it is common practice to put the test code in own test classes which are separated from the functional code but reside in the same package. It is a common convention to choose the name as a composite of the tested class and the 'Test'-postfix. For each method to test at least one own testing method must be available. A testing method is concerned to test a single functionality of a class. Therefore a class under test must be tested by multiple testing methods to verify its compliance to the specification and validate its functionality. To speed up tests and during development of tests, preconditions for all tests in a test class can be defined.

To keep tests clear and help the test reader to quickly ascertain the verified behaviour, the structure of the test logic should follow a four phase sequence. As described by Meszaros (2007) this sequence is divided into the setup, exercise, verify and teardown phase.
The setup is the precondition for a system under test (SUT). It is called prior the requested logic is executed. In this context the fixture setup sets up the so called test fixture, which consists of the test objects and their state.
The exercise phase can only be executed after the fixture setup phase is completed. Here the SUT logic, which has to be tested, is simulated by the test. This means that here the actual software parts are executed to test the behaviour.
The verification alias result verification phase follows after the exercise SUT phase. The behaviour of the SUT with the verification of the actual outcome compared to the expected outcome is displayed as a Boolean result, pass or fail.
The last step is the teardown phase. The fixture teardown destroys the

[7] http://junit.org/junit4/index.html

test fixture which was created by the test prior. This step is regarding organizational reasons, because teardown has nothing to do with a test as documentation and the clear test logic should not be disguised.

To combine tests, which for instance belong to the same category, it is possible to arrange them in suits. They offer the possibility to group single test classes to a bundle, which allows to run the containing tests successively. JUnit works with annotations for methods to distinguish test methods from boilerplate code or helper methods. The most commonly used annotations are listed below with along with a small explanation to get a quick overview of JUnit possibilities. As explained by Krümmel and Ried (2011) following annotations have to be written immediate above the test method's signature.

- `@Test`

Marks the method as test case which in turn is executed by the testrunner.

- `@Test(expected=...Exception.class)`

Marks the method as test case with the condition that the test only succeeds, if the defined exception is thrown.

- `@Test(timeout=...ms)`

Marks the method as test case with the condition that the test only succeeds, if the runtime is smaller than the defined value.

- `@Before`

Runs the annotated method before each test method is executed. This enables the developer to create a common setup method for all test cases.

- `@After`

Runs after each called test method to do clean-up/teardown operations.

Annotations for classes are used to define the test runner or bundles of tests (Krümmel and Ried, 2011).

- `@RunWith(... .class)`

If a different test runner should be used, the class executing the tests can be defined here. For instance this is used to run parameterized[8] tests.

- `@Suite.SuiteClasses(... .class, ...)`

A definition of classes that belong to this suite.

[8] https://github.com/junit-team/junit4/wiki/parameterized-tests

The following assert methods are used inside the test methods to verify the outcome of method calls on the classes under test (Krümmel and Ried, 2011).

Assert Method	Behaviour
`assertEquals(Object exp, Object act)`	Checks, based on the equals-method of the object, if the objects are equivalent. Can also be used for floats, doubles or arrays.
`assertEquals(float exp, float act, float delta)`	Checks equivalence of doubles or floats within the difference of delta.
`assertFalse(boolean condition)`	Verifies if condition is false.
`assertNotNull(Object object)`	Checks if the object is unequal null.
`assertNotSame(Object unexp, Object act)`	Verifies if two references point to different objects.
`assertNull(Object object)`	Checks if the object is null.
`assertSame(Object exp, Object act)`	Verifies if two references point to the same object.
`assertTrue(boolean condition)`	Verifies if condition is true.

2.2.2 Robotium

Robotium is an open source test framework based on the Android test framework (Vogel, 2016b). Robotium tests are derived from the `ActivityInstrumentationTestCase2` class. In contrast to JUnit, Robotium is a specific Android framework created by R. Contributors (2017) for user interface (UI) testing with integration in Android Studio and Eclipse. Robotium can be run with automated build systems, e.g., Gradle[9], Maven[10] or Ant[11] on physical Android devices and Emulators which makes these UI tests eligible to be run in a continuous integration (CI) build chain. Robotium provides full

[9]https://gradle.org/
[10]https://maven.apache.org/
[11]http://ant.apache.org/

support for hybrid and native applications. It primarily uses the solo class for testing. Only minimal knowledge of the tested application is required, so it can be used for testing applications with available source code and even applications where the implementation details are not known and only an APK file available becomes testable. It supports various Android features like toasts, activities, menus and context menus also multiple Android activities are handled automatically by the framework. Even for beginners in test development it is easy to write meaningful and solid tests, because of the simple click on item principle. The readability of Robotium test cases is much better compared to the standard instrumentation tests.

2.2.3 Espresso

Another user interface (UI) testing framework used in the Catrobat project's Android development is the Espresso framework. Espresso is an open source instrumentation based API which uses the JUnit test runner, more exactly the AndroidJUnitRunner(Android Contributors, 2016). With release 2.0 it became part of the Android Support Repository (Vogel, 2016a) and can run according to the Android Contributors (2016) on devices running Android 2.3.3 and higher.
Within a single target app Espresso provides APIs for UI tests to simulate user interaction. The framework can also be used for black-box testing, but unleashes its full power only when the codebase under test is known (Community, 2017). Test actions are automatically synchronized with the UI of the application. The framework determines when the main thread is idle and improves the reliability of the test by waiting to run the test commands at the proper time when the observed background activities have finished. This frees the developer from having to add any waits in the test code. Stated by Vogel (2016a) Espresso basically consists of the following three components:

- ViewMatchers
  The ViewMatchers find view objects in the view hierarchy.
- ViewActions
  With the ViewActions actions like clicking can be performed on the views.

- ViewAssertions
  The last checks asserts about view existence in the view hierarchy

2.2.4 Jenkins

Jenkins is a cross-platform, continuous integration and continuous delivery application (Kohsuke, 2016). The goal is to increase productivity of development by building the whole application continuously, running all configured tests and providing feedback of the build and test results.
This automation server software is written in Java (W. Contributors, 2017) and runs in any Enterprise JavaBeans container. It is released under the MIT License[12] and is therefore open-source. Jenkins is delivered with Winstone[13], a minimal servlet container, but can also be used with completely fully functional J2EE style servlet containers such as Tomcat or Jetty. By default various build-tools are supported like Apache Ant, Maven or Gradle and commonly used version control systems like CVS, GIT or Subversion. It comes along with support for automated testing tools like JUnit[14] or Emma[15].
There exist over 1300 plugins (J. Contributors, 2017b) to extend the functionality of Jenkins. It can control other compilers beside Java to also administrate, e.g., .NET, PHP, Python or Ruby based Projects. It has a REST-based API, so it can be controlled from other programs.
The main reason to use Jenkins is that continuous builds (J. Contributors, 2017a) and tests of software alleviate integration of and changes to the project. By defining your build pipelines and integrating various testing and deployment technologies, it provides a powerful way to continuously deliver software. At the time of writing this Master Thesis, Jenkins is according to a statistic of Maple and Shelajev (2016) the most widely used tool for continuous integration.

[12] https://jenkins.io/license/
[13] http://winstone.sourceforge.net/
[14] https://wiki.jenkins-ci.org/display/JENKINS/JUnit+Plugin
[15] https://wiki.jenkins-ci.org/display/JENKINS/Emma+Plugin

2.3 Arduino

Arduino is a platform consisting of easy to use open source soft- and hardware (Arduino Contributors, 2017b).

The Arduino software is a cross-platform Java integrated development environment (IDE) based on the IDE Processing[16]. The programming language is based on Wiring[17].

Arduino boards are programmed via the universal serial bus (USB). Since there is a bootloader preinstalled on the microcontroller, no external programming device is needed. The code editor of the Arduino IDE has a gcc compiler with additional avr-gcc and Arduino-libraries. Therefore the code can be written in a kind of C and C++ dialect. The IDE is able to compile the code and upload the binary directly to the board connected via USB. Instead of the common main function a program needs two functions to work properly, namely `setup()` and `loop()`. Readily identifiable the `setup()` is called once at the start of the program. This happens after uploading the compiled code to the board, by pressing the reset or power on the board. The `loop()` function runs continuously while the board is turned on.
The hardware of common Arduino boards is based on Atmel-AVR 8-bit microcontrollers. The power is supplied over USB (5V) or via an external power adapter (7-12V, $\geq$ 250mA).
Arduino boards provide digital I/O pins from the microcontroller. Some of them are able to output pulse-width modulation (PWM) signals. It has also a number of pins for Analog input and pins for 3.3V and 5V power supply. These pins can be used to control various electronic circuits. Because of the large number of different sensors and motors Arduino projects are diverse and can become complex. The range of project reaches from small circuits on a breadboard to embedded Internet of Things (IoT) applications to wearables or 3D printing. One reason for the Arduino platform's popularity (S. Contributors, 2017) is that it can be easily used by electronic and programming novices and is still flexible enough for advanced users. Therefore it can be used by freshmen to get in touch with the topic as well as sophisticated nerds to create any kind of standalone or integrated

[16] https://processing.org/
[17] http://wiring.org.co/

application. Another advantage is that Arduino boards are relatively cheap compared to other microcontroller platforms.

2.4 NFC

Near Field Communication (NFC) is an international communication standard based on radio frequency identification (RFID) technology. It is built on some of the RFID standards (Igoe, Coleman, and Jepson, 2014) with the goal to create a platform for multiple kinds of data and more complex exchanges between participants. Since it works as an extension, NFC readers can still read from and write to passive RFID tags.
NFC has a wide area of applications with the purpose to exchange data, e.g., swap records, exchange information and even initiate longer term communications through other technologies. Since NFC connections operate only on short range (less than 10 cm) and low power, there are almost no interferences with other devices. It is for short messages, initiating long term connection or exchanging credentials but not for high-speed communication. NFC devices can send simple or complex messages without the need for pairing or exchanging passwords. This may be risky, but when using NFC, the communicating devices only offers the basic exchange mechanism and the content is controlled, i.e., what and to whom data is sent.

According to Coskun, Ok, and Ozdenizci (2013) there are three main types of NFC devices.

- NFC-enabled mobile phone
  Most smartphones are equipped with NFC technology which increases acceptance, and potential use of this technology. These smartphones support all different NFC operation modes, which are briefly explained below.
- NFC reader
  As the name indicates, NFC readers receive data from other NFC components. One of the most frequently used NFC readers are contactless pay terminals in stores and supermarkets.
- NFC tag

NFC Tags are passive RFID tags, which means that they have only a small memory and no integrated power source.

Like RFID, NFC operations need a initiator and a target. Through the possibility to use smart or at least programmable devices it differs from RFID in the variety of use cases and capabilities. NFC targets can create unique content per message exchange and not only static data. To handle the exchanges, there exist three different operating modes for NFC enabled mobile devices.

- reader/writer
  This mode enables smartphones to exchange data with NFC tags.
- peer-to-peer
  Here data is directly exchanged between two NFC-enabled mobile devices.
- card emulation
  This mode enables to smartphone to be used as a smart card, e.g.,simulate a credit card to pay contactless in at NFC reader pay terminal.

The communication modes of NFC and regular RFID devices can be separated into:

- passive communication
  If the target has no own power source and gets their energy from radio frequency energy supplied by the initiator it is called passive communication mode.
- active communication
  In contrast to passive mode, in an active communication the initiator and the target have their own power source.

2.4.1 NDEF

The versatility of NFC comes from its format, the NFC Data Exchange Format (NDEF). This is one of the main benefits in comparison to RFID, because this common data format is used by all NFC devices, independent of the hardware. It is defined by the NFC Forum. At every data exchange,

Table 2.1: NDEF Record.

7	6	5	4	3	2	1	0
MB	ME	CF	SR	IL	TNF		
TYPE LENGTH							
PAYLOAD LENGTH 3							
PAYLOAD LENGTH 2							
PAYLOAD LENGTH 1							
PAYLOAD LENGTH 0							
ID LENGTH							
TYPE							
ID							
PAYLOAD							

one NDEF message is sent. This NDEF message consists of at least one NDEF record (Coskun, Ok, and Ozdenizci, 2013). This records can have a payload up to $2^{32} - 1$ bytes.

In Table 2.1 the bit wise structure of such a record is visualized. To understand the abbreviations and structure, it is following explained in detail.

- MB
  Bit no. 7, the MSB of the bit field is the Message Begin (MB) flag. It indicates the start of an NDEF message.
- ME
  Bit No. 6 is the Message End (ME) flag.
- CF
  Bit No. 5, the Chunk Flag (CF), indicates if it is the first or middle record chunk of a split payload.
- SR
  Bit No. 4, the Short Record (SR) flag, indicates a short record. The payload field is only 8 bit long (a single octet/byte) and can be set to a value between 0 and 255 meaning that the length of the records can be up to 255 bytes only.
- IL
  Bit No. 3, the ID Length (IL) flag, if set indicates that the ID LENGTH field is has 8 bits and hence hold values between 0 to 255 meaning the

length of the ID can be up to 255 bytes.

- TNF
  Bits No.2 − 0, the Type Name Format (TNF), form a three bit field which describes the content of the TYPE field. The represented content is one of these seven possibilities: a.) empty ($0x00$); b.) NFC Forum well-known type($0x01$); c.) Media-type($0x02$); d.) absolute URI($0x03$); e.) NFC Forum external Type($0x04$); f.) unknown($0x05$) or g.) unchanged($0x06$).
- TYPE_LENGTH
  The TYPE_LENGTH field has a size of eight bits. The stored value is an unsigned integer and specifies how many TYPE octet fields the record contains.
- PAYLOAD_LENGTH
  This field holds up to 32 bits (eight bytes). It is an unsigned integer and specifies the number of PAYLOAD bytes. The number of PAYLOAD LENGTH bits is affected by the SR flag, if the SR flag is set only eight bits (one byte), the PAYLOAD LENGTH 0 field, are used to specify the payload length and hence only up to 255 bytes can be used in as a payload.
- ID_LENGTH
  This field holds eight bits. It is an unsigned integer and specifies the number of ID field bytes.
- TYPE
  This field holds eight bits. It specifies the type of the payload.
- ID
  This field holds up to eight bits The ID field contains a URI reference which is used as an identifier.
- PAYLOAD
  The PAYLOAD field contains the payload for NDEF user applications.

2.4.2 ISO/IEC 14443

The norm 14443 of the international organisation for standardization (ISO) and the international electronical commission (IEC) specifies the physical and data relevant properties regarding the transmission between the reader

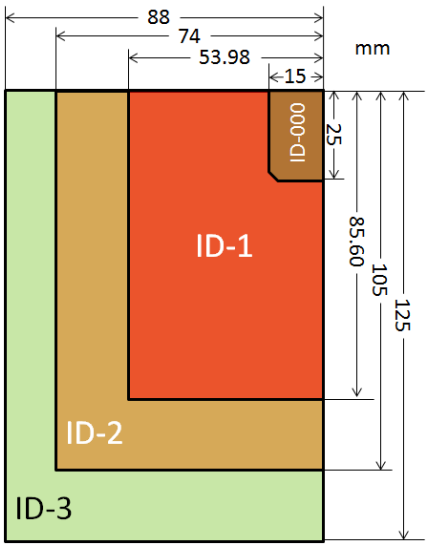

Figure 2.2: ISO7810 dimensions. ID-1 is used in ISO14443

and the contactless chip-card (GmbH, 2012). In this norm the reader is called proximity coupling device (PCD) and the contactless chip-card is called proximity integrated circuit card (PICC).

ISO/IEC 14443 norm specifications are separated into four parts.

The first ISO/IEC 14443 part (ISO/IEC, 1997) specifies the physical properties of the PICC. The norm says that the physical characteristics and the nominal dimensions have to be same as specified in ISO/IEC 7810 for card type ID-1. The specifications of the dimensions in ISO/IEC 7810 is displayed in figure 2.2, which shows that ID-1 meets the standard credit card dimensions. ID-3 has the dimensions of a passport, ID-2 of an identification card and ID-000 of a SIM card. There are additional characteristics specified in ISO/IEC 14443 – 1. Starting with the dynamic bending and torsional stress requirements, verified with testing methods from ISO/IEC 10373, as well as the requirement that the PICC has to function normally after (ISO/IEC, 1997, p.2):

> ”exposure of either face to medium-energy Xradiation, with energy 100 keV, of a cumulative dose of 0.1 Gy per year [...] exposure to a magnetic field of 12 A/m at 13,56 MHz [...]

> exposure to a electric field of average level [...]
> testing in accordance with the test methods described in ISO/IEC 10373 (IEC $1000 - 4 - 2 : 1995$), where the test voltage is 6kV [...]
> exposure to a static 640 kA/m magnetic field [...]
> over an ambient temperature range of 0 °C to 50 °C"

Part two is about the radio frequency power and signal interface (ISO/IEC, 1999). Here the ISO/IEC 14443 defines the frequency with $13,56$ MHz $+ -$ 7kHz. The consecutive operations of the initial dialogue of the PICC and the PCD are defined from ISO/IEC (1999, p.7) as follows:

> "- activation of the PICC by the RF operating field of the PCD
> - PICC waits silently for a command from PCD
> - transmission of a command by PCD
> - transmission of a response by PICC"

To couple the PICC, the PCD has to produce an energizing radio frequency field. This field shall be regulated for communication. There are differences between type A and type B communication.

Type A is defined to use amplitude-shift keying in the downlink and uplink between the PCD and the PICC. Amplitude-shift keying is a kind of amplitude modulation, which uses deviations in the amplitude of the carrier wave to represent digital data.
Type B is defined to use amplitude-shift keying in the downlink from the PCD and binary phase shift keying for the uplink.

The third part of the ISO/IEC 14443 (ISO/IEC, 2001) norm describes the communication at the data exchange between a PICC and a PCD. That includes the polling when a PICC enters a PCD operating field, the initial request command content and the answer to request command content. As well as the byte format of the frames and the timing , the initial dialogue from ISO/IEC $14443 - 2$ used for the communication. Further it defines the methods for anti-collision, to detect and communicate with only one proximity integrated circuit card among multiple other PICCs. Additionally required parameters for the communication initialization between the PCD and a PICC are also defined just as optional tools to facilitate and speed up anti-collision process resting upon application criteria.

The fourth and last part of ISO/IEC 14443 (ISO/IEC, 2000) specifies the transmission protocol. The appropriate transmission protocol for the contactless environment here is defined as a half-duplex block. With half-duplex the communication is limited to one direction at a time, which allows each party to communicate non-simultaneously with the other party.
Further on the protocol activation and deactivation sequence is defined.
The norm describes the different application of the transmission protocol for PICCs type A and type B.

2.4.3 PN532

A radio controller is the hardware core part of NFC communication regarding Igoe, Coleman, and Jepson (2014). One widely used NFC controller is the PN532. This popular NFC frontend was patented in 2007 by NXP (2007). As microcontroller basis a 80C51 core with 1 kilobyte (kB) of random access memory (RAM) and 40 kB of read only memory (ROM) is included.
Altogether this highly integrated transceiver module supports six distinct operating modes (NXP, 2012, p.1):

> ”
>
> - ISO/IEC 14443A/MIFARE Reader/Writer
> - FeliCa Reader/Writer
> - ISO/IEC 14443B Reader/Writer
> - ISO/IEC 14443A/MIFARE Card MIFARE Classic 1K or MIFARE Classic 4K card emulation mode
> - FeliCa Card emulation
> - ISO/IEC 18092, ECMA 340 Peer-to-Peer
>
> ”

According to the product data sheet of PN532 (NXP, 2012), the following features and benefits are present:
For ISO/IEC 14443A/MIFARE compatible transponders and cards, a decoder and demodulator for signals is implemented. A demodulator is the counterpart to the sender. It modulates the baseband signal to a device

specific format that the receiver can handle the input correctly.
The PN532 handles all error and framing detections of ISO/IEC 14443A. It supports higher transfer speeds at contactless communication in both directions with up to 424 kilobit/s using MIFARE. MIFARE Classic 1K and 4K card emulation mode is supported.
The PN532 provides the same functionality for FeliCa as for ISO/IEC 14443A/MIFARE to demodulate and decode the signals, handle the error and framing detection as well high speed communication.
Except the anti-collision, which have to be implemented in the firmware, PN532 supports the ISO/IEC 14443 B Reader/Writer communication scheme for layer 2 and layer 3. Other layers must be implemented in the firmware as well.
Both card interface schemes, FeliCa and ISO/IEC 14443A/MIFARE, can send Reader/Writer commands, which the PN532 in card emulation mode is capable to answer. For Reader/Writer or Card/proximity integrated circuit card(PICC) modes, it is possible to connect an external antenna to the PN532 without supplementary active components.
Following host interfaces are supported by the PN532 (NXP, 2012, p.2):

„

- SPI
- I2C
- High Speed UART (HSU)

"

To allow the direct connection of the device to a battery, an low-dropout voltage regulator is embedded.

3 Catrobat

One principle of the project is to develop in a test-driven way. This is an intransigent requirement, therefore all developers in the Catrobat team are encouraged to satisfy this. The tests can be separated into three categories. First are common JUnit tests, actual at Framework Version 4.1., which are mostly standard in a lot of projects. The second and little trickier test class is the User Interface (UI) testing, which was handled by using the Robotium Framework. These tests simulate user input by clicking on specific objects or positions on the screen. In this way it navigates through the interface and checks displayed text and dialogs on the screen.

Beside the previously mentioned tests categories using given frameworks, there is also the need for a bit more complicated testing class namely the hardware tests. These tests have to examine the real hardware and theirs functionality, like checking if the mobile phone really plays the sound when it should.

3.1 Why Test-Driven Development

Why does Catrobat is so strict on using Test-Driven Development (TDD)? There are already several publications about the benefits of TDD published on the Institute of Software Technology at the Graz University of Technology. Therefore this question is answered only in short here referring to these sources for further information.
In the work 'Aspects of Test-Driven Development' from Pulkit (2013) it is written that TDD has a positive influence on the quality of software design. This assertion is based on an experiment with three developer teams. One team had to follow the test first strategy, one the test last and the third team

had to write no tests at all. The outcome showed that the test first team completed the most requested features of the three teams. Furthermore the quality of the code was the best at the test first team. They had the smallest methods, modules and classes, what decreases the complexity and they had the highest code coverage rate.
In the Master's thesis of Slavec (2016) about 'Integration of controlling Arduino boards via Bluetooth with Pocket Code for iOS using test-driven development' he describes testing as a way to increase the satisfaction of the customer by writing more reliable software. A further important point about testing regarding Slavec (2016, p. 12) is that the quality estimation is only enhanced by successful tests, if they previously failed, what illustrates the removal of a fault.
Taking these advantages and the high fluctuation of people at the project into account, the decision to require TTD results as very clever to keep the code and testing quality high to guarantee a long-time success of the project.

3.2 Software Tests

Until January 2017 all user interface (UI) tests were written with the Robotium Framework (R. Contributors, 2017). Fortunately the number of tests, JUnit and UI, grows all the time. Because of the size of the application, there were about 1180 JUnit and over 1500 Robotium tests. Unfortunately some tests that should be tested with JUnit were written as UI tests, which have less impact on smaller project but have a large impact here, because in January 2017 all Robotium tests took already more than twelve hours to run once. To make the tests feasible again, the management and development of the project decided to switch to Espresso tests (Community, 2017). In the subsequent section 3.2.1 the reasonableness of this decision is explained, based on a Master Thesis with the topic 'Comparison of GUI testing tools for Android applications' from Lämsä (2017).

3.2.1 Robotium vs Espresso

Since the constantly growing number of user interface tests for the Pocket Code application the runtime of all test suites became impracticable long. For traceability reasons of the decision to switch to Espresso, following the two testing frameworks are compared regarding their assets and drawbacks. As there are hardly any scientific papers that treat with the direct comparison of Robotium and Espresso, the Master Thesis of Lämsä (2017) was the most profitable source and the comparison is therefore build on that outcome.

The functionality overview of Robotium is already summarized in chapter 2.2.2, additionally following is necessary to mention.
Robotium is derived from the instrumentation framework, which is a low level user interface testing tool with the opportunity to test single activities. The instrumentation framework gives the test developer lots of power but writing tests is very unwieldy. To abstract this difficult to use framework Robotium was built.
Same as Espresso it is an open source project that accesses methods and classes which are publicly accessible at the application under test(AUT) because the tests and the productive/tested code are in the same project.
Robotiums instrumented tests are executed in the same process as the application. It is built on the deprecated ActivityInstrumentationTestCase2 and the deprecated JUnit3 version of JUnit. Through the derivation of the instrumentation framework the only interactions that can be tested are within the specific application.
The interaction of a test with the device is possible within the `Solo` class. It provides various methods for the interaction and it is initialized by committing an instrumentation framework instance. A click on the Pocket Code play button is done with the command:

```
solo.clickOnView(solo.getView(R.id.button_play));
```

For test code development the same integrated development environment (IDE) as for the productive Android source code is used. The standard IDE for developing Android software is the Android Studio from IntelliJ[1].

[1] https://developer.android.com/studio/index.html

The procedure of test execution is the same as with Espresso. To run the test by use of the Android debug bridge (ADB) the AndroidJUnit Runner is used. At the first step .apk file is created by build the code.
The second step is the transfer of the application to the device.
Up next the .apk is installed and executed.
Finally the test suite is executed and the results of the tests are displayed in the IDE.

All these steps are done automatically by the Gradle file and Android Studio. The only thing the developer has to do is starting the test or a test suite. This makes it very comfortable to run the software tests.

Espresso works in a similar way as Robotium. It is deliberated for functional user interface tests especially for navigation within the AUT. It can also not be used outside the AUT. Writing tests in Espresso is fairly easy due to the fact that the tests are only executed if the application is stable. That implies that the developer does not need to think about timing of a progress when starting an intent.(Lämsä, 2017, p. 14)
The Espresso library comes with the Android Testing Support library[2]. Further it uses the AndroidJUnitRunner that allows developing tests comparable to JUnit test. These tests and the AUT are loaded by the test runner into the device, where the packages are executed. In contrast to Robotium version 3 and 4 of JUnit are supported.
It is also derived from the instrumentation framework which limits the execution of the tests to the actual AUT. Although Espresso is the only Android testing framework which autonomous handles the synchronization while testing. This eliminates every wait and sleep in the test code. This is accomplished by waiting for the main thread to be idle before continue executing the code. This approach is very fail secure, because all UI display updates in Android are handled by the main thread[3].
The structuring with annotations in Espresso tests is the same as for JUnit, which is already described in detail in chapter 2.2.1.
Test scripts using the Espresso library look quite different to Robotium test code. Here the action is a method performed on the view object. The same action as previous displayed in Robotium code, which clicks on the Pocket

[2]https://github.com/google/android-testing-support-library
[3]https://developer.android.com/training/testing/ui-testing/espresso-testing.html

Code play button, looks like this:

```
onView(withId(R.id.button_play)).perform(click());
```

The direct comparison of those two testing frameworks shows many similarities, like the derivation. Both are based on the instrumentation framework, which restricts them to AUT because they can only run in the process of the application. This makes it also necessary to run these tests on a real or emulated device.
The more current JUnit version is supported by Espresso, namely JUnit4.
Both support Android UI testing starting with API Level 8 (Android 2.2) and they use the AndroidJUnitRunner to execute commands through the ADB.
Robotium is unlike Espresso a 3rd party tool but both use Java as test development language.
Espresso can only be used for white box testing, Robotium in contrast can be used for black and white box testing.
The general trend, according to Google trend, goes to Espresso as displayed in Figure 3.1. The continuous downwards trend of Robotium is unmistakeable recognizable.

The most likely reason for that is the very different execution time. Espressos unique attribute regarding thread security is the crucial superiority. Lämsä (2017, p. 51) says

> ”Espresso proved to be significantly faster than the other tools in driving the UI of the applications”

To compare the execution time of various UI testing tool, he took several Android applications and UI testing frameworks, among Robotium and Espresso, and wrote equivalent tests for each application. The logged execution time and the robustness of each framework were then displayed in tables to compare them.
The first logged test suites for the applications 'Amaze File Manager' and 'Notes' revealed that Robotium tests took 488 seconds and the Espresso tests only 179 (Lämsä, 2017, p. 52). This points out that Robotium tests took 2.72 times longer than Espresso tests.

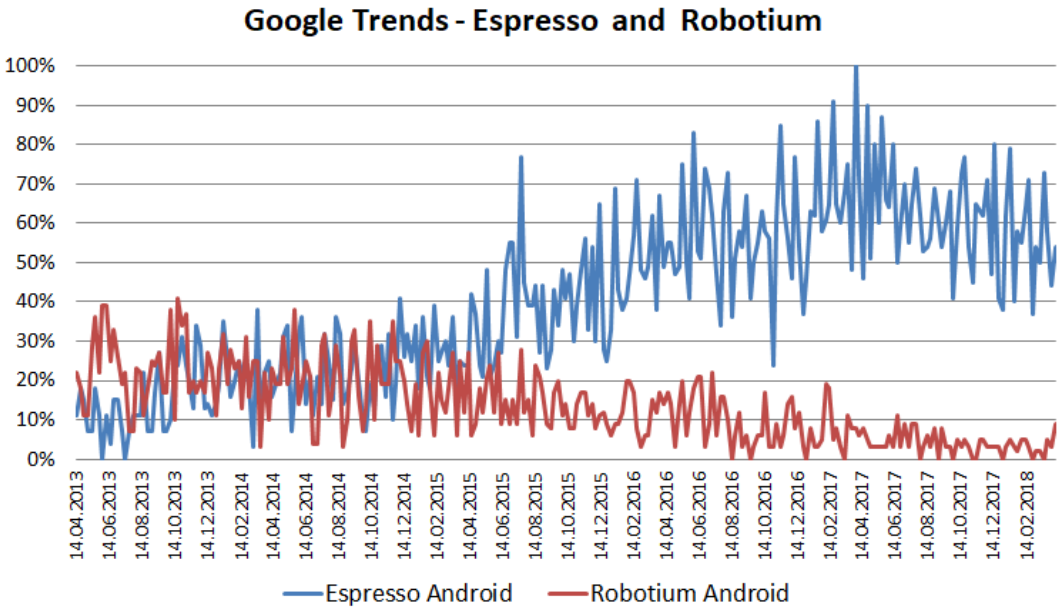

Figure 3.1: Google Trend: Robotium VS Espresso

It also turned out that another weakness of Robotium compared to Espresso is the reliability. With 1200 tests in the Espresso test suite and 1265 tests in the Robotium test suite the number of tests that fail (without a good reason) is about four times higher at Robotium tests (Lämsä, 2017, p. 65).

The biggest difference between Espresso and Robotium is however the number of waits in the code. Since there are none waits necessary in Espresso the output for number of waits for the two applications was 0 to 13.

Unfortunately it is not possible to deliver a meaningful comparison of the complete runtime of Robotium and Espresso tests for Pocket Code, because still not all Robotium tests are rewritten for Espresso and a main focus was also to convert unreasonable UI tests to JUnit if possible, to get an extra performance boost.
Anyway the execution time of Robotium and JUnit tests before the switching was more than twelve hours and had a high false negative (fails without a good reason) rate. Espresso and JUnit tests need only a fraction of that time and the amount of failing tests makes a dent.

3.2.2 Emulator vs Real Device

All written software tests that are developed have to be executed somewhere on a Android device. There are two different possibilities to run the test, on a real hardware device or on an emulated device.
The Android Emulator can be used, according to the official documentation of Android Contributors (2018), to simulate several Android devices like tablets, smartphones, Wear OS or Android TV on the computer without any physical Android component. This is feasible trough Android Virtual Devices (AVD), which can be configured regarding the used Android version, size, form factor and multiple other hardware characteristics. This way all relevant device types (the application is designed to run on) can be modelled as a separate AVD and used for testing.
Every virtual device created this way acts as an independent device. User data, installed apps and also SD card data are stored permanently to the AVD directory and is loaded when the emulator is launched.
In contrast the tests can also be executed on a real device. This includes all smartphones and tablets with enabled USB-Debugging in the developer options in the settings. The device only needs to be connected to the computer with this option enabled.
To make a decision what option should be preferred following the advantages regarding certain issues are confronted in Table 3.1. This table is a merged list from G. Contributors (2018) and Android Contributors (2018).

Beside these strong points both kinds also have their specific disadvantages. This negative points are listed in Table 3.2

After looking at the pros and cons of using real devices and emulators for testing Android software, it points out that both testing in both is mandatory to be able to guarantee a high quality and strong standard.
For this reasons Catrobat makes use of both techniques. The Emulator is irreplaceable for continuous integration since the Jenkins servers have to be able to make multiple test code executions simultaneous when several pull requests are made. In this case the performance factor is improved through parallelization.
On the other hand the emulator has very distinct boundaries when it goes

Table 3.1: Advantages of Real Devices VS Emulated Devices for specific issues

Issue	Real Device	Emulated Device
Availability	Strict performance testing issues like for waiter remote ordering applications with a continuous runtime of 8h cannot be emulated.	AVDs can be used for free, they just need to be downloaded and they are ready to run.
Situation based	Only real devices can be used testing applications in real situation context like handling during a sport activity or in the rain.	In particular situations there is no time to test in real situation context or specific phones are not available, then the simulation of these circumstances could be the best option
Device specific feeling	Testing usability issues like color, size or brightness at day and night gives a good look and feel of the application.	The wide variety of different Android devices for testing can hardly be covered with real devices. This is a perfect situation to use multiple AVDs for free.
Web Application	It is more expressive to test web applications on real devices relating to reliability.	Testing of opening web applications is easier since only the URL needs to be copied and pasted.
Make screenshots of bugs	The interoperability can be tested more meaningful and making screenshots is a standard feature on smartphones	Screenshots can either be captured by the computer OS or withing the Android Emulator.
Validation of battery scenarios	It is easily possible to test various battery scenarious.	In the Android Emulator the battery status can be adjusted in the settings, like the state of charge,
Interrupt handling	Incoming interrupts can easily be simulated and tested.	The later versions of the Android Emulator also simulate interrupts like incomming phone calls or SMS.
Color readout	Checking the displayed color on a real device is more meaningful	This can hardly be tested on the emulated device, because sunlight and reflection behaves different on the computer screen.
Performance	Since only the software is executed real devices are faster than emulators	Emulators have to simulate the hardware and the software, what makes them slower than real phones

Table 3.2: Disadvantages of Real Devices VS Emulated Devices

Real Devices	**Emulated Devices**
Building a proper testing cluster with different devices is quite expensive. Each test device must also be maintained.	Not all hardware features can be simulated, like NFC or using headphones Performance issues can hardly be tested on emulated devices.
Devices are or sometimes bound to specific countries and therefore they are hard to get.	Test execution takes longer, since emulated devices are slower than real ones.
Real devices can be harder to connect with the computer and the IDE what can cause time-consuming problems.	Factors like battery consumption or overheating cannot be tested.
To test with real devices a USB port is used permanently. This can be another point of failure.	Setting up well configured emulators is costly in terms of time.
Adequate security is necessary, especially if expensive devices are used, to prevent theft.	Unexpected behaviour like defect hardware components cannot be simulated.

to test hardware features. Near Field Communication (NFC) is only testable on real devices. The topic of this Master Thesis is the NFC extension for Catrobat. Due to the fact that the Catrobat project is built on test-driven development it is mandatory to also connect a real smartphone to the Jenkins servers to be able to test NFC features.

Summarizing this section, testing on real devices is the preferred method when it comes to test implemented hardware features. For testing functional code the Android Emulator is a powerful tool. The UX decisions in this project are made by the UX team therefore this usually important aspect can be neglected when it comes to the automation of testing.

3.3 Hardware Tests

A special feature of Pocket Code is the possibility to work with almost all sensors and features of a smartphone. To test some of these hardware features the Arduino hardware testing box is necessary. The construction of the box is explained in detail in section 4.2 in the Implementation part of this thesis.
This section explains how to make use of the hardware testing box in Espresso tests. Since testing the functionality of hardware is most reasonable in the stage activity, this cannot be done by JUnit tests but UI tests.
The hardware testing box IP address is configured to be in the subnet of the Jenkins servers. This is necessary for the test servers to be able to work with the box. For Pocket Code developers this means that new hardware tests can either only be executed on the Jenkins or the one who wants to verify/falsify new tests can use the second box, which can be configured to work in any network. The configuration of the box is explained in detail in section 4.2.6.

The box can be used to test five hardware features of a smartphone. In Pocket Code this testing functionality is located in the public class `SensorTestArduinoServerConnection.java`.
The first testable functionality is the reading of NFC tags. Therefor the Espresso test need to call the Java function `emulateNfcTag(boolean writable, String tagId,String ndefMsg)` what describes the content of the

tag, which is then emulated by the Arduino hardware testing box. A NFC tag must contain information about if it is writeable or not. This is for the emulation in Pocket Code of minor importance, since there is no impact on the testing for now.
Further the tag ID has to be specified as hex string. This ID is important for the mapping of read tags to already known tags. In Pocket Code known tags can be used in program logic with the `WhenNfcBrick`, which is executed if the previously set NFC tag ID equals the ID of the new NFC intent.
The last parameter of the function is the NFC message. The emulation is preconfigured to add the message as universal resource identifier (URI). This NFC Data Exchange Format (NDEF) message enables the representation resources over a network like websites or network files. A detailed explanation of NDEF is available in section 2.4.1. The content of the last NFC intent NDEF message and the tag ID as well can also be used/accessed via the NFC sensor variables, which is explained later in detail in section 4.1.1.

The second testable feature is the vibration. For this purpose the Arduino hardware testing box checks if the smartphone actually vibrates. The Java function `checkVibrationSensorValue(int expected, int timeoutMillis)` for verification/falsification takes two parameters. The value `expected` is 0 (`OFF`) if there is no vibration expected and 1 (`ON`) if there should be a vibration. `timeoutMillis` describes for how many milliseconds the function should try to get a positive response from the Arduino box.
Additionally the function `calibrateVibrationSensor()` should be called prior the actual checking, to guarantee a accurate result.

The third functionality to test is the LED light of the smartphone. The handling in Java is similar to the vibration check.
`checkLightSensorValue(int expected, int timeoutMillis)` checks for a number of `timeoutMillis` milliseconds if the 0 (`OFF`) or 1 (`ON`) value of `expected` is returned from the Arduino hardware test. Here is no prior calibration necessary, since the read sensor values are significant.

Checking if an audio signal is transmitted via the auxiliary output of the smartphone is the fourth testable hardware functionality. The `checkAudioSensorValue(int expected, int timeoutMillis)` function in

`SensorTestArduinoServerConnection.java` is like the light check. The parameters are the same and there is no calibration necessary for the Java developers. Though the runtime of the check on the Arduino lasts two seconds, because the audio signal is detected via the wave oscillation of the signal where two seconds are necessary to function properly.

The last feature is the check of the network access. This is tested automatically by running any of the previous tests. Since the Arduino hardware testing box acts as a server, the smartphone has to establish a socket connection to send the specific requests. Therefore the WLAN connection of the smartphone is coincident used and checked.
More details regarding the background of the hardware tests on the Arduino hardware testing box can be found in section 4.2

3.4 Workflow

In Pocket Code there is a strict workflow which every developer has to go through. Every work respectively issues or problems have to be committed as a so called ticket. The used ticketing system for the Catrobat project is Jira[4].
A ticket has to contain all necessary information about the work to be done. This can be a description of a new feature which has to be implemented as well as a report of a bug which must be fixed.
Bugs can be reported by everyone without a Jira account. To create a ticket of another type, the project lead has to create a Jira account and grant the required rights.
Every developer needs a ticket to start the productive work. This ticket goes through the predefined workflow displayed in Figure 3.2.

The blue coloured states have a dual function. The first role is to act as a queue for following (yellow) progress states. This is like a 'to do' state prior the progress states. The second role is a 'done' state after the progress states. The yellow coloured states are 'in progress' markers. This means that it is actively worked on the ticket, either through writing code, creating a design

[4]https://www.atlassian.com/software/jira

or reviewing code and/or design.
The green coloured states are final states. Tickets in a final state need no further processing. There are three final states, the successful done MERGED and the denied DUPLICATE and REJECTED states.

During the Jira workflow these following states are passed through:

- ISSUES POOL
  The Jira workflow starts with the ISSUES POOL. Here are all new created tickets as well as older tickets that are not merged, rejected or not a duplicate. The ISSUES POOL is cleaned up at least once a year at the so called ticket party, where members of each sub-teams come together to discuss the need of every single ticket. This is necessary, to keep the work order oversee able.
  Tickets from the issue pool can then be moved to the final states DUPLICATE, if for example a bug is reported twice, or REJECTED, if for example a feature request is denied, and it can also be moved to the BACKLOG queue. These transitions can only be accomplished by administrators, which are the coordinators of the project, and senior members, which gain this status by working for Catrobat since more than 1000 hours.
- BACKLOG
  The BACKLOG has multiple purposes. It serves as container for tickets which are relevant for the next release or planning game. The effort estimation has to be performed on the tickets as well as the prioritization of these. To be able to do effective effort estimation, prior a detailed ticket description has to be added with well-defined acceptance criteria.
  To get this detailed information, sometimes the interaction with the UX consultants is necessary. This is the case, if layout relevant changes or extensions are planned like new bricks, changes in the menu and so on.
  Finally the BACKLOG is a reserve if the READY FOR DEVELOPMENT container is empty before the next planning game.
  Thus tickets from the BACKLOG can be forwarded to the queue states UX BACKLOG or READY FOR DEVELOPMENT as well as to the final states REJECTED and DUPLICATE.
  The transition to the READY FOR DEVELOPMENT state can only be accomplished by administrators because the team has to discuss the need and

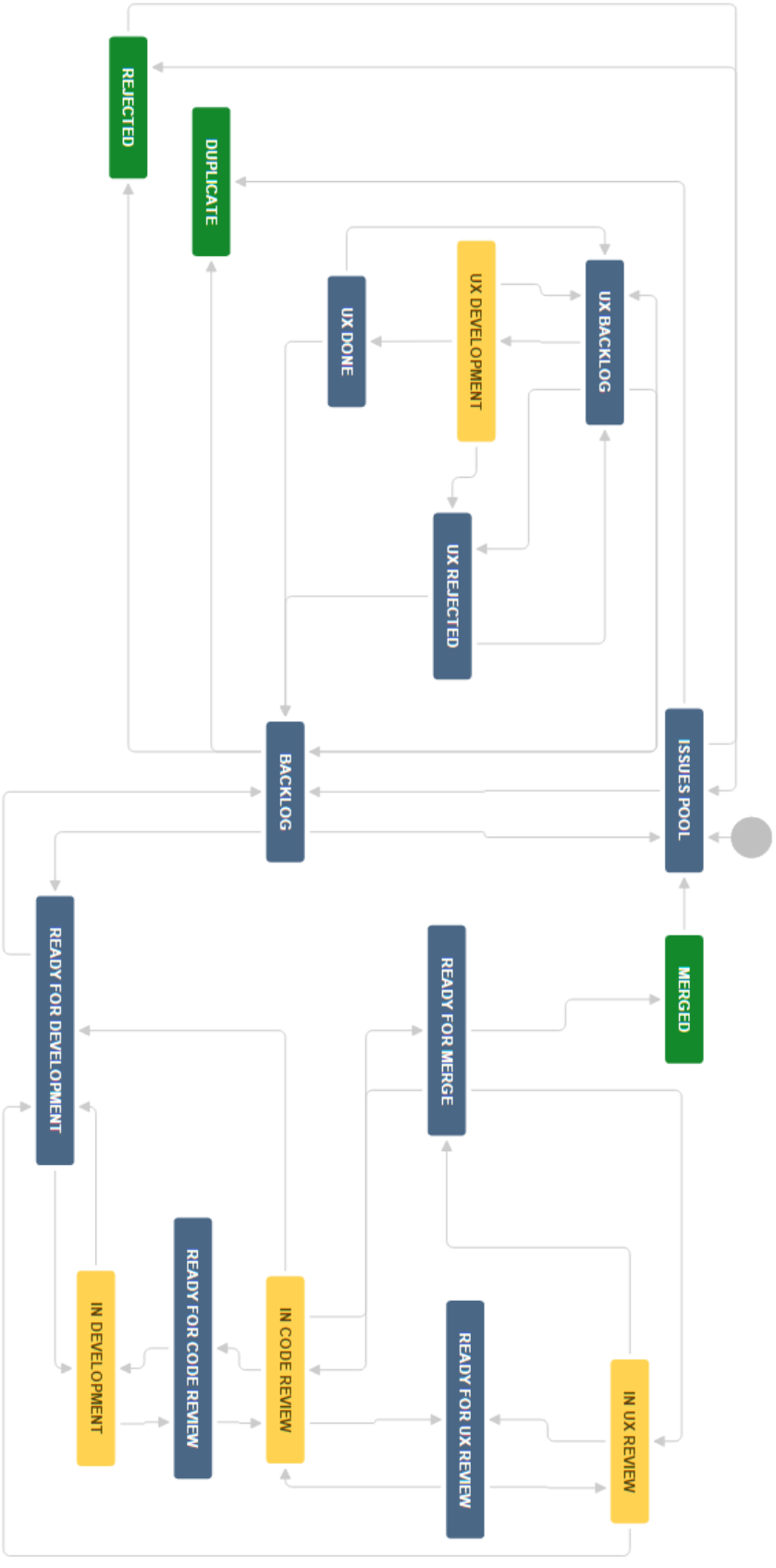

Figure 3.2: Pocket Code Workflow

realization of each ticket. The forwarding to the final states and the UX BACKLOG can be carried out by administrators and seniors.

- READY FOR DEVELOPMENT

  READY FOR DEVELOPMENT is a queue similar to the former BACKLOG. It only contains tickets that are well described, prioritized and effort estimated.
  Every member of the team can access this queue and assign single tickets to themselves and forward them to IN DEVELOPMENT, but only administrators and sensors can put them back to the BACKLOG.
- IN DEVELOPMENT

  This state describes that the ticket is actual worked on. It has to be assigned to exactly one member, but can be processed by multiple members. Here the main work of the developers is done. All defined ticket, respectively issue specifications have to be implemented under abidance of all development restrictions like agile software development. A ticket may only be in the IN DEVELOPMENT state during it is actively worked on. If the work on a ticket is paused for longer time it has to be put back to READY FOR DEVELOPMENT that other developers can continue working on it. As soon as the development phase is completed, the ticket has to be set to READY FOR CODE REVIEW by the developer. Finally a GIT pull-request can be applied.
- READY FOR CODE REVIEW

  The READY FOR CODE REVIEW state is a queue between the IN DEVELOPMENT and IN CODE REVIEW states. The assigned developer can put it back to IN DEVELOPMENT but only administrators and seniors can conduct the code review and therefore only they can forward the ticket to IN CODE REVIEW.
- IN CODE REVIEW

  The status is set to IN CODE REVIEW, if a code reviewer (as previously mentioned an administrator or senior member) actively checks the code. If the review of the ticket is not finished it can be set back to READY FOR CODE REVIEW. Otherwise if the review is finished and there are changes necessary, the ticket is put back to READY FOR DEVELOPMENT. If the implementation is okay and there are no UX adaptations, the ticket is forwarded to READY FOR MERGE. If there has been any UX adaptations it forwarded to READY FOR UX REVIEW.
- READY FOR UX REVIEW

This queue is processed by members of the usability team. One member is assigned to the ticket and the status has to be changed to `IN UX REVIEW`.

- IN UX REVIEW

  The usability team member does the review and the changes can then be accepted and the ticket forwarded to `READY FOR MERGE` or declined and the ticket is put back to `READY FOR DEVELOPMENT`.

- READY FOR MERGE

  `READY FOR MERGE` is the last queue state in the implementation workflow. Only senior members and administrators have the permission to merge a feature branch into the develop or master branch. To avoid problems, it is necessary to have the merged feature up to date with the current codebase of the branch it is merged to.

- MERGED

  The status of the ticket is set to `MERGED`, if the feature is successfully merged into the master branch. In most cases the ticket workflow ends here and the work is done. In some cases the already merged features need further treatment for example if a fix does not eliminates the problem on specific devices.

- DUPLICATE

  A ticket gets marked as `DUPLICATE` as soon as it is unmistakeable that another ticket with the same content already exists. Duplicates occur sometimes if the same bug is reported by different persons with an different description. Especially non-programmers tend to use unequal descriptions for the same thing. Only administrators and seniors can set the ticket status to `DUPLICATE`. The duplicates should be unmasked at the latest at the next planning game when the issue is discussed by several members. Once a ticket is set to the final state `DUPLICATE`, it cannot be touched and changed anymore.

- REJECTED

  The `REJECTED` state is similar to the `DUPLICATE` state. This status can only be set by administrators and seniors. The decision to reject a ticket is mostly made during the ticket party or the planing game. In contrast to the `DUPLICATE` state the ticket can still be edited and moved back to the `ISSUES POOL` for further processing.

- UX BACKLOG

Tickets with impact on the user interface respectively the user experience go through some more stages. Before such a ticket is set to READY FOR DEVELOPMENT the design and behaviour for the corresponding elements has to be drafted by members of the usability team. Therefore the ticket is moved to the UX BACKLOG by an senior member or administrator. Usability team members can then access this queue to do their work.

- UX REJECTED

  If the planned features or changes, described in the ticket, do not go along with the opinion of the usability team, they are set to UX REJECTED. Therefore this queue is to decide if the ticked is moved back to the BACKLOG either of the project or of the UX team. Anyhow there are changes necessary in the specification of the ticket.
- UX DEVELOPMENT

  The developers of the user experience team have to take tickets from the UX BACKLOG and set their state to UX DEVELOPMENT before they start processing the orders. The UX developer can set the state of the ticket to any UX state.
- UX DONE

  UX DONE is the last state of the UX workflow. If a ticket is set UX DONE, the usability part is approved. This is requirement for UX tickets to be forwarded from the BACKLOG to READY FOR DEVELOPMENT.
  If the ticket is in UX DONE, it can be forwarded by usability team members to the BACKLOG or put back to the UX BACKLOG for further adaptations.

3.5 Use Arduino in Pocket Code

Pocket Code supports the control of a number of external devices like Raspberry Pi, Lego NXT and Arduino. Following the usage of Arduino in Pocket Code is explained representative for all other supported features, since the steps are similar and the Arduino hardware is a essential part of this thesis. All external hardware is connected via Bluetooth to the smartphone except for Raspberry Pi, which is connected via wireless local area network. Most Arduino boards do not support Bluetooth out of the box. There is a special Arduino BT (Bluetooth) microcontroller (Arduino Contributors, 2018) which has a build-in Bluetooth module WT11-A. This Arduino BT is like the

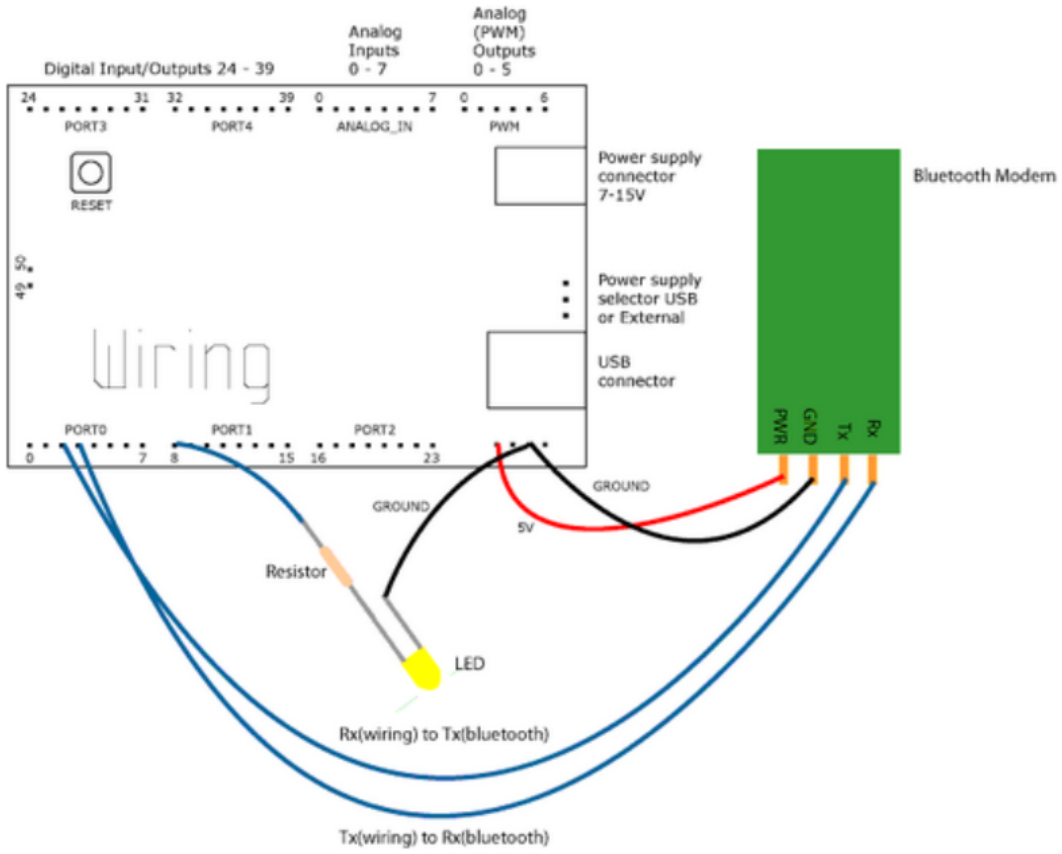

Figure 3.3: Arduino Bluetooth wiring

Diecimila Arduino board. It can be powered with 2.5V up to maximum 12V. It has 14 digital pins and 6 analog pins. The factory Bluetooth connection setting is named ARDUINOBT with password 12345. However this product is already retired from the official Arduino range of goods.
Another solution is to make a standard board Bluetooth capable. The best way to do so is using an external Bluetooth module. A common module for this is the HC-05 Wireless Bluetooth Serial Module (Reichelt, 2018).
It works with serial communication and is powered with 5V by the Arduino. For the serial communication, between the board and the module, the Rx and Tx pins are used. The via Bluetooth received data is send through the Tx (Transmitter) pin on the module to a Rx (Receiver) pin in the Arduino.

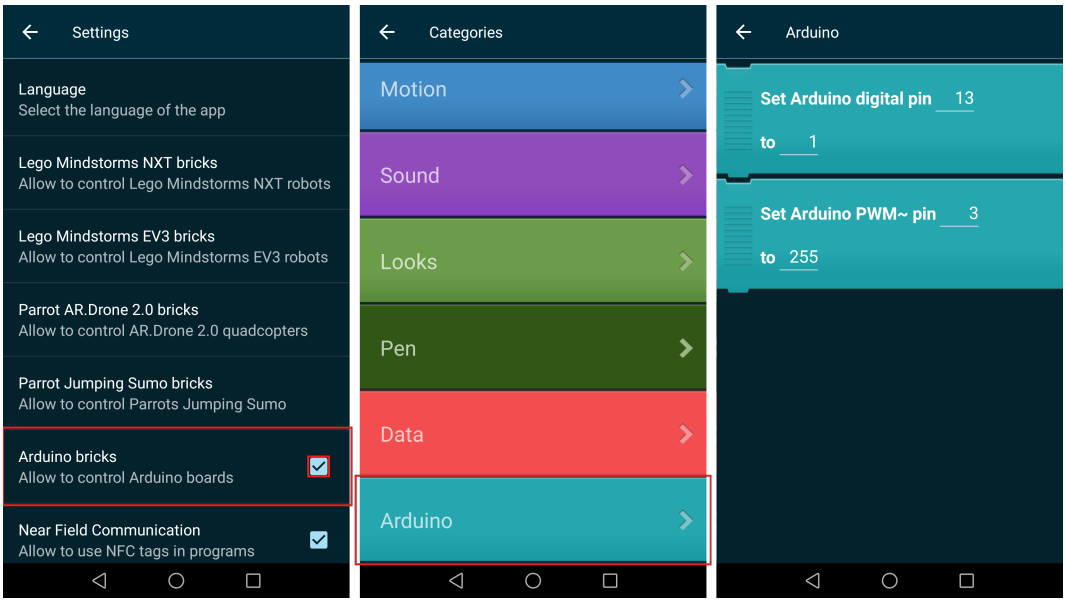

Figure 3.4: Enable Arduino bricks

Vice versa data is send through the Arduino Tx to the Bluetooth module Rx and finally to the Bluetooth connected device. The described wiring of a HC-05 module is displayed in Figure 3.3.
The factory Bluetooth connection is named 'HC−05−' followed by the MAC-address and the standard password is 1234.
The first step to write an Arduino program in Pocket Code is to enable the Arduino bricks in the main menu settings. This setting is displayed in Figure 3.4 on the left. As soon as the checkbox is enabled, an additional category is available when pressing the add brick button in the script activity. This Arduino category and the content of the same is shown in Figure 3.4 in the middle screenshot and the right one.
Both bricks can set actions by changing the output value of a pin. At first the requested pin number has to be entered. In the second place the value of the requested pin. Digital pins can only be set to 0 (`LOW`) and 1 (`HIGH`). If a higher number is entered, it is interpreted as `HIGH`. The power width modulation pins, in other words the analog pins, can handle values from 0 to 255. 255 is the maximum value, because Arduino is a 8-bit microcontroller ($2^8 = 256$).
In the background the Arduino commands `digitalWrite()` respectively

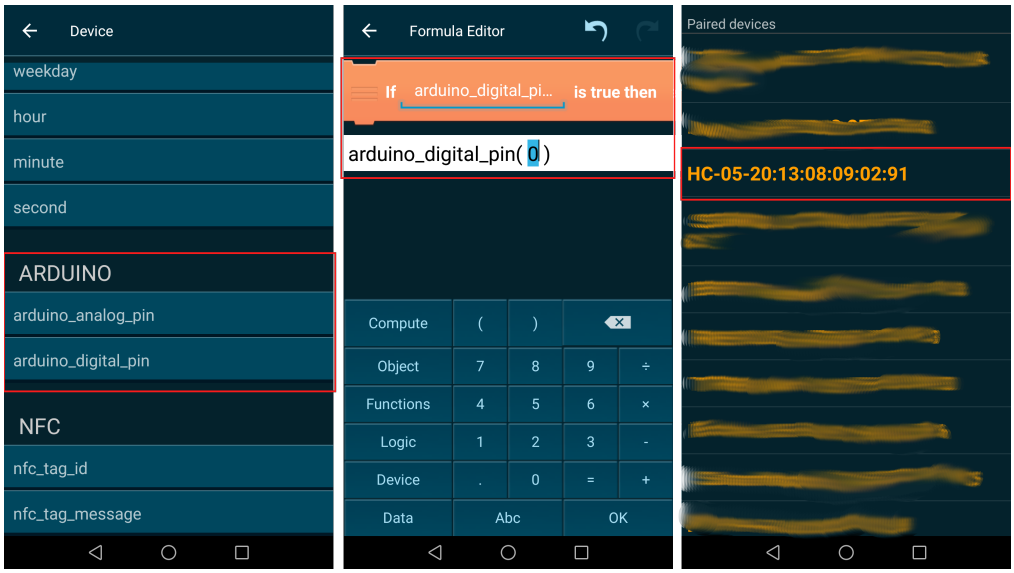

Figure 3.5: Arduino device variable and connection

`analogWrite()` are executed with the typed in parameters.
The second Arduino functionalities in Pocket Code are the device variables `arduino_analog_pin` and `arduino_digital_pin`. As complement to the previous explained write commands, these variables can be used to read out input pins. In Figure 3.5 the left screenshot shows these two variables in the device variables folder. The middle screenshot shows the digital pin number input field in the formula editor for the readout of a digital pin value to control a `IF` condition. Here only the pin number needs to be entered and the requested values are then send from the Arduino to the smartphone. The device variables are equivalent to the Arduino commands `digitalRead()` and `analogRead()`.
As soon as the play button is pressed to start the stage activity, Pocket Code checks if an Arduino is connected. If there is no active connection, the pairing activity starts before the stage activity. In Figure 3.5 on the right screenshot this activity is displayed with a highlighting of the Arduino Bluetooth module. As previously mentioned the name of the device is 'HC−05−' followed by the MAC-address '20 : 13 : 08 : 09 : 02 : 91'. After choosing a device and entering the password the stage activity starts and the program is executed.

4 Implementation

Since the kick-off of the Catrobat project in 2010, Test-Driven Development became more and more important. It is a non-negotiable matter regarding this project.
With the main focus on implementing only testable code, the extension with Near Field Communication (NFC) features is described in this chapter. To test NFC and other hardware features like sound or the integrated flashlight as best as possible, a special solution is shown. Therefore this chapter further describes in detail how an Arduino board, equipped with sensors, is used as server to check all hardware activities of a smartphone that cannot be tested on the Jenkins server on an emulated device.

4.1 NFC

Near Field Communication is already well-integrated into daily routines. According to Coskun, Ok, and Ozdenizci (2013) it is an enabler for ubiquitous computing which simplifies human - machine as well as human - human interaction and the application areas are still increasing.
To add Pocket Code to this application area, the following chapters describe an extended integration of NFC technology. Instead of recognising tags only, the application is also enhanced by reading and creating of NDEF messages during runtime in the stage activity.

4.1.1 NFC sensor variables

Pocket Code has a very powerful formula editor. Beside standard arithmetic operations he has the opportunity to use specific values corresponding

the actual object like transparency or position settings as numerical values. Various functions like mathematical functions with logarithm, trigonometric functions, string related functions like length and concatenation and list functions. Further Boolean operators like `AND`, `NOT`, `TRUE`, `FALSE` and comparison operators like $\neq$ and $\geq$. Moreover user variables and user lists can be accessed through the formula editor and therefore used in calculations. The last container is summarized with the name device variables.

Device variables are further divided into several subcategories. The standard category is called device sensors. This collection provides all internal measurement unit (IMU) sensor values as acceleration in x,y or z, compass or inclination and also global position system (GPS) related sensor values for latitude and longitude.
Other standard subcategories are touch detection, face detection and date and time.
Beside this phone related sensors, Pocket Code is compatible Lego NXT, Lego EV3, Phiro, Arduino, Parrot AR Drone and the Parrot Jumping Sumo drone. For each of these external hardware devices several sensor variables are provided.

When Near Field Communication was activated there have not been any NFC related variables yet. To be able to work in the stage activity with current read NFC values new sensor variables have to be added.
The first important NFC sensor variable is the last read NFC tag identification (ID). It is available with the same being called name `nfc_tag_id`. To provide a meaningful value the SensorHandler class has to be changed to allow a string as return value, because a NFC tag ID is usually displayed as hexadecimal value. To store the ID of the last read NFC tag, the NfcHandler class is used. It processes each NFC intent and is therefore perfect to store the last read values. The ID is delivered as byte array, following converted into hexadecimal numbers and finally stored as string with this format. This string then looks like $0687CB75AE1000$, which is the seven byte UID of a Maestro debit card.

The payload of the read NFC tag is written into the `nfc_tag_message` device sensor variable. Therefore the NFC intent is scanned towards NFC Data Exchange Format (NDEF) messages. More information about NDEF messages can be found in subsection 2.4.1 NDEF. When the tag contains multiple

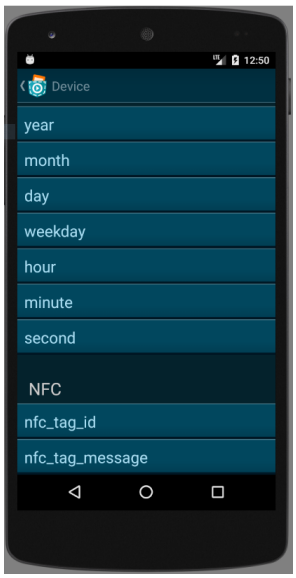

Figure 4.1: NFC device sensor variables

messages, due to display reasons only the first message is used. Its payload is concatenated character by character. This only happens, if the tag is not encrypted and the NDEF messages are stored in plain text, otherwise the sensor variable is set to an empty string.
The NDEF type of the message is not readout in the Pocket Code application, because this information is considered irrelevant for children which start to learn programming.

It was previously mentioned that the sensor variable is stored each time the NfcHandler is called to process the intent. Additional it is important to annotate when a NFC intent is processed in Pocket Code.
To act in java on a NFC intent at all, a NFC adapter object from the library `android.nfc.NfcAdapter` has to be initialized. This adapter is used to enable a foreground dispatch for the current activity. This means that when a NFC tag is read, the intent is not processed by the operation system, but by the current running activity. Important to consider on enabling the foreground dispatch, when leaving the activity, it has be disabled. Otherwise it leads to unexpected behaviour.
In Pocket Code the foreground dispatch is enabled in the NfcTagFragment

and in the StageActivity. As a consequence the two NFC device sensor variables are updated, if a tag is read in the NfcTagFragment and when the Pocket Code program is started and therefore in the StageActivity.

4.1.2 Set NFC tag brick

The former NFC implementations in Pocket Code target to read from a NFC tag. The first NFC extension was the WhenNfcBrick. The Brick has a spinner that allows selecting NFC tags from a list, which is filled with scanned tags that have been added to the NFC fragment list previously. The chosen tag is the condition that the attached bricks are executed, as soon as the chosen tag is scanned in the stage activity.

The above in subsection 4.1.1 described device sensor variable `nfc_tag_message` was the next step to use some advantages of NFC tags and NFC Data Exchange Format (NDEF) messages in Pocket Code.
To tap more potential from NFC it is necessary to further extend the range of function of the application. The reading of NFC tags is a powerful property in combination with using them as variables in a running program. Nonetheless it is a useful extension to enable Pocket Code users to write to NFC tags.
Such functionality is provided by a new brick. Each new brick underlies design criteria and has to be coordinated with the user experience (UX) team of the Catrobat project. The decided layout is, as displayed in figure 4.2, a brick with three lines. The first line is composed out of a TextView label with the text 'Set next NFC tag to' and an EditText element. EditText elements open the formula editor when tapped. This is displayed in figure4.4. The layout is stored in the Extensible Markup Language (XML) file `brick_set_nfc_tag.xml`.
The second line is a simple TextView label with the text 'as NDEF record type'. It contains a scrollable spinner with NDEF record types.
As default value, when the brick is added to the script, the EditText element contains the string 'www.catrobat.org' and the spinner is set to `HTTPS`.

All selectable elements from the spinner are explained in table 4.1.

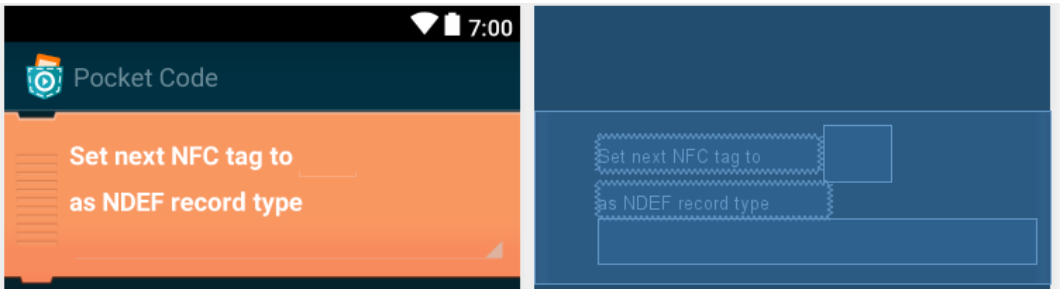

Figure 4.2: SetNfcTag brick layout

Table 4.1: SetNfcTagBrick spinner NDEF record types

ID	**Name**	**TNF**	**Type**
0	Text	TNF_MIME_MEDIA	"text/plain"
1	HTTP	TNF_WELL_KNOWN	RTD_URI
2	HTTPS	TNF_WELL_KNOWN	RTD_URI
3	SMS	TNF_EXTERNAL_TYPE	nfclab.com:smsService
4	Phonenumber	TNF_WELL_KNOWN	RTD_URI
5	Mailto	TNF_WELL_KNOWN	RTD_URI
6	External type	NFC Forum external type	catrobat.com:catroid
7	Empty	TNF_EMPTY	{}

ID	**Payload**[0]	**Final NDEF message with 'www.catrobat.org', '+436641234567' or 'contact@catrobat.org'**
0	message[0]	· · · text/plainwww.catrobat.org
1	0*x*03	· · · U·www.catrobat.org
2	0*x*04	· · · U·www.catrobat.org
3	smsMessage[0]	· · (nfclab.com:smsServicesms:+436641234567?body= SMS from Catrobat
4	0*x*05	· · · U·+436641234567
5	0*x*06	· · · U·contact@catrobat.org
6	{}	· · · catrobat.com:catroid
7	{}	· · ·

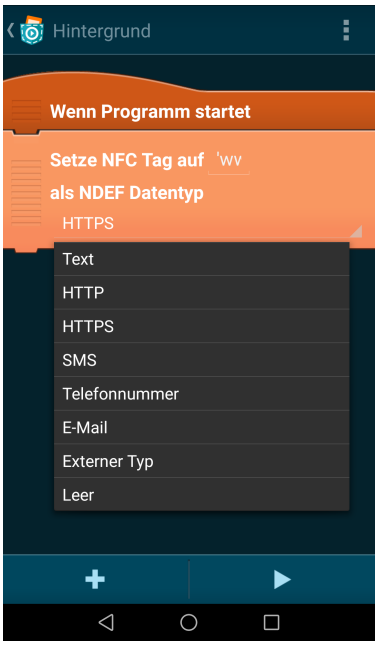

Figure 4.3: SetNfcTag brick

Figure 4.4: Tag message in formula editor

The final brick with extended spinner is shown in figure 4.3.
As displayed in table 4.1, there are different type name format (TNF) types used. `TNF_WELL_KNOWN` are standard NDEF types which are identified by the reader through the payload prefix. Every NFC capable mobile operating system has predefined actions like at '$0x04$' open the browser with the unified resource identifier (URI) string from the message plus a `HTTPS://` prefix. With the prefix '$0x06$' the standard mail client program is opened and the mail address in the payload is set as receiver.
The spinner option `External Type` is a Pocket Code specific setting. Android has the opportunity to refer to a Google Play Store applications. If an application should be able to be referenced, a specific intent filter has to be added to the Android manifest file. In Pocket Code the manifest file has been extended by following lines:

```
<intent-filter>
    <action android:name="android.nfc.action.NDEF_DISCOVERED" />
    <category android:name="android.intent.category.DEFAULT" />
```

```
    <data
        android:scheme="vnd.android.nfc"
        android:host="ext"
        android:pathPrefix="/catrobat.com:catroid" />
</intent-filter>
```

The `pathPrefix` in the last line allows Android to refer to Pocket Code when an intent with the domain `catrobat.com` and type `catroid` is called. This means, when a NFC tag has the TNF 'NFC Forum external type' with type `catrobat.com:catroid`, the payload is in this particular case negligible, the operating system opens Pocket Code if it is installed. Otherwise, if it is not already installed, it opens the Google Play Store direct with the download site of Pocket Code.

To handle the usage of multiple SetNfcTag bricks a special queue is necessary. Since the application runs on multiple threads and the brick can be used by several objects at the same time, it has to be a thread secure synchronized queue. Java provides such a requested data structure, called `BlockingDeque`. Each time a `SetNfcTag` brick is executed, the corresponding NDEF message is added last to the queue.
The `onNewIntent` function in the stage activity, which is executed each time a NFC tag is read, checks if the queue is not empty and runs the `NfcHandler.writeTag` function with the polled NDEF message.
Finally the phone specific NFC control sound is played, as soon as the writing process is completed.

Consequently Pocket Code is now equipped with useful NFC extensions to read NDEF messages from NFC tags and able to create new NDEF messages, which can be written to any writeable NFC tag, in a child friendly way.

4.1.3 NFC tag Bluetooth pairing

The principle idea behind NFC technology is always to simplify things and/or speed things up. One common usage of NFC in everyday life is to shorten up the pairing of Bluetooth devices.
Pocket Code has multiple brick extensions to be able to control various

Table 4.2: Bluetooth Secure Simple Pairing record

Description	**Content**
Record type	Bluetooth Secure Simple Pairing record
Type Name Format (TNF)	Multipurpose Internet Mail Extensions
Type	application/vnd.bluetooth.ep.oob
MAC address	$98 : D3 : 31 : 90 : 96 : 5E$
Complete local name	$PHIRO-00-041$

hardware. One of those supported hardware is the Phiro Pro of Robotix Learning Solution[1]. This robotic car is a platform that targets to help teach computer science for children ages 9 and above[2]. To run the Pocket Code program on the Phiro, a Bluetooth connection has to established. This is done by switching to Bluetooth mode on the vehicle and subsequent the right Bluetooth universally unique identifier (UUID) has to be chosen.
This pairing process is easy to accomplish, if the number of found Bluetooth devices is little or even limited to a single Phiro robot. Though the field of application targets also to be used in school classes, where multiple Phiros and other Bluetooth-able devices are in range. It can lead to time-consuming problems if pupils connect to wrong devices. Furthermore it can be demotivating for young people in learning time if the hardware shows now reaction.
To handle this problem, NFC tags can be used to store information about the Bluetooth connection to enable an easy pairing process.
The content of such a so called Bluetooth Secure Simple Pairing record is shown in Table 4.2. A NFC tag sticker with the appropriate information can be put on every Phiro which eliminates the risk to pair the wrong device.
To support this simplified Bluetooth pairing process in Pocket Code, some adaptations have to be done.
The task was to create a button in the ConnectBluetoothDeviceActivity to allow the users to activate and deactivate the pairing option via NFC. Regarding the guidelines of the Catrobat project, all visual modifications must be done in agreement with the user experience (UX) team. The final layout of the not activated button is displayed in Figure 4.5. It is the same

[1] http://robotixedu.com/
[2] http://robotixedu.com/phiro/

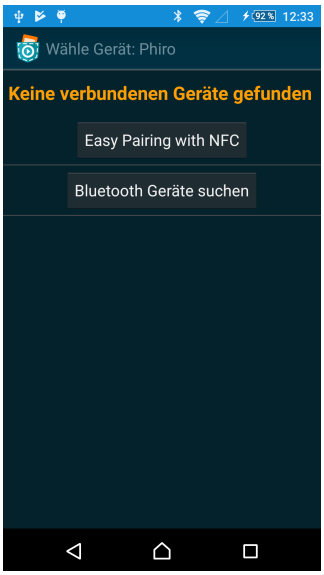

Figure 4.5: Pairing selection

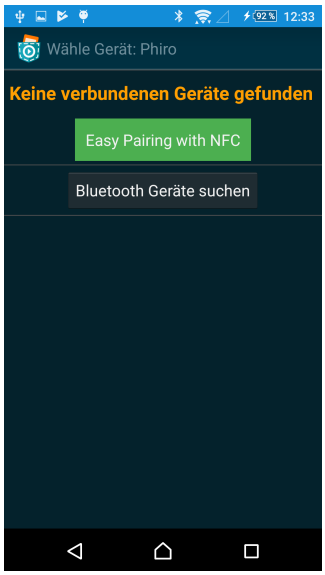

Figure 4.6: NFC pairing activated

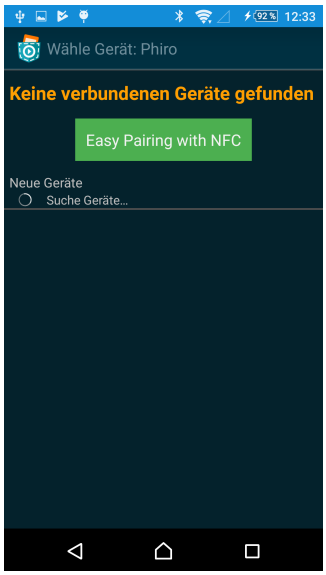

Figure 4.7: Search for Bluetooth devices

as the already existing scanButton. The layout of the pressed button is displayed in Figure 4.6. In comparison the searchButton disappears when pressed, as displayed in Figure 4.7.

The ConnectBluetoothDeviceActivity (Figure 4.5) starts every time if a Pocket Code program is executed for the first time and if it contains Lego, Phiro and/or Arduino bricks. By default the NFC foreground dispatch is disabled when ConnectBluetoothDeviceActivity starts. Briefly explained a foreground dispatch in Android controls where an intent is processed. A more detailed explanation can be found in Subsection 4.1.1.
The foreground dispatch is deliberately only enabled when the "Easy Pairing with NFC" Button (Figure 4.6) is pressed because of the following reasons. The user is informed that this functionality is now available and enabled. On the other hand NFC pairing is neither necessary nor used for all Bluetooth devices.
This is meaningful, because the NFC adapter has to be enabled to dispatch the foreground. As a consequence the NFC functionality has to be activated. If a user wants to connect a Bluetooth device without any NFC activity, a pop-up with NFC network settings would be annoying and misleading.

Because of this the NFC adapter is only enabled on an explicit request of the user, by pressing the ”Easy Pairing with NFC” Button.

By pressing the enabled/green ”Easy Pairing with NFC” button again, the foreground dispatch is disabled again.
If the smartphone has no NFC chip, the button is coloured red and when pressed it displays the hint that there is no NFC adapter available.

The pairing with the Phiros is secured with a standard password. This password has to be entered at the first pairing process, when it is paired via the Bluetooth settings. Because of security reasons it is not possible to add the password information to the NFC tag. It is not convenient to force the user to enter a password, if the NFC pairing process should be a simplification.

The used solution for this problem is to store the password hard-coded in Pocket Code. The converted password byte array is then added at the `android.bluetooth.BluetoothDevice.ACTION_PAIRING_REQUEST` if the `android.bluetooth.BluetoothDevice.PAIRING_VARIANT_PIN` is necessary.

4.2 Arduino Hardware Testing Box

Android thus Android Studio provides good possibilities to use virtual hardware within an emulator, but as in subsection 3.2.2 already shown, a hardware device is needed to improve the stability of an application. It is essential to simulate hardware intends in Android and therefore this method is preferred at writing Pocket Code tests.
Beside the ability to test with simulated hardware intends the Pocked Code application should also be automatically tested with real hardware. Because there is no ready to use solution available, it was necessary to create a special solution for the Catrobat project.

The idea of the hardware testing box is, to have a compact server mounted with all necessary sensors. This server box should be in the same local area sub-network as the Jenkins testing server. The wired Ethernet connection allows the Jenkins server to communicate with the sensor server.

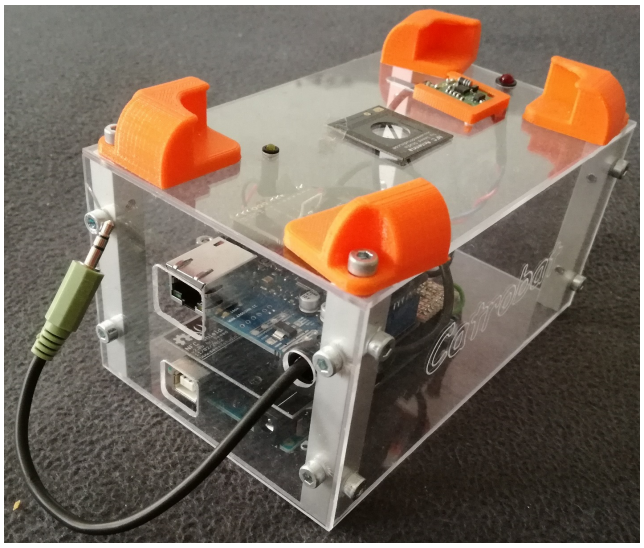

Figure 4.8: Hardware testing box with all necessary sensors

To keep the costs and the space consumption low, an Arduino Mega 2560[3] forms the basis computing unit. Since Pocket Code covers a wide range of hardware features like flash light, vibration, play sound and read/write NFC, the functionality of these features also needs to be controlled by automatic tests.
The Arduino has to be capable to test each of these functionalities. For that reason multiple extension shields are needed additionally. The first requirement is to make the Arduino network-compatible to act as a server. In the Arduino store a plug-and-play Ethernet shield has been offered with the corresponding library *Ethernet.h*. The NFC-Shield is bought from www.seeedstudio.com and combined with the library *PN*532. The audio shield as well as the shield with light and vibration sensor is self-made.

4.2.1 Container Construction

In the planning phase of the container, the first step was to collect all requirements towards the testing box and dimensions of the used parts.

[3]https://www.arduino.cc/en/Main/ArduinoBoardMega

Length and width of the Arduino parts in comparison to the dimension of modern smartphones is negligible because they are far bigger. The height was determined by measuring the fully assembled Arduino components and shields.
As next step the position of flash light LED on the smartphone must be measured precisely. Since the angular of divergence of the flash light is small, the position of the photocell is important. For the NFC antenna only a small hole is necessary. The cable which connects the shield and the antenna has a plug on both ends, what makes it possible to just put the wire through top of the box and connect the antenna element afterwards.
The vibration sensor by contrast is soldered with the cable. Its hole in the top therefore has to be bigger.
The audio cable for the audio measures has the 3.5mm jack on the box outside end and is soldered on the shield. Therefore it is necessary to make a hole large enough that the jack fits through.

Additional to the basis height of the fully assembled Arduino components it is important to consider the position of the sensors. The final height inside the box must be large enough to be able to easily put the sensors through the designated holes in the top panel.

The next design criterion was the access to the interior of the box.
The easiest way to realize would be a screwed on bottom plate. On this plate the Arduino could be fixed to guarantee the security of the hardware. While taking into account that the cables are long enough, it could be possible to thread the sensors through the top of the box before putting the Arduino into and fasten the bottom in place. Since here is only one part of the box flexible, it is easy to make the box robust.
This design is not feasible because of the used NFC antenna. The cable to connect the shield with the antenna is only about twelve centimetres[4] long and there is no longer cable available. This makes it very uncomfortable to connect the cable with the antenna or the shield every time the box is opened for any reason.

The problem with the short NFC antenna cable can be avoided by making the top plate detachable. This option makes it easier to connect the sensors with the shield when the box is put together. It is also easier to disconnect

[4] https://www.seeedstudio.com/NFC-Antenna-p-1805.html

the sensors controlled when the box is opened for any reason.
The drawback of this solution is the maintainability of the Arduino and the access to the pins especially of the lower shields. If the lowest shield is glued to the base there is access hardly possible. If the hardware is screwed on the base, good fine-motor skills are needed, because only thin screws can be used which fit the small bores in the shield.

This leads to the insight that from the maintainability point of view the most sustainable solution has to be a box where each single side can be detached separately. This should also motivate future developers to handle the box and fix possible problems regarding the Arduino hardware testing box. For example when a shield breaks, a sensor outside the box is damaged or new hardware should be added.

Before creating a detailed construction plan, the material for the walls and the inner frame had to be determined.
A good material to tinker with would be wood. It is an easy to get sustainable commodity and, depending on the timber species, easy to process. Furthermore it is cheap, sturdy and the tools for the processing process are widely used and easy to get. If the wood is handled right, the result could also look like a high quality design product.
The problem with wood on the other hand is that stable hardwood is difficult to handle and the solid wood plates are too large in diameter. Also the box should look special and modern to represent the characteristics of the Catrobat project.

One main property of the project is that it is open source. Considering this aspect in the design of the box, it is a good statement to make the walls transparent. Normal glass is not an option, because it breaks too easy during processing or usage and it is too expensive.
The resulting decision is to use acrylic glass. It is cheaper and more flexible than normal glass. Nonetheless it is relatively difficult to process to get a satisfying result.
After the wall material is determined, the material for the inner frame has to be chosen. Since the outer parts should be screwed on the frame, acrylic glass is not an option, because it would get scratched while drilling the female thread for the screw. Wood is also not the ideal material as frame, because the female threads wear off fast and the screws loose grip.

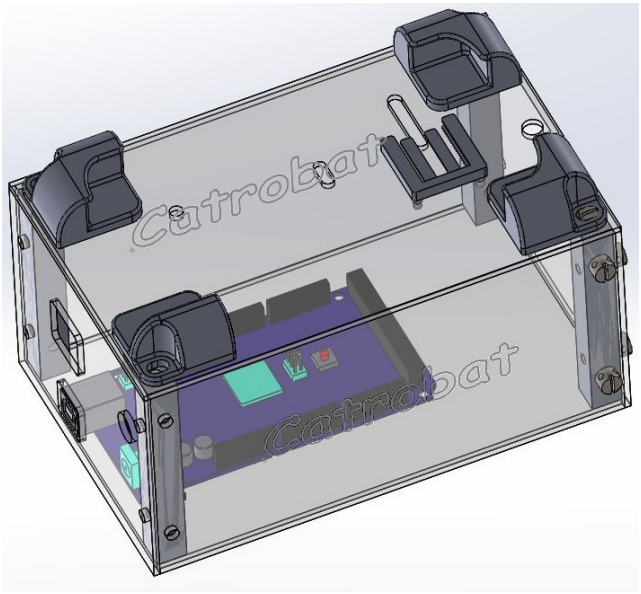

Figure 4.9: Hardware testing box plan in Solidworks

A light and affordable matter but still robust, perfect for this use case, is aluminium. It is available at the hardware store[5] as bar with different thicknesses. To be as precise as possible the whole box is planned in Solidworks Student Edition[6]. The final construction plan is displayed in fig The final dimensions for the acrylic glass box are 100 x 160 x 80 mm (length x depth x height) with a wall thickness of 2 mm. To gain the needed accuracy at the production process a computerized numerical control (CNC) machine is used. For that all wall elements are exported in the Drawing Interchange File Format (DXF) separately. These files are imported into Estlcam[7].
Estlcam is a CNC software, which can generate tap files. It allows to place the previous exported Solidworks files on a plane and set specific CNC settings like diameter and rotation speed of the mill cutter, the speed the machine moves in x, y and z direction and how many millimetres in depth it has to take at once.

[5]https://www.obi.at/profile/ba-stange-vierkant-natur-10-mm-x-10-mm-x-1000-mm/p/2437523
[6]http://www.solidworks.at/sw/education/student-software-3d-mcad.htm
[7]http://estlcam.com/

The final tap code consists of one command per line, which describes where to move in the three-dimensional space(X,Y,Z) and with what speed(F). The following code snippet describes the drilling of one whole for the screw. The variables I and J are polar coordinates.

```
G00 X10.0000 Y69.4527 Z5.0000
G00 Z0.5000
G01 Z-1.0000 F50
G02 X9.4527 Y70.0000 I0.0000 J0.5473 F300
G02 X10.0000 Y70.5473 I0.5473 J0.0000
G02 X10.5473 Y70.0000 I0.0000 J-0.5473
G02 X10.0000 Y69.4527 I-0.5473 J0.0000
G01 Z-2.0000 F50
```

The produced tap file is executed on the computer of the CNC machine. A software named Mach3[8] interprets and controls the motors. After a few failures where the acrylic glass coalesce with the mill cutter, it turned out that it is possible to get the hang of the problem by using soap water to cool the material during the processing process. The Arduino-, NFC- and Ethernet-shields are fixed on the bottom plate with multiple threaded rods and nuts to adjust the height.

The mounting system for the smartphone on the top has the specification to hold a LG Nexus 4 or a LG Nexus 5. To manage this, the retainer has to be adjustable, because the LG Nexus 4 has dimensions of 133.9 x 68.7 x 9.1 mm and the LG Nexus 5 has dimensions of 137.9 x 69.2 x 8.6 mm. This results in a difference of 0.5 mm in the width and 4 mm on the long side of the phone. The used solution for this problem was to use a 3D printed retainer on each edge, which is fixed with the screw of the top acrylic glass plate. Each part is adjustable about the recess for the screw with 2.5 x 0.3 mm (long side x short side). In Figure 4.10 the printed retainers are shown. The colour orange is chosen in the style of the Catrobat cat.

For the final part of the container the 3D printer is needed once more.
The vibration sensor needs to be pressed against the smartphone to measure the vibration correct. The sensor should not be placed on an inflexible object,

[8] http://www.machsupport.com/software/mach3/

Figure 4.10: 3D printed mount parts

because this prevents the transfer of the vibration from the smartphone to the sensor.
It took a couple of tries, until a solution was flexible enough to obtain good sensor measures on the one hand and fix the sensor at the position on the other.
The answer for this problem is a retainer, which holds the sensor on 3 sides with enough gaps. Additional a self-bent spring at the bottom pushes the vibration sensor against the smartphone. This spring is initially printed as two even stripes connected with the middle part of the retainer. Under heat and pressure they are curved to function as spring.

The finished box with the screws on the outside, the smartphone retainer on the top, the Arduino shields inside, without a smartphone, has a dimension of 108 x 168 x 100 mm (length x depth x height) and a weight of 391 gram.

4.2.2 Audio Shield

The general purpose of the audio shield is to detect if sound signals are send via the auxiliary output of the phone.
It is not meaningful to directly connect the audio output from the phone with the Arduino, because the basic audio signal is a wave consisting of positive and negative wave parts. This leads to an electrical AC audio signal,

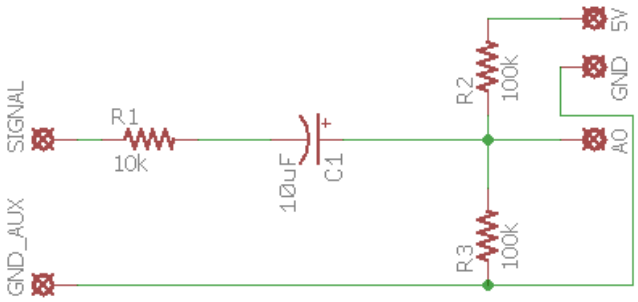

Figure 4.11: Audio board schematic circuit

which cannot be handled by the Arduino, because the analog to digital converter can measure only positive voltages.

To begin with the hardware implementation, the circuit layout is displayed in Figure 4.11. It is a simple solution with three resistors and one capacitor. As connection between the phone and the audio shield a 3.5 mm auxiliary cable is used. One end of this cable is cut off and the wires are spliced to get the mono signal wire and the auxiliary ground. Further the spliced end of the cable is soldered on the shield.
The jack has to be outside of the box. The auxiliary input plug on the used Nexus 4 is on the top of the phone, therefore the hole for the cable is beside the recess for the USB and Ethernet connection. The length of the cable is chosen that it can reach any possible auxiliary input plug position in case the used mobile device changes.
The board is 55.88 mm x 35.56 mm (length x depth) in size with 24 drills for pins to be solid attached to the Ethernet shield below. The Eagle software files are available on the Catrobat Confluence page.

Within the software implementation the processing of the incoming audio signal is handled. The pin drills on the audio shield are named respective the Arduino board convention, thereof it is obvious in Fig4.12 that the auxiliary signal goes to the analog input 0 on the Arduino. This signal is an integer between 0 and 1023 related to the frequency of the audio signal level.
Pocket Code can play sounds in the sound fragment and in the stage activity.

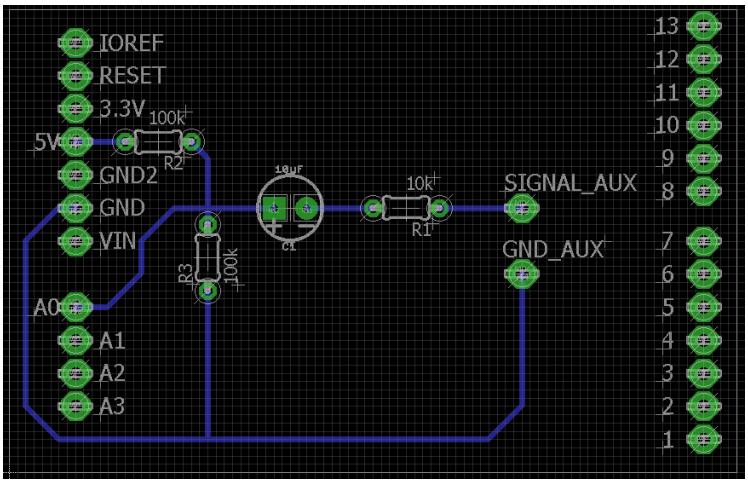

Figure 4.12: Audio board print plan

To handle time delays, such as the start of the stage activity when the play button is pressed, the audio signal is measured for a time period of two seconds. A sufficient accuracy for the frequency of measures for this application is ten samples per second. The Nyquist-Shannon sampling theorem supports this sampling frequency, because the used sounds change the pitch less often, therefore a higher measure rate per second has no advantage. Fewer measures in contrast corrupt the result, because peaks in the signal are missed in some cases.
The check if a sound is played respectively received is done with the computation of the variance σ^2.

$$\sigma^2 = \frac{\sum_{i=1}^{n} (sound(i) - \overline{sound})^2}{n}$$

To calculate the variance, the first step is to collect a set of samples by calling $sound()$, in this case two seconds with ten measures per second are collected, which results in a set of 20 measures $n = 20$.
The next step is to calculate the mean value $\overline{sound}$ by summing up the measures.
The second last step is to sum up the squared difference of the measures and the mean value, which is finally divided by the number of measurements n to obtain the variance.

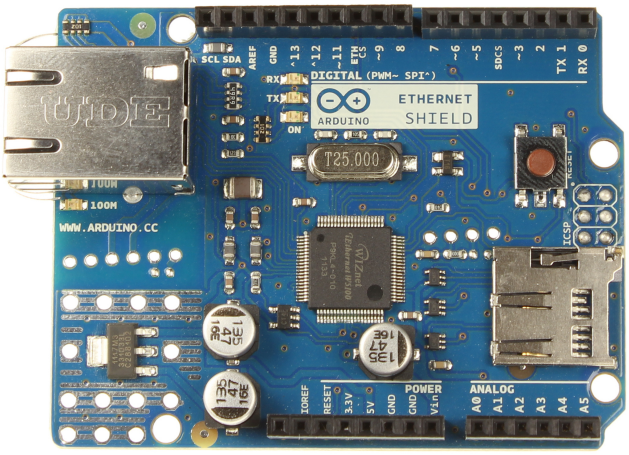

Figure 4.13: Arduino Ethernet Shield

4.2.3 Ethernet Shield

The Arduino Ethernet Shield is the connecting element between the Arduino and the Jenkins server. It is a plug and play shield delivered with a library (`Ethernet.h`) which allows the Arduino to connect with a local area network. The board acts as a server to accept incoming connections from the Jenkins. Its capacity is limited to four concurrent connections.
There are few hardware specifications which are worth mentioning.
The Arduino Ethernet Shield uses a standard W5100 Ethernet controller chip[9] as basis.

The Shield communicates with the Arduino board with the SPI bus. Therefor the Arduino Mega2560 has the Slave Select (SS) pin per default on pin 10 (Arduino Contributors, 2017a). This SS pin is responsible for enabling or disabling specific devices. When the SS pin is high, the Ethernet shield ignores the master, When the SS pin low, the shield communicates with the master. The MAC addresses of the Ethernet Shields must be enabled to join the Jenkins server local area network subnet to establish a connection.
The Arduino hardware testing box assigned Internet Protocol (IP) address is `192,168,8,8`. The Zentrale Informations Dienst (ZID) of the TU Graz au-

[9]http://www.wiznet.io/product-item/w5100/

Table 4.3: Sensor specific commands receivable with the Ethernet shield

Command	Identifier
NFC tag emulation	0
Vibration measurement	1
Light measurement	2
Vibration calibration	3
Audio measurement	4

thorized the MAC addresses `90:a2:da:0f:15:57` and `90:a2:da:0f:15:0f` in the firewall settings to join the Jenkins sub network.

The software implementation is done by means of the `Ethernet.h` library provided functions.
In the `setup()` function the `EthernetServer` object is instantiated with the above described IP and MAC settings.
In the `loop()` function the `EthernetServer` object is checked continuous, with a small time delay, if it got a request from a client. In our case the requester is the mobile phone testing device connected with the Jenkins server. It requests a sensor check from the Arduino and therefore an `EthernetClient` object is created. The request of this client is checked and processed. This means the sensor specific commands are executed.
The five possible commands are listed in Table 4.3

After all computation is done, the response of the check is sent to the `EthernetClient` object and the connection is terminated. This ends one iteration of the `loop()` function and the process starts again with waiting for the next incoming request.

4.2.4 NFC Shield

To test some of the new implemented NFC features in Pocket Code, Java provides the possibility to simulate NFC intents. This feature is nice to begin with when writing tests for NFC, but the tests can only be executed on devices with a NFC adapter.
Therefore it is impractical for the use in combination with running the tests

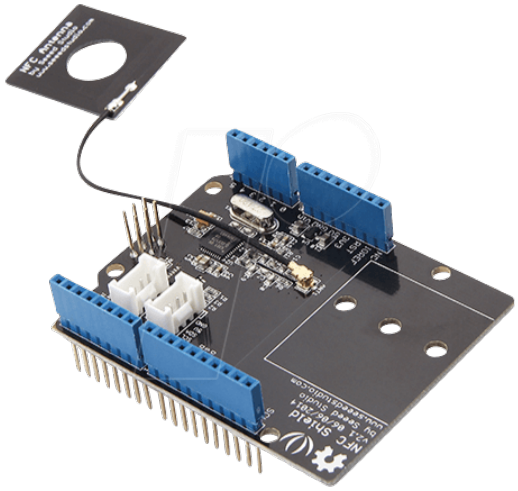

Figure 4.14: Arduino NFC Shield V2.0b

on an emulated Android device. This also means that this tests cannot be run on the Jenkins servers by default.
As for the testing of the other hardware features described in this chapter, a hardware device is needed. But not all NFC features of Pocket Code can be tested with program driven intents. An intent provides the opportunity (Android Contributors, 2017) to launch activities at any time during runtime. Intents mainly consist out of a passive data structure which describe the execution of an action. The described functionality excludes the possibility to test the SetNfcTag brick with intents only.

At this part the need for an external NFC device comes up. There exist several NFC extensions for Arduino. Some of them have the reader fixed on the shield, which is not usable for this specific solution, because the smart phone is outside the container.
The standard Arduino NFC reader has to be connected via seven wires. One for power ($3.3V$), ground (GND), reset (RST), serial data (SDA), master output slave input (MOSI), master input slave output (MISO) and serial clock (SCK). The usage of this NFC reader makes the wiring inside the box confusing and clutters the organized layout.
Another solution could be the NFC Shield V2.0b from seeedstudio. As described by SurveyMonkey (2017) it uses the ICSP headers for SPI. That

implies that the applied Arduino Mega2560 is compatible with the shield. The communication frequency is, as standard for NFC, 13.56MHz.
Through using the SPI protocol, the shield needs only four pins for full control, instead of seven, like the standard Arduino NFC reader does. Further the antenna is wired with only one cable. A drawback is that the length of the cable is only twelve centimetres. However the cable has simple push- and pull-plugs on both ends, which make it easy to connect the shield to the antenna.

A further specification of the shield is that it is connected with 5 volt and its average power consumption is 100 milliamperes.
The range of the antenna is specified with 5 cm, which is far more than needed in this setup.
Further it supports peer-to-peer(P2P) communication, which means that it is able to communicate with another NFC shield as equally privileged participant via the antenna. Therefore the ISO14443 Type A and Type B protocols are supported. A detailed description of ISO14443 is in section 2.4.2.
Same as at the Ethernet shield, this board has the SPI SS on Pin $D10$. Therefore one of the boards has to be modified. The simplest solution, which is chosen for this project, is to disconnect the $D10$ pin from one shield through bending them aside. Afterwards a cable connects pin $D10$ on the board with pin $D9$ on the Arduino.
The second option was more difficult and risky. The NFC shield offers the opportunity to remove the SS pad on the connection to pin $D10$ and solder it on the $D9$ pad. Additionally the connection to pin $D10$ has to be scraped off.

The integration into the Arduino code is handled with the three additional libraries `PN532_SPI.h`, `emulatetag.h` and `NdefMessage.h`. In the Arduino `setup()` function the SPI handling is done. The Ethernet shield and the NFC shield are connected with SPI to the main board. But only one shield can be active with Arduino at a time. Thus the Ethernet shield pin $D10$ has to be set to `HIGH` if the NFC shield is used with pin $D10$ and vice versa.

Additional to the identifier number from table 4.3 a NFC emulation command needs further information. This information has to be send at a single

Table 4.4: NFC command structure from the Ethernet client

length	1 byte	1 byte	6 byte	max 128 bytes
description	ID	writeable	tag UID	NDEF message
example 1	0	0	123456	https://www.catrobat.org
example 2	0	1	987654	https://www.tugraz.at/home/

blow after the identifier 0 from the Ethernet client for instance the Jenkins server. The layout of the needed request is shown in the table 4.4.

4.2.5 Light and Vibration Sensor

There two more testable hardware features left. One is the flashlight LED of the camera and the second feature is the vibration of the smartphone. They are integrated on a single shield together with an additional functionality, namely the status led. To combine this three functionalities, at first each feature has to be considered on its own.

To measure light a photo-resistor is used as sensor. A photo-resistor is a resistor whose resistance is controlled via the input of light. In the dark the resistor value is at its maximum and it decreases with increasing light intensity.
Here a $10k\Omega$ photo-resistor is used in combination with a $10k\Omega$ pull-up resistor prior. The pull-up resistor is a normal resistor which is used to hold, if the light is turned off, the input voltage level on the ADC pins above the specified threshold of 500. ADC pins on a microcontroller are able to convert analog voltage into a digital number, therefore the name analog to digital converter (ADC). If the flashlight LED of the smartphone is turned on, the input voltage level drops to low value below the threshold.

To avoid inaccurate measures the positioning of the light is of the greatest importance. Smartphone flashlights have a small angular for the light expansion. In combination with the very small distance, between the build in sensor on the topside of the hardware testing box and the smartphone, of a few millimetres there is hardly a margin in positioning.
Fortunately the position of the flashlight LED on the used phones (Nexus

4, Nexus 5) is very similar what makes it possible to use both phones for testing.

The vibration measurement is accomplished with an accelerometer. The used sensor module from Neuhold-Elektronik[10] has a size of 20x20 mm. The board consists of a ADXL330KCPZ, a voltage controller, a reverse polarity protection diode, a tantalum capacitor and ceramic capacitors. The description of the used acceleration sensor ADXL330KCPZ from the manufacturer Analog Devices is described in (A. D. Contributors, 2007, p.1) in the following way:

> "The $ADXL330$ is a small, thin, low power, complete three axis accelerometer with signal conditioned voltage outputs, all on a single monolithic IC. The product measures acceleration with a minimum full-scale range of $\pm 3g$. It can measure the static acceleration of gravity in tilt-sensing applications, as well as dynamic acceleration resulting from motion, shock, or vibration."

The sensor is very sensitive regarding all inclination changes and even small movements, like the vibration of a smartphone, are determined. The module is hold in position in a 3D printed mounting. The sensor has enough mobility in this mounting and is pressed toward the smartphone with a spring, which is also printed as part of the mount.
After holding the sensor module against a vibrating phone, an evaluation of the analog sensor values turned out that the most vibration was on the z-axis. Due to this here only the z-axis measurements are used for determining a vibration. Each vibration check collects a set of 2048 samples in an average time of 2.3 seconds, because of a 1 millisecond delay after each analog read of the z-value. After the collection, the minimal and maximal values are searched by iterating of the collected data array. Finally the maximal minus the minimal value has to be greater than a threshold.
To ensure a flawless usage of the sensor, the threshold to determine a vibration has to be set dynamically. Therefore the function `vibrationCalibration` has a similar behaviour as even described, but has to be run in a non-vibrating situation. It stores the difference of the minimum and maximum value as threshold for the vibration check.
The schematic setup of the vibration sensor circuit is displayed in figure

[10] https://www.neuhold-elektronik.at/catshop/product_info.php?products_id=3586

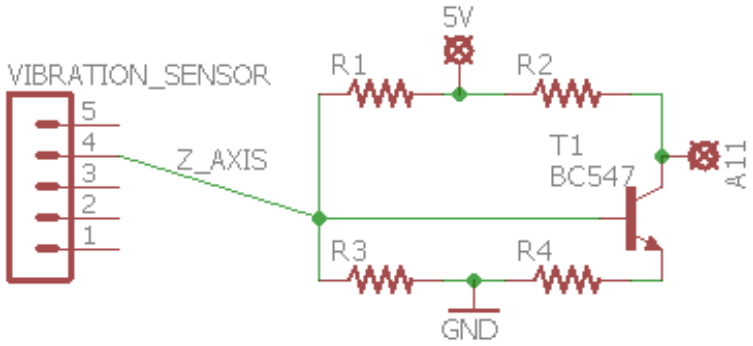

Figure 4.15: Circuit diagram for vibration sensor

4.15. The resistor value calculation is done step by step.

- The voltage over resistor R4 is chosen to be 20% of V+, which results in $5V \times 0.2 = 1V$.
- With a V_{CE} of $5V$ the transistor $BC547$ is specified, regarding the datasheet (Corporation, 2002), for a collector current (I_C) of $2mA$.
- Assuming that the transistor base current (I_B) is zero, the source voltage must be divided to gain a supply voltage of $2.5V$ for the analog to digital converter pin $A11$.
  By this specification the values of the resistors $R2$ and $R4$ can be calculated with Ohm's law
  $$R = \frac{U}{I}$$
  .
- R2:
  $$R_{Base} = \frac{V_{Resistor}}{I_{Base}} \Rightarrow R2 = \frac{5V - 1V}{0.002A} = 2000\Omega$$
- R4:
  $$R4 = \frac{1V}{I_c + I_b} = \frac{1V}{0.002A} = 500\Omega$$
- For the lowest amplification, the base current equals
  $$\frac{I_C}{B_{min}} = \frac{0.002}{200} = 10\mu A$$

- The current of the base voltage divider is chosen to be 10 times the base current

$$I_1 = I_3 = 10\mu A \times 10 = 100\mu A$$

By this specification the values of the resistors $R1$ and $R3$

- R3:

$$R3 = \frac{(1V + 0.7V)}{100\mu A} = 17k\Omega$$

- R1:

$$R1 = \frac{(5V - 1.7V)}{10\mu A + 100\mu A)} = 30k\Omega$$

- Finally the amplification can be calculated:

$$A = \frac{d(V_{adc}}{d(V_{z\_axis}} = \frac{R2}{R4} = \frac{2000}{500} = 4$$

To save space, the light measurement, the vibration measurement and the status LED have to be assembled on one board. The schematic of this combined multi-function board is displayed in Figure 4.16.
The connecting points $5V$, GND, $A10$, $A11$ and $D39$ are the connectors to the respective power, ground, analog and digital pins on the Arduino. The PHOTO_RESISTOR is the $10k\Omega$ photo-resistor with the $10k\Omega$ pull-up resistor prior.
The VIBRATION_SENSOR is the board with the soldered ADXL330KCPZ.
The DEBUG_LED is the red status LED, which is turned every time an Ethernet-connection is established with the Arduino.

4.2.6 Debug Hardware Testing Box

This final section describes for future developers how the hardware testing box can be debugged easily.
As first component the Arduino integrated development environment (IDE) (Arduino Contributors, 2017b) has to be installed. The installer also creates a library subfolder in the personal OS user folder which will be needed

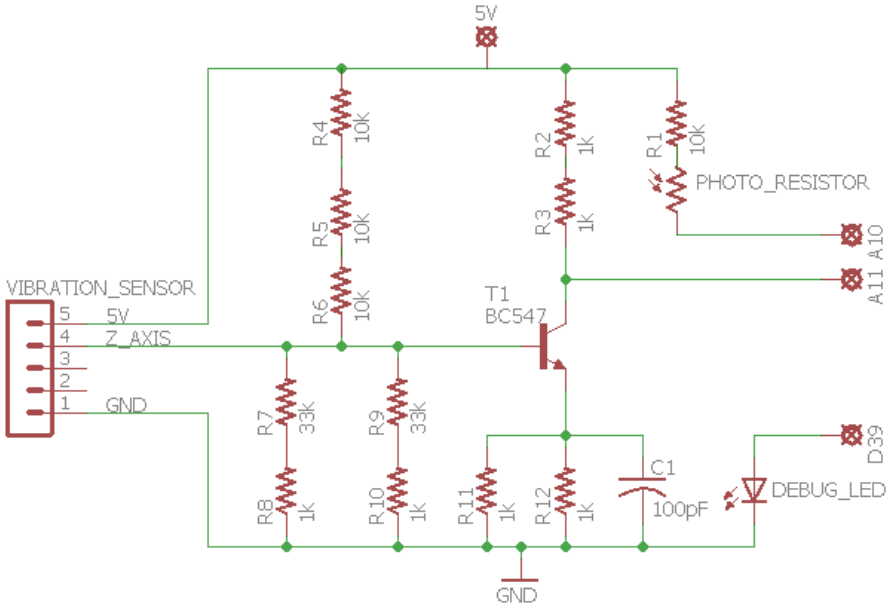

Figure 4.16: Circuit diagram for vibration and light sensor board

later on.
If the Arduino is connected via a USB type B cable with the computer the IDE can be started. The right port has to be chosen under the menu ”tools” and ”port”. After that the Serial Monitor can be started and the output of the Arduino gets already be printed in the monitor window.
The source code of the hardware testing box is available on GitHub[11]. To activate the debug output, which is deactivated on the running box for performance reasons, the ”sensorserver.ino” file needs to be compiled and uploaded to the Arduino with the uncommented code line:

```
#define DEBUG_SERIAL
```

Further a static IPv4 address has to be assigned appropriate the network subnet mask. This IP address has to be entered in the IPAddress object:

```
IPAddress server_ip_( 192, 168, 8, 9 );
```

[11] https://github.com/Catrobat/Arduino/HardwareTestingBox

To enable a successful compiling of the code additional libraries are necessary. These are located in the GitHub subfolder libraries. All subfolders in this libraries folder need to be copied to the Arduino libraries. As previous mentioned this folder can be found in the current user directory "Arduino/libraries/". Once they are copied, they must be imported to the IDE by selecting them in "Sketch→Include library→Contributed libraries". Now they can be used for compiling. After compiling and uploading this to the box it is ready for testing/debugging.
A faster way to send network requests, than executing Java test code, is to use a telnet client. For debugging during this thesis, the open source software PuTTY[12] has been used. When using such a tool, the connection type has to be set to "Telnet", the host IP address the IP address of the box and the port is originally set to 6789 on the box.
As soon as the connection is established the commands to check the phone can be sent. The listing with the possible commands is displayed in Table 4.3. The structure of NFC commands is explained in Table 4.4.
During debugging it can happen that the Ethernet Shield returns `0,0,0,0` as the actual IP address. This happens if it is not connected properly to the Arduino board. The best way to fix this is to plug the Ethernet Shield off and on.

[12] https://www.putty.org/

5 Conclusion and Outlook

This final chapter starts with a summary about Test-Driven Development (TDD) and why it is that important for the project. The subsequent sections describe the implemented features, which were finally merged into the project, followed by the lessons learned and finally an outlook on possible future work regarding this topic.

Testing is an essential factor for the success of this project. Through the high fluctuation of people it is a crucial advantage that the documentation of the source code is done via writing tests for every class and in the best case for every function. This does not erase all possible errors but the error detection rate during the development phase is higher than without testing.
In order to get the greatest success out of testing, TDD is the approach of choice in Catrobat for writing code. To test code in a meaningful way, it is important to design interfaces rather than starting with the beneath functional implementation. This allows to mock objects if relations are tested and enables testing with focus on single functions. Thereby failing tests show accurate which part of the code is flawed, since the dependencies are simulated and do not need to be debugged.
Through developing in a test-driven way the clean code standards are easier to meet, since the methods are kept small and clear. The continuous code refactoring eliminates code duplication and raises the code quality and readability.

For beginners it can be very time consuming and annoying to write tests for everything and before implementing the feature. The mindset of thinking in interfaces instead of functionality takes some time. Fortunately the available testing frameworks are easy to use and after a short work in the testing procedures become very intuitive. Espresso is very powerful therefore and multiple times faster than the previously used Robotium testing framework.

The tests are shortened in runtime and length, which has a huge positive impact for the developers and the continuous integration team.
For executing tests the Android emulator is a very powerful tool. It is possible to simulate some hardware intents and the performance of emulators is only little worse than the performance of real hardware devices. There is a wide range of devices that can be simulated for free, which enables to run the tests on multiple platforms and Android API levels. The only disadvantage appears when using the emulator because then the hardware functionality is restricted, since there is no possibility to work with NFC. For this reason a physical device is necessary to enable TDD for all hardware features.

5.1 Implemented Features

As a follow-up, an overview of the results of the implemented features is presented starting with the Near Field Communication (NFC) extensions for Pocket Code and followed by an evaluation of the hardware testing box.

5.1.1 NFC

The first of the three implemented NFC features in Pocket Code are the sensor variables. The `nfc_tag_id` device variable is useful to control conditions by checking the ID of the last read tag. It has a similar functionality to the `WhenNfc` brick when filtered on a specific tag ID. The advantage of this sensor variable however is, that a condition can also be checked regarding two or more specific tag IDs.
The `nfc_tag_message` device variable brings a completely new feature to Pocket Code, since this is the first possibility to work with the content of NFC tags. This device variable contains the content of the NFC Data Exchange Format (NDEF) record of the last read NFC tag. To display this content in the stage activity the `ShowVariable` brick can be used with this variable. Same as the `nfc_tag_id` device variable the `nfc_tag_message` can also be used for conditions to trigger content specific behaviour independent of the NFC tag ID.

The second new NFC feature is the `SetNfcTag` brick. This brick allows Pocket Code to actively write to NFC tags. The ID of an NFC tag is defined by the manufacturer, but the content of writeable tags can be set by the users. To enable Pocket Code users for this powerful capability, the `SetNfcTag` brick was added to the control brick panel. The NDEF supports multiple record types and the most common and useful eight types are available in the spinner of the brick. By using this functionality, users can easily exchange data through writeable NFC tags. This could be a website link, an email address, a phone number or a plain text. A special new available feature is the NDEF external type format.
Through adding a new `intent-filter` with `/catrobat.com:catroid` to the Pocket Code Android manifest file the application can be opened, respectively found, in the Google Play Store via an accordingly formatted NFC tag.

The third NFC feature, implemented for Pocket Code users, is the NFC tag Bluetooth pairing. This pairing was planned for an easier connection establishment between the smartphone and the Phiro robot. The setup for this part consisted of a Phiro, a writeable NFC tag sticker and a smartphone. With the use of a 3rd party application the Bluetooth connection details of the Phiro can be written to the NFC tag. To improve the pairing process which can be done by skipping the authorisation, the required password for the connection has to be stored in Pocket Code. As a result only the NFC tag needs contact to the phone once and the devices are paired.
This feature worked quite well, however it was not merged into the `develop` branch, because the use case disappeared since the Phiro manufacturers plan to switch from Bluetooth to a wireless local area network as connection technique.
Consequently this is the only NFC feature that is still not available in Pocket Code.

5.1.2 Hardware testing box

The Arduino hardware testing box is a really innovative tool for automatized smartphone application testing. Five hardware features can be tested with one compact box. It is only necessary to put the smartphone, the testing box

and the computer, which executes the tests into the same local area network. Developers who want to use the hardware testing box can either execute their tests on the Jenkins server, which is permanently connected to one box or they can configure the reserve box to their needs in their network.
This may be an obstacle for some developers but there is no easier way.
A little disadvantage of emulated devices is the manual maintenance of the phone. From time to time the smartphone freezes because running tests stresses the phone more than a normal usage. Combined with the continuous charging the durability of the phone is noticeably shortened. This is the price to pay for having an automated hardware testing tool.
The audio and light measurements work completely flawless, but the vibration sensor is slightly vulnerable to oscillation caused by anything other than the vibrating phone. Depending of the placement of other objects or persons or other things that effect vibration such as writing powerfully on a keyboard or being to close to the air conditioner, the oscillation can falsify the measurement, especially if the sensor is not at its designated position because someone moved the phone out of the mount.
The NFC shield is very powerful and supports various operations but also has its small flaws. The reason for that is that smartphones only react to new NFC tags in their electro magnetic field, which is meaningful for ordinary use cases since it is not necessary to read the same NFC tag several times when added only once. Each read tag is associated with an intent that should normally only be executed once. The problem with the used tag emulator is, that the smartphone does not recognize if the value of the tag changes, since its physical position stays unchanged. As a consequence the emulated NFC tags can only cause an intent every few minutes, when the NFC electro magnetic field of the phone is refreshed.
Having that in mind the execution of tests with emulated NFC tags can be timed to avoid this flaw.

5.2 Lessons Learned

The following section presents my personal experiences regarding the practical work of this thesis.
Before joining the Catrobat team I have never worked with Test-Driven

Development (TDD). So far testing was not that important for me and that is why test-first development changed my mindset. It took several coding sessions until I got used to this kind of work. In the beginning I often distrusted the test outcomes if they still failed after implementing the piece of the productive code. I often searched for possible errors in the tests, but usually the tests were right and the errors were in the new implemented functions. This showed me the advantages of TDD because it was quite easy to find the errors if only a few lines of code were added.
The largely well-named and structured tests were very helpful when I investigated into other features. It was also the first time I was forced to write with Clean Code methods. That pushed my code readability also for other projects noticeably.

In the course of this master's thesis I also had my first contact with Arduino. This is a powerful microcontroller with a wide area of application. Nevertheless, it is strongly limited regarding multithreading. Multithreading usually prevents busy states were external sensor commands can get lost. Maybe a Raspberry Pi could handle that in a more sophisticated way.

Also the used tools for the construction of the hardware testing box container were completely new to me. It was exciting to work with a computerized numerical control milling machine and also the 3D printer was an uncharted territory for me. The milling was extremely difficult with acrylic glass that I would prefer to use a laser cutter next time.

In the two years I worked at the Catrobat project, two Nexus 4 got broken. They were charged continuously what caused an expansion of the battery. Blessedly it did not caught fire. To prevent this behaviour in future phones we decided to root the used smartphones and install a battery charge control application. This restricts the battery charge to a maximum of 80% and stops charging until the battery discharges to 60% before charging again up to 80%.

5.3 Future Work

This thesis was planned to handle an isolated part of Pocket Code. The Near Field Communication (NFC) implementations support all relevant NFC features for children and teenagers which start with programming. Nonetheless it will be necessary to advertise these features at Pocket Code events and hand out NFC tags. Standard writeable NFC tags are fairly cheap, however most teenagers may not own writeable tags. It is also important to upload a few more programs with various NFC features to the Pocket Code game store. These games can serve as a guide how to use these NFC features.

The Arduino hardware testing box is a bit more predestined for future works. Beside the regular maintenance it is open for changes or extensions. A possible extension could be an automatized testing of the bluetooth connectivity. That would require a hack into the rooted smartphone, to be able to delete the existing Bluetooth pairing connections information before the new pairing test, but this seems to be an interesting topic.

A good case of application for the reserve box would be to use it for iOS hardware device testing. On the hardware side this would require some adaptations on the mounting parts and the light sensor position to fit the iPhone, but it would be possible with reasonable effort and could enable TDD for future iOS developers.

Bibliography

Beck, Kent (2000). *Extreme Programming Explained: Embrace Change*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc. ISBN: 0-201-61641-6 (cit. on pp. 9, 10, 13).

Community, GooglePlus (2017). *Espresso*. URL: https://google.github.io/android-testing-support-library/docs/espresso/ (cit. on pp. 16, 28).

Contributors, Analog Devices (2007). *ADXL330. Small, Low Power, 3-Axis ±3 g i MEMS® Accelerometer*. data-sheet. Analog Devices. URL: http://www.analog.com/media/en/technical-documentation/data-sheets/ADXL330.pdf (cit. on p. 70).

Contributors, Android (2016). *Testing UI for a Single App*. URL: https://developer.android.com/training/testing/ui-testing/espresso-testing.html (visited on 05/05/2017) (cit. on p. 16).

Contributors, Android (2017). *Intent*. URL: https://developer.android.com/reference/android/content/Intent.html (cit. on p. 67).

Contributors, Android (2018). *Run Apps on the Android Emulator*. URL: https://developer.android.com/studio/run/emulator.html (cit. on p. 33).

Contributors, Arduino (2017a). *Ethernet / Ethernet 2 library*. URL: https://www.arduino.cc/en/Reference/Ethernet (cit. on p. 65).

Contributors, Arduino (2017b). *What is Arduino?* URL: https://www.arduino.cc/en/Guide/Introduction (cit. on pp. 9, 18, 72).

Contributors, Arduino (2018). *Getting Started with the Arduino BT*. URL: https://www.arduino.cc/en/Guide/ArduinoBT (cit. on p. 43).

Contributors, Catrobat (2017). *Catrobat*. URL: https://www.catrobat.org/ (cit. on p. 9).

Contributors, Guru99 (2018). *Real Device Vs Emulator Testing: Ultimate Showdown*. URL: https://www.guru99.com/real-device-vs-emulator-testing-ultimate-showdown.html (cit. on p. 33).

Contributors, Jenkins (2017a). *Jenkins Documentation*. URL: https://jenkins.io/doc/ (cit. on pp. 9, 17).
Contributors, Jenkins (2017b). *Plugins Index*. URL: https://plugins.jenkins.io/ (cit. on p. 17).
Contributors, Robotium (2017). *Robotium. User scenario testing for Android*. URL: https://github.com/RobotiumTech/robotiumv (cit. on pp. 15, 28).
Contributors, Scanandmake (2017). *Arduino Accessible robotics*. URL: https://scanandmake.com/arduino/ (cit. on p. 18).
Contributors, Wikipedia (2017). *Jenkins (software)*. URL: https://en.wikipedia.org/wiki/Jenkins_(software) (cit. on p. 17).
Corporation, Fairchild Semiconductor (2002). *BC546/547/548/549/550*. (Visited on) (cit. on p. 71).
Coskun, Vedat, Kerem Ok, and Busra Ozdenizci (2013). *Professional NFC Application Development for Android*. 1st. Birmingham, UK, UK: Wrox Press Ltd. ISBN: 9781118380093 (cit. on pp. 19, 21, 47).
GmbH, DATACOM Buchverlag (2012). *ISO 14443*. Ed. by Klaus Lipinski. URL: http://www.itwissen.info/ISO-14443-ISO-14443.html (visited on 12/11/2017) (cit. on p. 23).
Igoe, Tom, Don Coleman, and Brian Jepson (2014). *Beginning NFC: Near Field Communication with Arduino, Android, and PhoneGap*. O'Reilly Media. ISBN: 9781449372064. URL: http://shop.oreilly.com/product/0636920021193.do (cit. on pp. 19, 25).
ISO/IEC (1997). *Identification cards - Contactless integrated circuit(s) cards - Proximity cards Part 1: Physical characteristics*. ISO/IEC 14443:– 1. International Organization for Standardization. URL: https://nfc-wisp.wikispaces.com/file/view/fcd-14443-1.pdf (cit. on p. 23).
ISO/IEC (1999). *Identification cards - Contactless integrated circuit(s) cards - Proximity cards Part 2: Radio frequency power and signal interface*. ISO/IEC 14443:– 2. International Organization for Standardization. URL: https://nfc-wisp.wikispaces.com/file/view/fcd-14443-2.pdf (cit. on p. 24).
ISO/IEC (2000). *Identification cards - Contactless integrated circuit(s) cards - Proximity cards Part 4: Transmission protocol*. ISO/IEC 14443:– 4. International Organization for Standardization. URL: https://nfc-wisp.wikispaces.com/file/view/fcd-14443-4.pdf (cit. on p. 25).
ISO/IEC (2001). *Identification cards - Contactless integrated circuit(s) cards - Proximity cards Part 3: Initialization and anticollision*. ISO/IEC 14443:–

3. International Organization for Standardization. URL: http://www.icedev.se/proxmark3/docs/ISO-14443-3.pdf (cit. on p. 24).

Kohsuke, Kawaguchi (2016). *Meet Jenkins*. Ed. by Larry Shatzer. URL: https://wiki.jenkins-ci.org/display/JENKINS/Meet+Jenkins (cit. on p. 17).

Krümmel, Nadja and Malte Ried (2011). *Kurzanleitung JUnit*. Technische Hochschule Mittelhessen, Campus Giessen, Institut für SoftwareArchitektur. URL: https://homepages.thm.de/~hg11260/mat/junit.pdf (visited on 05/05/2017) (cit. on pp. 13–15).

Lämsä, Tomi (2017). "Comparison of GUI testing tools for Android applications." MA thesis. University of Oulu, Department of Information Processing Science (cit. on pp. 28–32).

Langr, Jeff (2004). *Agile Java(TM): Crafting Code with Test-Driven Development*. Upper Saddle River, NJ, USA: Prentice Hall PTR. ISBN: 0131482394 (cit. on pp. 9–11).

Maple, Simon and Oleg Shelajev (2016). *Java Tools and Technologies Landscape Report 2016*. URL: https://zeroturnaround.com/rebellabs/java-tools-and-technologies-landscape-2016/ (cit. on p. 17).

Martin, Robert C. (2009). *Clean Code: A handbook of agile software craftsmanship*. Prentice Hall (cit. on pp. 10, 11).

Meszaros, Gerard (2007). *XUnit Test Patterns*. Refactoring Test Code. Addison-Wesley Professional (cit. on p. 13).

Nagappan, Nachiappan et al. (2008). "Realizing quality improvement through test driven development: results and experiences of four industrial teams." In: *Empirical Software Engineering* 13.3, pp. 289–302. ISSN: 1573-7616. DOI: 10.1007/s10664-008-9062-z. URL: https://doi.org/10.1007/s10664-008-9062-z (cit. on p. 11).

NXP (2007). *PN532 User Manual*. URL: http://www.nxp.com/documents/user_manual/141520.pdf (cit. on p. 25).

NXP (2012). *PN532/C1*. URL: www.nxp.com/documents/short_data_sheet/PN532_C1_SDS.pdf (cit. on pp. 25, 26).

Pulkit, Chouhan (2013). "Aspects of Test-Driven Development." MA thesis. Institute of Software Technology Graz University of Technology (cit. on p. 27).

Reichelt (2018). *ARDUINO HC-05-6*. URL: https://www.reichelt.de/www.reichelt.at/Entwicklerboard-Zubehoer/ARDUINO-HC-05-6/3/index.html?ACTION=3&GROUPID=8244&ARTICLE=170172 (cit. on p. 44).

Slavec, Marc (2016). "Integration of controlling Arduino boards via Bluetooth with Pocket Code for iOS using test-driven development." MA thesis. Institute of Software Technology Graz University of Technology (cit. on p. 28).

SurveyMonkey (2017). *NFC Shield V2.0*. URL: http://wiki.seeed.cc/NFC_Shield_V2.0/ (cit. on p. 67).

Vogel, Lars (2016a). *Android user interface testing with Espresso*. URL: http://www.vogella.com/tutorials/AndroidTestingEspresso/article.html (visited on 05/05/2017) (cit. on p. 16).

Vogel, Lars (2016b). *Android user interface testing with Robotium*. URL: http://www.vogella.com/tutorials/Robotium/article.html (cit. on p. 15).