Marc Slavec, BSc

Integration of controlling Arduino boards via Bluetooth with Pocket Code for iOS using test-driven development

MASTER'S THESIS

to achieve the university degree of

Master of Science

Masters degree programme: Telematik

submitted to

Graz University of Technology

Supervisor

Univ.-Prof. Dipl-Ing. Dr.techn. Wolfgang Slany

Institute for Softwaretechnology

Graz, April 2016

AFFIDAVIT

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 indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis dissertation.

Date

Signature

Acknowledgments

I like to thank everybody, who supported me during my study.

First and foremost, I want to thank my parents Anton and Michaela for their support.

I thank all my friends for everything what they have done for me and that I could always depend on them.

Finally, I want to thank Professor Wolfgang Slany and the whole Catrobat team.

Abstract

Catrobat is a visual programming language used on mobile devices to introduce programming to children in a playful manner. Since the area of Internet of Things devices is rapidly growing, Catrobat and the Pocket Code application, which is a mobile IDE for the Catrobat language, have to integrate features allowing to control these devices wirelessly to be able to tackle these new developments. This thesis deals with implementing a wireless Bluetooth connection for iOS devices running Pocket Code with Bluetooth enabled devices, in particular Arduino boards. The implementation will be complete, but the design of the Bluetooth connection is very open to integrate new BLE enabled devices to Pocket Code without much effort. Every implementation needs to be tested, that is why the used developing workflow will strictly follow the Test-Driven Development principles, aiming for a well tested and designed code as well as an assurance for not breaking these code parts, because the tests will immediately show the defects. The general topic of testing, especially ”Unit Testing” will also be covered. Furthermore, since it is the used workflow, ”Test-Driven Development” will be discussed, compared with other workflows and analysed.

Kurzfassung

Catrobat ist eine visuelle Programmiersprache, die dazu verwendet wird um Kindern den Einstieg in die Welt des Programmierens spielerisch zu erleichtern. Die Sprache wird in Pocket Code, einer Applikation auf mobilen Geräten, eingesetzt und visualisiert durch das verschieben von Blöcken. Da heutzutage Geräte im Bereich "Internet of Things" immer stärker vertreten sind, wurde bei Pocket Code eine neue Eigenschaft hinzugefügt. Diese beinhaltet das kabellose steuern von anderen Geräten mit Hilfe von Bluetooth. Diese Arbeit handelt von der Implementierung der Bluetooth Kommunikation für Pocket Code auf iOS Geräten. Das Steuern von Arduino Geräten wurde im Zuge dieser Arbeit vollständig implementiert, jedoch durch das Design der Kommunikation ist es sehr einfach weitere neue Geräte zu integrieren. Jede Implementierung gehört getestet, deswegen wurde beim Workflow strikt die Prinzipien der testgetriebenen Entwicklung, welche sich auf gut getesteten und gut durchdachten Code beruht, eingesetzt. Diese dazu angelegten Tests dienen auch der fortlaufenden Wartbarkeit des Programms, da sie sofort auf neu entstandene Fehler hinweisen. Weiters wird der Überbegriff Testen, im Speziellen "Unit Testing" behandelt. Außerdem wird die testgetriebene Entwicklung, da diese im praktischen Teil benutzt wurde, erklärt, mit anderen Methoden verglichen und analysiert.

Contents

List of Figures

1. Introduction

Catrobat, a visual programming language inspired by Scratch, has its target to help children come closer to the topic of programming. But programming needs special syntax to write. Instead of this, visual blocks are arranged to create the desired behaviour. This programming happens on mobile devices, which are nowadays widely spread in the target group. Over the time, Pocket Code, the mobile application, which allows to create and design programs, has added a lot of features, like face detection or the use of multiple mobile device sensor values to create more sophisticated and exciting programs.

To be up to date with the current growth of smart devices and the huge area of the Internet of Things, Pocket Code has to be further extended. In the practical part, a Bluetooth connection is introduced for the iOS version. The created Bluetooth framework has a very open design to add different kinds of Bluetooth devices to the application easily. Therefore, the application is ready to control several distributed devices or for example to steer different kinds of robots via Bluetooth Low Energy.

The first device, which is added to Pocket Code during this thesis, is Arduino. Due to the open source micro controllers and a huge fanbase, it has a wide range of usage possibilities. It could be the core of a robot, a central unit of diverse sensors or anything else the user connects to the provided pins. Arduino is commonly used for fast prototyping or easily creating different kinds of hardware functionality.

With visual programming and easy hardware prototyping, the combination of Pocket Code and Arduino has a very high learning effect for children in the field of software, hardware and the combination of both. The tight interconnection between Software and Hardware will gain importance increasingly. Especially for kids, who are interested in electronics or informatics, this would be a great starting point.

1. Introduction

The most important part of software development is to deliver the right software, which has the functionality as expected. Unsuccessful software with error prone code will lead to unsatisfied users and therefore abortive projects or companies. Over the years, the demand of good tested software has not dropped. Accordingly, over the past years different approaches to the

Part I.

Theoretical Part

2. Catrobat

2.1. A Visual Programming Language

Catrobat [1] is a visual programming language to comfortably learn to code. It is built on Scratch [2][3] that was introduced the Lifelong Kindergarten Group (LLK) at the MIT Media Lab [4]. Programming is achieved by simply choosing so-called blocks and putting those one beneath each other. Therefore children only have contact with the blocks, perhaps familiar with playing Lego and do not think about the coding part. It is playing with these blocks and creating something new, which is in this case a new program, game or interactive story. This type of programming is less complicated than writing lines of code, because there is no worrying about a complicated programming language or syntax errors.

Inspired by this block type programming, Catrobat is also using this way to quickly and easily create custom programs. Instead of using the personal computer or laptop, like Scratch does, Catrobat is using its own mobile applications called *Pocket Code* [5] to create and execute Catrobat programs on mobile devices. Due to this fact, more people are reached, because smartphones and tablets are more widespread than laptops or personal computers nowadays especially when talking about children. In that manner, children carry their *learn-to-code-tool* always with them in their pocket.

The integrated development environment (IDE) and the interpreter are grouped in a mobile application called Pocket Code. This implementation is natively available on *Android*, *iOS* and *WindowsPhone*. Furthermore, there is a *HTML5* version to run Pocket Code in a standard browser. All versions behave the same in the core Catrobat language. Catrobat is using *bricks* and *scripts* as blocks similar to Scratch. Scripts contain bricks and run from top to bottom but different scripts can be executed concurrently.

2.2. History

Catrobat and it's core *Catroid* team were introduced at the University of Technology Graz. The first goal was to port Scratch to Android, therefore the name Catroid was given, which is a fusion of Catrobat and Android. At the initial start only few people were working on this project, but after some time and advancement of the team, it became popular at the university. More and more students joined the different teams. At this point over 100 people at the university of technology of Graz and external people are collaborating on the different Catrobat projects. With this influx of young programmers, a bunch of sub-teams were established and features were added progressively. In 2012, the iOS team was formed and is since then working on the iOS version of Pocket Code.

The potential of the project was rapidly identified and therefore Catrobat was awarded in the year 2013 in the category Multimedia with the *Austrian National Innovation Award*. However, the reputation was not only national. Google chose Catrobat a few times to be part of their "Google Summer of Code"[1] program, which was a huge success for Catrobat. Recently the team was also granted the *Lovie Award*[2].

2.3. Subprojects

As mentioned before, a lot of sub-teams were put in place to break down complexity and to work in a smaller team better together. Here is a short list, which teams the Catrobat project exists of:

- Catroid - Android core team
- Chromecast
- Drone
- Lego Robot
- Live Wallpaper
- Musicdroid
- NFC
- Phiro

[1] **https://developers.google.com/open-source/gsoc/, visited:22.03.2016.**.
[2] **http://lovieawards.eu, visited:22.03.2016**.

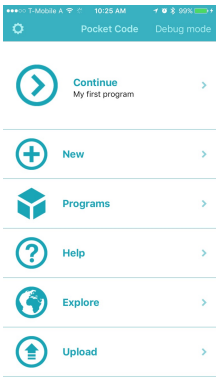

Figure 2.1.: Pocket Code for iOS start screen

- Physics Engine
- Rasperino
- Catroweb
- Design
- Education
- HTML5
- iOS
- Paintroid
- Usability
- Windows Phone

In the further thesis the iOS version of Pocket Code is considered because new bricks will be introduced, which are not supported in all of the other versions. Currently the iOS version of Pocket Code is in beta-phase and not available on the iOS AppStore. The current design is shown in Figures 2.1, 2.2, 2.3.

2.4. Team Collaboration

Since so many different teams exist, one can think that each team works encapsulated on its own. Nevertheless, this is the other way round. There is

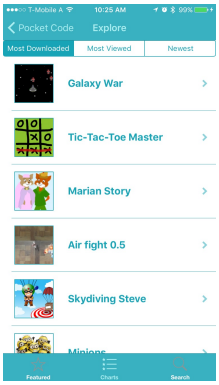

Figure 2.2.: Pocket Code for iOS download projects of other users

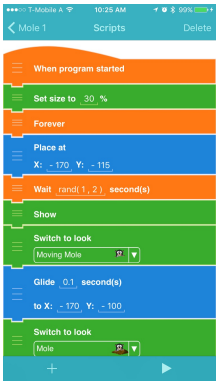

Figure 2.3.: Pocket Code for iOS script editor to create and remix projects

an interconnected network between the teams. Coordinators of each team gather questions and examine these with the other coordinators. Questions in the chatroom can lead to prompt answers, because all Catrobat members are reached.

There are not only real sub-teams, for example the usability and the design team work across all platforms. Accordingly, they are highly embedded in the development process of every programming team.

This fact is really relevant for a programmer, because he/she only has to concentrate on programming and not designing or spending too much time thinking about the detailed usability. In my opinion, this is a good approach for the workflow, so that everyone can focus on one task and do not have to "reinvent the wheel".

2.5. Internationalization

As highlighted previously, the team members are not entirely located in Graz, since the project is very international and open minded to new contributors. It is an open source project, where everybody is able to check out the code and commit their changes and additions as long as they follow the preferred workflow containing test driven development that will be later described in detail.

Programming is mostly done in English as one can see in the main programming languages like Java, Swift and others. On account of wanting to reach as many children to learn to code in a playing manner, it was a concern from the ground up to support as many languages as possible. Because learning a language for Pocket Code would distract from the core concept. Therefore it should be possible that every child uses the application in their native language, so that no linguistic barriers exist when it comes to complicated patterns. Not only the language should be familiar to the children, but the application should match the cultural definitions like left-to-right or right-to-left reading. Another point would be the mathematical operators, which differ in some countries. We, at Catrobat, want to reach as many children as possible, so this mentioned and a lot of other differences are a big adjoin for us. For reaching that goal, we work together with native speakers of diverse countries.

3. Software Testing with Focus on Test-Driven Development

3.1. About Software Testing

Written software without tests can be unreliable and there is a high chance that the program behaves differently under certain circumstances. Since the beginning of programming in the 1950's, testing the written lines of code was part of the development workflow to deliver programs of appropriate quality. Of course software testing changed over the history, but it was always an essential part of a development workflow. According to Myers et al. [6]: "Testing is the process of executing a program with the intent of finding errors."

Myers [6] stated that these days, when hardware and machine cycles were very expensive, double checking the written code to ensure that it will work afterwards was part of the testing process. This method, called desk checking, was part of the programmers everyday work. No thoughts were given letting the machine do the job, because the computing power was too slow and the costs tremendously high. The computer should not be utilized to test the program, but to solve the given problem running the program. This fact changed over the years, because machine cycles became very inexpensive and it was not difficult to run the code and see the outcome. Unfortunately, a good discipline was lost. The desk checking practice fundamentally has advantages, but it just needs too long to work through it for today's time schedule. A combination of both could be one solution, but other more sophisticated models were developed to tackle this core problem. For example, writing tests using the computer as a resource and execute them as often as possible to check correctness [6].

3.1.1. Why Test?

But why should software be tested? Of course, there are different types of software. To write software, which is used for security or safety reasons, is very critical for programmers, because it has to be very reliable. More precisely, it has to be guaranteed to work and thus it should be sufficiently tested. As Graham [7] mentioned, concerning other types of software, it is mostly just about gaining money, without that very critical part. That's why testing the product is a kind of increasing the chance to satisfy the customers so that they will buy the program or application. Testing is therefore helping the company to reach the goal of making profit. There is a tradeoff between testing and profit, because if testing in some level of detail is too expensive and do not improve the quality significantly, then the company should stop testing that part and go on without whole test coverage. However, this can only be done if that part is not that important. This fact is based on testing costs that can excel the profit afterwards. Due to this case the best solution is to find a balance between creating tests to control the development process and checking the program to provide confidence that it will work [7].

The interpretation of software testing divides the developers opinions into two parts. Of course, it is always about testing the code to see if it is working as expected. On the one hand, some programmers see it as kind of quality insurance to prove that the program will work afterwards. On the other hand software testing is seen as finding bugs to be able to fix them, but the quality measurement is not in first place. It is logical if one bug is solved that the quality will rise. However, nobody can know how many additional faults are in that certain lines of code. Obviously, it sounds the same, but there is a small difference. As mentioned by Meyer [8], "Testing a program to assess its quality is, in theory, akin to sticking pins into a doll—very small pins, very large doll.". He also mentions in his paper that there are infinite data inputs and a lot of logical ways (even in a tiny, easy program) that testing is not able to cover everything. According to him,

> "A successful test is only relevant to quality assessment if it previously failed; then it shows the removal of a failure and usually of a fault." [8]

Having this in mind compared with the resources and time to market, it is not feasible to write completely error free code. However, a certain

amount of well thought out tests can provide confidence that it will work most of the time. Certainly the focus should be set to critical components first and moved on to lower ones for the sake of confidence. [7]

3.1.2. Testing Models

According to Graham [7], when a controversial subject arises, different models and solution approaches are composed. This also applies in the case of software testing. In the early years, as mentioned before, checking the code several times by hand or giving the lines of code to another people to double check them were the common types of testing. However, that routine was lost and drifted to a "Code and Fix Model". This model and the "Waterfall Model" will be covered in this section, because they are even today widely used workflows.

Code And Fix Model

As the name of this model reveals, the developer writes lines of code, runs the code and if it is not working as expected, then the written code lines are edited. This process is iteratively done till the feature is implemented. The model is based on the previously mentioned fact that testing software on hardware is nowadays very cheap and fast. That is why, programmers lost the habit of checking the code to a trial and error system. They are trying to compile the code and if it does not compile, errors and warnings will assist them to fix the problem. If it compiles, then the simulator/emulator is started and the developer checks if it is working as expected or not. However, the fundamental core of testing is completely set aside with this approach [7].

Waterfall Model

Another approach would be the Waterfall Model. Managers create product requirements and specifications, which are then implemented by developers. According to Graham[7] each part of the process is handled as a complete phase with an end product to hand over to the next phase. After developers have written the code, an individual testing team (sometimes called "testing specialists") is testing the output. This workflow is shown in figure 3.1 [7].

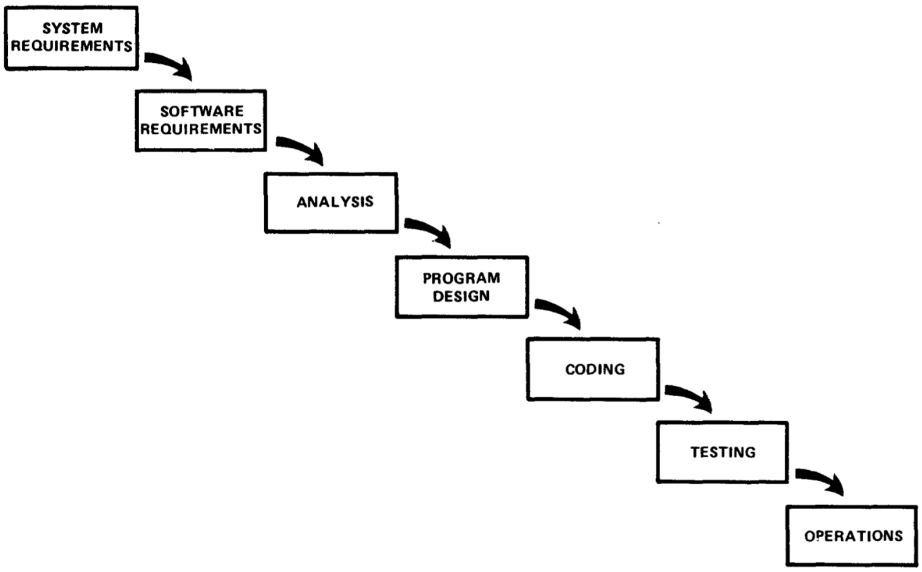

Figure 3.1.: Implementation steps of the waterfall model.[9]

More people are involved in the developing process and there is a higher chance of finding more "bugs". Another positive aspect is that since more people are elaborating, it is more likely that the given specification is fulfilled. Sometimes programmers misunderstand some ambiguous specifications or it is difficult to implement the requirements as the customer has required it [7].

When developers are testing their previously written code, they have complete insight and knowledge about the concept of the program flow. Accordingly, there is a high chance that they are using the product in a different way than ordinary users would do. In this regard, the method outperforms others, but it also has some negative flaws.

The first and biggest disadvantage would be that the testing happens at the end of the "waterfall" process. For example, if a tester finds a bug and reports it to the developers, they have to search for and understand the error. Perhaps it was a long time ago that this part of the code has been written, then the problem has to be debugged. Some debugging sessions are very time consuming and therefore very expensive.

Another negative aspect is that misunderstood equivocal requirements are not discovered till the end. This can be expected to result in a complete redesign of the software.

These two models are not the only ones, but latter were used very often in the history of software development. New methods have been developed, for example *agile development* with its test driven approach, which will be covered in section 3.3.

3.1.3. Who Should Test When?

As discussed previously, the earlier bugs are found, the cheaper it is to fix them. Imagine, software is checked at the end, as it happens in the waterfall model, and then a tester reports that a requirement has been misunderstood. It is probable that everything concerning this requirement has to be reimplemented, which is very time consuming and therefore leads to a rise of the developing costs. So testing and coding have to go side by side.

Graham [7] says that, "software should be tested all the time" and in the best case all parts of it, which is not possible, "within reasonable time and budget constraints" [7].

Modern concepts, like *test driven development*(Section 3.3), indicate that it is important to test everything at any given time. This is called "agile" development that is managed in short iterations. So after every brief repetition, the requirements and specifications are reviewed and checked if the code needs changes or one can continue with the next part. Therefore testing is done continuously in projects that are following the agile concept. Without regard to the fact that software was traditionally managed with testing at the end of the development workflow with beta testers, this type of testing is just one part of the whole testing process nowadays [7].

As mentioned before it is nice to have different people developing and testing to gain a higher chance of finding errors in code. These testers can be specialised for testing reasons to find more errors in a shorter time period. For internal testing of code parts it is useful to let experienced developers write the tests instead of starting developers, because writing tests is not comparable with trivial tasks. Generally said it is virtually impossible to

cover all possible ways a user will use the product. According to that fact, a chance of a user finding an error is always alive [7].

3.1.4. Different Testing Types

As reported by Myers et al. [6] two main different testing strategies have been proven. The latters are now covered in detail:

Black Box Testing

The first strategy is *black box testing*, also known as data driven testing. In this strategy, one part is seen as a black box and the test is checking if there is the correct output for a given input. Accordingly, the goal is to find input data where this "black box" is not showing the desired behaviour. With this method, one should bear in mind that *every* input is a possible test case, which is an infinite number of test cases, because all "possible" inputs have to be tested and not only all valid ones. A user sometimes does not know anything about the valid inputs. Myers et al. [6] named it "exhaustive testing" and mentioned that is not feasible to cover every input. Hence one has to maximize found errors with a limited number of tests [6].

White Box Testing

White box testing (logic-driven testing) has a different approach than black box testing. This testing type allows to see the structured code inside the box. So one can see it as a glass box. Each control flow path in the box should be considered in one test case. This will lead to exhaustive path testing, because the number of logical paths in a common program is indeed not infinite, but very high. The same principle holds that one should maximise found errors with a limited number of tests as described in black box testing [6].

Though, there might be another source of failures that are not covered by white box testing. These white box tests should cover "all" implemented logical paths, but are all paths implemented or are some paths missing in the implementation? These not implemented tests will remain untested and ergo a source of possible failures [6].

More detailed subdivision

White and black box testing are kind of concepts and they are further divided into special testing techniques:

`White Box Testing:`

- **Unit testing**: Each unit (for example class) should behave as expected. It is mostly done by the developer to check every part of an application programming interface, if it is working as expected. Unit tests, used in test driven development, will be covered in Section 3.2.
- **Integration Testing**: More Units/groups together should produce a desired output.

`Black Box Testing:`

- **System Testing**: The full system is tested in different environments.
- **Stress Testing**: As the name reveals the program is tested under stress conditions.
- **Usability Testing**: The program is tested from usability experts to see how easy and comfortable it is to use by the end user.
- **Acceptance Testing**: The system is tested if it meets all the given requirements.
- **Regression Testing**: If there is an update of the program, regression tests will check if all features implemented before are still working.
- **Beta Testing**: It is done by end users before the software is released. It should be the last testing before the release and only special cases of errors should be found.
- **Performance Testing**: Testing how good the software performs.
- **Penetration Testing**: Checking if there are security problems.

With this listing, one can see that testing is no longer a trivial subject and not only about checking lines of code. Several people are involved in *completely* testing a software in different ways. There are even special teams, which are only responsible for usability testing. The importance of testing can obviously be seen in these examples [6] [7].

3.1.5. Principles Of Software Testing

Meyer [8] evolved seven principles of software testing, which will be described in this section. These principles are kind of rules that one should have in mind when developing software and implementing test for it.

- "To test a program is to try to make it fail"[8]
- Tests cannot substitute specifications
- Testing has be an automatic process
- Test success and failure should be detected automatically
- Test cases should be created manually and automatically
- Every testing strategy should be considered
- The number of faults is crucial for a testing strategy

3.1.6. Conclusion

Dijkstra once said that, "Program testing can be used to show the presence of bugs, but never to show their absence!" (as cited in [8]). Hence, the intention should be to find bugs to limit errors when testing. Ergo it should not act as a proof for the complete correctness of the program.

A good way of testing is not done by the developers themselves, because they know too much about their own implementation. Furthermore, test cases are interesting and successful if they find an error. Several testing strategies should be considered to raise confidence about the product. However, a balanced tradeoff between testing effort and profit should be kept in mind. Overall testing is very important to deliver good quality products and accordingly to be successful.

3.2. Unit Testing and its applied techniques

3.2.1. An Introduction To Unit Testing

Nearly all developers agree on that testing can be really helpful during the program's evolvement and afterwards if some parts of the code have to be rewritten. However, some of them do not test, even if it could raise the software quality, because it needs kind of overcoming to start testing. According to Osherove [10] this is the fact, because lots of tests are poorly

written and then not useful as mentioned in the last section. This will lead to an unmotivated attitude to testing.

Motivation

Koskela mentioned in his work on *Effective Unit Testing* [11] that writing tests is a kind of stopping the developing phase and do not come closer to the goal in the thoughts of programmers. Of course tests will not add more functionality to the program, but there are plenty of reasons why testing should be done.

Unit testing can help to forget about checking the same code lines several times. Because if tests exist which are checking the correctness of this part, then just running the tests is easier than checking each code line for several times. This will improve the overall speed of the development [11].

Another advantage of writing unit tests is that it will help to improve the whole design and will raise the understanding of the content of this unit or how it should be organized. Where does the design of the code fit in this matter? - A poorly code base is very difficult to test, because of dependencies and methods, which are implementing too much functionality. Testing will help to encapsulate classes and split methods to have one core functionality and therefore increase the quality of the design [11].

If the code coverage is very high, then the testing suite turns into a kind of living documentation. The first thing one will do to understand an application programming interface (API) is to check some examples, how to use it properly. These written tests will act as those examples and are always up-to-date [11].

Projects with an integrated well written test suite are more likely to be resistant to regression errors. These errors arise, if some parts are rewritten or refactored (see Subsection 3.2.2) in the code and then a functionality is not working, which was working before. After each change, only the tests have to be executed to see if everything is still working as expected. This will help to speed up the whole refactoring process and improves the workflow of more people working together on a single project. Therefore, every team member can refactor, change parts of the code or add new functionality without thinking about breaking some working service because the tests will show. The whole confidence about the code is raised without double checking and accordingly losing valuable time [11].

Overall, appropriate testing will require some implementation skills for tests and therefore some training. It will also take some time to implement those tests, but afterwards it will help tremendously and will meet the effort [11].

Definition

The term *Unit testing* has to be split in two parts to reach a base for understanding the definition. *Testing* is kind of checking if something is working as desired. A *unit* has no exact definition in software development, because it can have more possible meanings. In object oriented programming, when using the term unit it means a class most of the time. But globally seen unit is nothing else than a piece (a part) of code. Osherove [10] gives a global definition of unit testing,

> "A unit test is an automated piece of code that invokes the unit of work being tested, and then checks some assumptions about a single end result of that unit. A unit test is almost always written using a unit testing framework. It can be written easily and runs quickly. It's trustworthy, readable, and maintainable. It's consistent in its results as long as production code hasn't changed." [10]

This is a quite compact definition and ergo complete understanding cannot easily be reached without knowledge about unit testing in detail. A unit test will run automatically and checks if this unit (this part of the code) is executing the expected functionality. Unit testing frameworks are in place to help writing unit tests. These frameworks contain functionality to assert code parts that are not working as expected. The implementation of those tests should be straightforward, if the design is satisfactory thought through. A very important fact is that the tests should be consistent, which means without changing the code, there needs to be the same result for each test run [10].

Another question arises. What should be tested? - According to Osherove [10] every function/method which has a logic in it should be considered as a potential error source. If a function includes a control flow instruction element, for example an *if* statement, *for* loop, or any other type of conditional structure, then it should be considered.

Principles

In the definition, it is written that a unit test should test exactly one single code part that is completely encapsulated and does not invoke other logic. A test poses an assumption that needs to be fulfilled. If the assumption is valid, then the test passes, if not, the test has failed, so one can conclude that the code does not implement the desired functionality. However, this is not an instruction for writing effective unit tests, just how they are defined [10].

> If you're going to write a unit test badly without realizing it, you may as well not write it at all and save yourself the trouble it will cause down the road with maintainability and time schedules [10].

With this statement Osherove [10] describes that badly designed tests will not help to improve your program, but to cause more troubles. Good unit tests have to be written, because otherwise there is no need for them to be implemented. To succeed in writing useful tests it is necessary to follow a few principles. The tests should

- be automated and repeatable
- easy to implement
- be long-lasting
- run quickly
- be executable for anyone
- be consistent
- be fully isolated
- be easy to debug

In the following paragraph these principle definitions are described in detail according to Osherove [10]. The computer should do the testing for us, so the tests need to contain all prerequisites to run for example on a server without monitoring. If a unit test is difficult to implement then something is wrong with the design. With long-lasting, it is meant that a test should be for instance also runnable in a year. That is why it has to be independent of other services, where a possibility exists that this service is not provided anymore. This could be for example a network service. Since

these tests are only checking minor parts, the execution time is forced to be low. Everyone, who wants to run the tests, has to be capable to do it. Therefore, environmental dependencies have to be avoided as far as possible. As mentioned in the previous section the consistency of the tests needs to be given in every single test run. It is anticipated that there is no difference in the result of the tests if they are rearranged. This means that each test should be fully isolated and run independently of others. If a test fails, it ought to be clear what was expected and how to change it that the assumption is fulfilled in a reasonable time [10].

Osherove [10] states that if these principles are integrated in the testing development process, then there is a high chance to write useful unit tests. If not, unit tests can cause more trouble than benefit. Furthermore, a naming convention and a structured hierarchy for unit tests should be taken into account for easier maintenance of the test suite, like practiced with the working code directories. If the maintenance for the tests is already too difficult, then the tests are falling by the wayside and therefore the same problem arises as before writing tests. There should not be too much maintenance for unit tests, but since tests are also kind of *living documentation* they should have a logical order [10].

3.2.2. Refactoring

> Refactoring is the act of changing code without changing the code's functionality. That is, it does exactly the same job as it did before. No more and no less. It just looks different. A refactoring example might be renaming a method and breaking a long method into several smaller methods.[10]

This statement by Osherove [10] completely describes what the content of *refactoring*. Rewriting the code with the same functionality afterwards, but improving the design. Without unit tests, refactoring can be a very hard task, because one cannot be sure about everything is working after rewriting parts without tests to show the outcome. Unit tests will help to be confident that everything is still working [10].

Accordingly the goal of refactoring is to make the code cleaner and easier to read. To achieve the goal, no recipe exists to follow, but it is a good

start to follow commonly used patterns. Refactoring is also a part of test-driven development. The way it is used in test driven development is to do refactoring in every iteration to end up with clean code without a big refactoring in the end [10].

Refactoring in terms of testing can be seen to open up private classes to make them testable. These encapsulated classes have private properties which are not required by the external components. For testing reasons some of these properties have to be accessed and that's why the design has to be refactored to be fully testable. Sometimes it is claimed that this opening is a wrong approach, but according to Osherove [10] this is an acceptable way, if it is used for testing reasons only.

3.2.3. Dependency Injection

Dependencies in code parts, which should be tested, are the biggest barrier to implement effective unit tests. To have a proper testable class it is necessary that it is independent of other classes or services as far as possible. To achieve independence, a pattern called *dependency injection* is applied. There are several techniques for removing a dependency by injecting something in its place [12].

The first and probably the starting approach for decoupling classes is **Constructor Injection**. As the name already reveals, a dependency is injected in the constructor of a class. This fact gives the programmer the ability to inject a test example to test the class [12].

Another technique is **Property Injection**, where the dependency is stored in a class-own property and the injection takes place by setting the property accordingly. Of course, this can only be conducted, if the property is public accessible [12].

If the dependency is only present in one function then **Method Injection** would the right solution. For applying this method, the function call holds the dependency and if it is called, the dependency gets also injected.[12]

These three techniques are commonly used to solve the dependency testing issue. Which one of these approached is used builds upon how often the dependency is used and where it is used. It needs some experience to notice, which one is used in a certain case. Dependency injection is a core part to succeed in writing unit tests. To avoid it, programming with

interfaces of each class can be very useful. If the dependency is another class, then it is not trivial to encapsulate for testing reasons. This problem is tackled in the next section with so-called *double patterns* (Section 3.2.4) [12].

3.2.4. Double Patterns

In this section covers double patterns that are helping to get rid of object dependencies, where no control is given. This object can be of any kind, for example a class object, a web service or as later showed in the practical part a bluetooth connection that is connecting a device with the program. But there are many other cases, where double patterns are demanded. The only way to say whether the logic works or not, is to force complete independence for the unit under test [13].

Globally seen, the dependency is removed by injecting some fake code that is replacing the actual code in this dependency object. First a few definitions are required to distinguish between the different double types [13].

Mocks, Stubs And Fakes

In a recent article, Lazer Walker [13] defined and compared these different test double types.

The name **double** was already mentioned. This is a kind of general term for all "edited" objects.

A **stub** returns a specific fake value when a method is called. One can define this fake value before starting the test and see if the test is working as expected with the given value. For example, one can fake a communication with a web server using these stubs and just pass back a static server answer.

A **mock** is very similar to a stub, but a mock can be asserted as well. This means, one can verify if a method of this mock is called or if it is called with the right argument. Another difference would be that mocks can have some expectations beforehand, where the mock "knows" what will or should happen and verify if the correct behaviour is discovered. The test will fail, if the method is not called or called with the wrong argument. Mocks are often used for interaction testing, because these tests can consist of functions with no return value. So you need a mock to verify the result.

Finally, there are **fakes**. Fakes behave the same as real objects of that class, with one exception. For example if a class is a data storage connected to a database, then a fake would remove the database connection and have some test data entries in-memory.

All of them are used to simplify testing by replacing dependencies, but each one with other features. Additionally these three terms are often mixed up and otherwise used. These three approaches can be combined, but one should bear in mind that a unit test should only test one specific responsibility. So having two mock objects would not fulfil the unit test definition. More stubs would be ok, because they can't be asserted. Sometimes identifying what actually should be tested is not that easy and one needs some experience in it [13].

How to use doubles?

Osherove [10] presents in his work different ways how to use double patterns in tests. The first approach is to set up all classes with interfaces (in iOS interfaces are called protocols) and then double objects can implement those interfaces easily. In Swift, it is suggested to program with protocols. If this is done from the beginning, then testing will be very straightforward. If no interface exists or before rewriting all of the code one should use a framework. There are a lot of frameworks for that particular reason, which are implementing these double patterns, so that one does not have to do it for each class. The only thing to do is include the framework and it does the work of decoupling the classes by creating mocks. These frameworks are often called *Mocking Frameworks*, but actually most of them implement all of the mentioned doubles. The latter greatly facilitate the process of creating and injecting those double objects, because it is only one function call. But there is one reason not to use it, since they are helping to test there is a high possibility that the developers do not give attention to the design of the code [10].

3.2.5. Continuous Integration

Osherove [10] also mentions Continuous integration (CI) in his is guide about effective unit testing. CI not only for unit testing, it consists of an automated build and integration process. Often the continuous integration

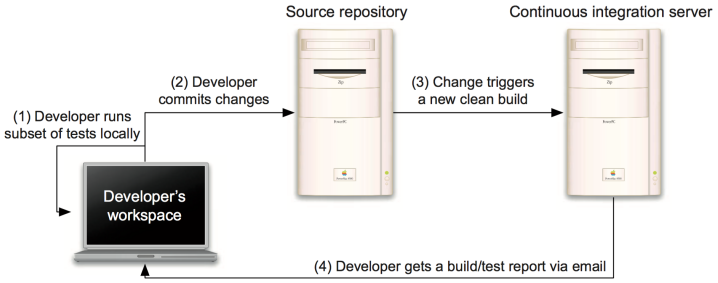

Figure 3.2.: Basic setup for continuous integration with the help of a server[14]

process is misunderstood and equated with continuous integration servers. A server is not necessarily needed to use the CI pattern. These servers only provide the functionality and facilitate the use, because some CI parts are then done automatically [10].

However, according to Koskela [14] using continuous integration in a team can be very useful. All team members are constantly changing the code and pushing those changes to the source repository. This frequent integration of the developers' work and the source repository is called continuous integration. So far, the testing topic was not mentioned. The code in the source repository is not only meant to integrate and build, but it should also work as expected, which will be covered by the tests. The integrated source code is verified by running automated unit tests (certainly integration and user interface tests will also be executed) [14].

Koskela [14] also mentions that the straightforward approach would be that developers only test a specific code part themselves and delegate the testing of the entire test suite to the CI servers. Due to the fact that testing the whole test suite will take too long and the very likeliness that most of the tests are not affected, this is a proper way of handling it. In Figure 3.2 a setup for this approach is illustrated [14].

Osherove [10] states that the servers main job consists of triggering the building and executing the tests on specific events, for example source code change or time based events (nightly builds). The second important job is to notify the developers about the status, build states and the outcome of the test run. For maintenance, the server can also keep a history with code snapshots and some metrics [10].

Continuous Integration in combination with a server is about automated building and testing, which can be a really useful tool in today's developing workflow. Several tools exist to set up a continuous integration server, for example *Jenkins* or the Apple own implementation called *XCode Server*.

3.3. Test-Driven Development

3.3.1. The Idea Behind TDD

Kent Beck introduced in his *Extreme Programming guide* [15] Test-Driven development (TDD). Sometimes the strategy is also called test first development, because according to Beck the main goal would be to write tests before creating the actual code that indeed is necessary to be tested. He also mentions that afterwards nearly anybody would write tests, because developers are striving for implementing functionality. Therefore testing is often neglected in traditional approaches of development [16].

Koskela [14] defines test-driven development in one sentence,

> *Only ever write code to fix a failing test [14].*

This is the complete thought behind test-driven development. At first sight it sounds a little unfamiliar, but after understanding the core parts of TDD it will turn out to be a very useful strategy.

Koskela [14] states that every software development project starts with specifications, defining what the program should be able to do. These specifications are then translated to requirements that often tend to be more specific. In a normal development cycle, the requirements would be split up into tasks and these will be implemented one after each other. In test-driven development requirements are formed of tests instead of tasks. These tests build an acceptance criterion that has to be fulfilled. So all tests should pass to prove that the code is good enough. The way unit tests created by developers are written and also their advantages were covered previously in Section 3.2. That is why tests are only about one specific part and not a whole functionality or requirement [14].

Koskela [14] also mentions that the biggest problem in software development is tantamount to solve the *right* problem right. This fact has to be subdivided. The goal consists of solving a given problem that is extracted

from the specification. If something is misinterpreted in this translation phase, then a problem is solved, but it is certainly not the desired one, which was required by the customer. Writing tests that are easier to understand for the customer can lead to an earlier discovery of a difference in the implementation and specification [14].

The second part is to solve the problem right. This can be interpreted in different ways. According to Koskela[14], *right* means that the design of the code should be properly thought through. Poorly designed code has a high defect rate and lacks to be maintainable. This will slow down progress and lower the quality of the end product. Both will lead to longer debugging sessions and therefore longer development phases and higher costs [14].

Therefore Koskela's [14] solution is to work test driven. The quality of the overall product will rise and there is a higher chance that the software will fulfil the customer's expectations. Except for the external quality increase, also the code quality will rise, because before typing the code, thoughts were given to how to solve the problem and how to pass the test properly. Another advantage will be that less time is devoted to fix defects, because the tests will show, where the problem is located and therefore it can be faster corrected. Time for debugging is decreasing dramatically [14].

Test-driven development supports an iterative process of an incremental developing workflow. New functionality is added after every cycle with keeping up previous services working correctly. Each cycle consists of three separated phases. The first phase would be to pick a test of the global test list, then implement the test, which will fail, because the functionality is not implemented at the moment. Then write just enough code to pass the test in the second phase. After all tests are passing try to refactor the written code parts. Refactoring is the last part of every cycle and then the iterative process starts over again [14].

According to Graham [7] people tend to achieve better results when working iteratively on small problems and finish them, before switching to another context, instead of working on multiple problems with different contexts at the same time. Test-driven development helps the developer to focus exactly on one specific task and improve the quality through adequate testing [7].

Together with the acceptance test driven development, which is covering the specification tests, described in Section 3.4, confidence is given that the internal requirements fulfil the external specification. Therefore the software

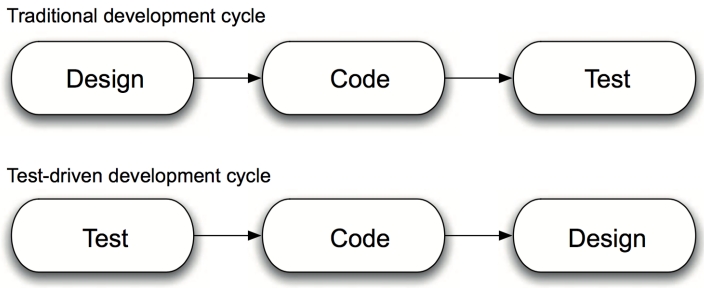

Figure 3.3.: Test-Driven development cycle compared with traditional development cycle [14].

will have a higher chance to satisfy the customers needs [14].

Figure 3.3 compares the traditional approach of a development cycle with the test-driven approach. The phases only changed the order. At first sight, it feels unfamiliar that the design phase is at the end. However, design is interpreted in different ways. In the traditional approach the global design/structure, how to tackle a task, is meant. In the test driven approach, design is recognised as a refactoring process, where code parts are redesigned. The structural design is done in the test phase, when defining the application programming interface (API) for handling the given test. Stepping through the cycle in detail will raise the awareness of the test driven way [14].

3.3.2. Test-Driven Development Cycle

The 3 main steps of the development cycle which are iteratively used were already mentioned:

- Test
- Code
- Design/Refactor

The first part before one can use this cycle would be to decompose the requirements and set up a test list. This step is not trivial, because the tests should be more concrete, more example-like, but satisfy the global specification. This list of tests is a working document, where new tests are

added during the developing process, for example to react to different or new specifications. When everything is set up, a test can be selected and the test-driven cycle starts [14].

Test First

According to Koskela [14], the first part is tantamount to write the test, because according to the definition no code is written if no test for this code exists. The key question would be how to test something that does not exist. This is where the thought about the implementation design comes in place. The developer thinks about how it should be organized, how the method should look like, which arguments does it need and many other factors. Two different cases exist: On the one hand the code has to be extended, but parts of it exist and on the other hand no code exists for this part [14] .

Concerning the first case, just writing a test by using the available classes and see the outcome of the test is enough. Then one would continue with the next step and implement the extended functionality.

The second case is a little more convoluted, but probably emerges on occasion. The task would be to write the test just by imagining that the required class or method exists. Koskela [14] calls this concept *programming by intention*, when using code parts, which are not existing at the time of using them in the tests. Of course, the initial failure would be the compiler warnings and errors. Nevertheless, this is certainly permitted for the first round of this cycle. Compiler errors can be interpreted as a test failure. Therefore, the developer should go on to the next part and implement those classes and method declarations to satisfy the compiler. Then the software will compile, but running this freshly written test will fail, because the method heads that were implemented are not implementing the functionality. This will result in the situation of the first case and the developer can go on to implement the feature so that the test passes. The iterative process is clearly shown in this case-example [14].

Graham [7] noted when beginning to use test-driven development, this part will make the developer think about nothing meaningful has been done, but according to Koskela [14],“a failing test is progress”. This statement is founded on the fact that with the failing test, an indicator (the test itself) exists, which tells the developer if that particular task is finished. The test passes, when the code behaves like it is supposed to do. Later on the

developer finds out that the implementation time is nearly the same, because testing first helps to organize and identify which code has to be written [7].

Kent Beck [15] describes this step as "test infection", because the developer thinks about the requirement in another way, which covers the questions how to test the implementation and not about how to debug it, which is common in the traditional development cycle. If some code part is easily testable, then there is a high chance of having a good design [7].

Implementation

The second phase of test-driven development is called *implementation*. We have given an implemented failing test and the goal is to implement the functionality to satisfy this test. Adding a new class or implementing a method would be possibilities for this specific part. Graham [7] mentioned, "At this stage, it doesn't really matter how you write the code that implements your new API, as long as it passes the test." This means that the code just needs to provide this certain functionality. The core service should be stable as quickly as possible, so it is definitely allowed to introduce some fake values, for example hard coded elements, in the implementation. As far as the implementation covers the test case it is even advised to implement the solution as easy and fast as possible [7].

Graham [7] listed an example to understand the way of implementing using the test first approach according to the given needs. Let's go through that example of a method that should return a greeting. For the first test, which only checks, if a greeting is returned, it is absolutely granted to return a given hard coded string to satisfy the test (as seen in Listing 3.1). Of course, afterwards there might be a need for a more open greeting, for example to dynamically change the name of the person (as seen in Listing 3.2), but this is not covered by this test. According to the definition, the more general implementation might be too much outlay, because there is a chance that it is not needed, because the requirement is not demanding the general version. For the case of a more general version, another test (for a more open requirement) should be added, which would cover this usage. This test is added to the test list and can be selected after this current cycle [7].

Listing 3.1: Implementation example: the first cycle (adapted from Graham [7]).

```
func greeting (name:String) -> String {
        return "Hello, Bob"
}
```

Listing 3.2: Implementation example: 2nd cycle - dynamic name (Graham [7]).

```
func greeting (name:String) -> String {
        return "Hello, " + name
}
```

Koskela [14] calls this technique of implementing in incremental iterative phases *triangulation*. The developer is trying to force the implementation to be a proper one. However, this triangulation service is spread out over a few test-driven development cycles. That is why more tests are needed for a proper use of this technique. This approach can support the developer not to over-engineer and premature optimise the system and accordingly not to waste time on something, which is subsequently not required [14].

Design/Refactor

Koskela [14] calls the last stage of this cycle *Design*, or *Refactor*. The test is passing, so it can be assumed that the implementation is correct. To receive a maintainable design additionally to the testable design, which is already given, the implementation sometimes must be rewritten. In the second stage, it was only about a "quick and dirty" solution. Now the goal is to find the simplest possible design, without breaking the tests. Koskela [14] puts every important part of this last step into one sentence.

> *Refactoring is a disciplined way of transforming code from one state or structure to another, removing duplication, and gradually moving the code toward the best design we can imagine [14].*

Kent Beck [15] also introduced the term "code smell". If the code "smells" then it may implement the correct functionality, but something is looking awkward and not right. It is advised to remove such code parts and to try to solve it in another way [7].

Refactoring can be achieved in different consecutive steps, but after each step it should be checked that the software is still working and was not obviously influenced by those changes.

It can consist of removing a test that appears to be redundant afterwards, because another test is covering the case as well.

Another step would be to refactor to smaller, more compact methods to avoid methods with dozens of lines. A method should cover one thing and it would not be a good design to overload it.

A further step would be to refactor to certain approved patterns. If a pattern exists that does the right thing, it is advised to use it, instead of reinventing the wheel.

Refactoring is covering a wide area of topics, which also includes performance refactoring as well. The mentioned steps are the most used ones in the refactoring stage and which should definitely thought of every time reaching the refactoring stage [14].

3.3.3. Patterns

Some patterns evolved for test-driven development which can be helpful to follow. The latter act like guide sentences one should have in mind for a successful use of the described test-driven development cycle.

Red, Green, Refactor

The first guide called *Red, Green, Refactor*, introduced by Beck [16], is the heartbeat of the development cycle in a more memorable way. *Red* visualises the stage of writing the failing test. Many integrated development environments, or IDEs, show a failing test with a big red bar or other red symbolics. The *Green* phase is exactly the other way round, where all tests are passing and the green colour should represent that progress. The *refactoring* stays the same. So every developer wants to reach the save green place [7].

"Ya Ain't Gonna Need It"

According to Graham [7] this mnemonic is about implementing the initial failing test. It advises to just write the code, which passes this specific test, because then unused code is never written. Developers tend to think about

what can be required later on and therefore write methods very open to different scenarios, sometimes with a big effort. For test-driven development one should stick to only pass the test without any other feature, because it is not needed at the moment, and it could be the case that is not required in the future as well. Spending time on implementing not required additions would be a lost effort and lead to unused code. Graham [7] says that, "Test-driven development encourages building applications from the outside in.You know that the user needs to do a certain task, so you write a test that asserts this task can be done". Just implement what is necessary for the moment [7].

Fake It('Til You Make It)

Another mnemonic introduced by Kent Beck[16] is about faking values or even whole methods. He advises to return the expected constant to support the test. And then use the previously described technique *trianglation*, with writing other tests and fulfilling them to reach the more general goal [16].

Breadth-first, Depth-first

Breadth-first and Depth-first are two different approaches described in Koselas [14] work about *Practical Tdd*, which can be compared to the searching through for example a decision tree. *Breadth-first* would cover to first select all the API concerning tests and just fake the methods to pass those tests, and then go one step deeper in the hierarchy and implement more functionality for all those tests. *Depth-first* would be about taking one test and implement this functionality till reaching the fully integrated functionality before going on to another test context. Of course, the outcome will be the same one, but it helps selecting the next test, if strictly following this concept. The use of these approaches has to be decided in every case itself, because of the different environment/background conditions [14].

3.3.4. Usage Of Test-Driven Development In A Project

Graham [7] states that test-driven development teaches to incrementally implement the software. If starting a project and following these rules, the result would be an architecture of separated components, which are

somewhere linked together. This would not be a proper architecture because as Graham mentions it only "shares many of the characteristics of concrete" [7]. No overall structure will be recognisable, and therefore it is difficult to predict how each feature behaves together with other ones. These ones will always be separated parts that will not relate to each other.

Graham [7] advises that it would be advantageous to have at least a rough general idea of the overall structure, before beginning with the test-driven development cycle. Accordingly, the global design should be defined and the component's design will form of the writing of the tests and thinking about the implementation before writing code parts. The design should consist of what the general idea is, how each feature fits with the others and how they will be communicating. This is called *Domain Model* in object oriented programming. Thinking about the use of the user, how they would interact with the services or features at which time and create an architecture plan of this information. In this plan, the communication or double use of methods can be defined, so that a rough guide exists to lead the test-driven development into the right direction [7].

To summarise, the main idea is to have knowledge of the overall structure and then build up the software with small components fitting perfectly together using test-driven development.

The general test-driven circle, with its three components (test, implement and refactor) is just a coarse overview. To use test-driven development efficiently in a project this circle needs to be refined into a concrete workflow, which every developer has to follow. This workflow must include triangulation, several test runs and an incremental refactoring phase. A more detailed test-driven development workflow, which was used in the practical part of this thesis, is shown in Figure 3.4. If this workflow is followed, it is guaranteed that the test-driven development circle will be maintained throughout the project development [7].

3.3.5. Role Of The Documentation

Osherove [10] covers the problem that each software documentation has to be kept up to date. If something is changed in code, then the documentation needs to be changed accordingly. As mentioned in Section 3.2.1 tests can

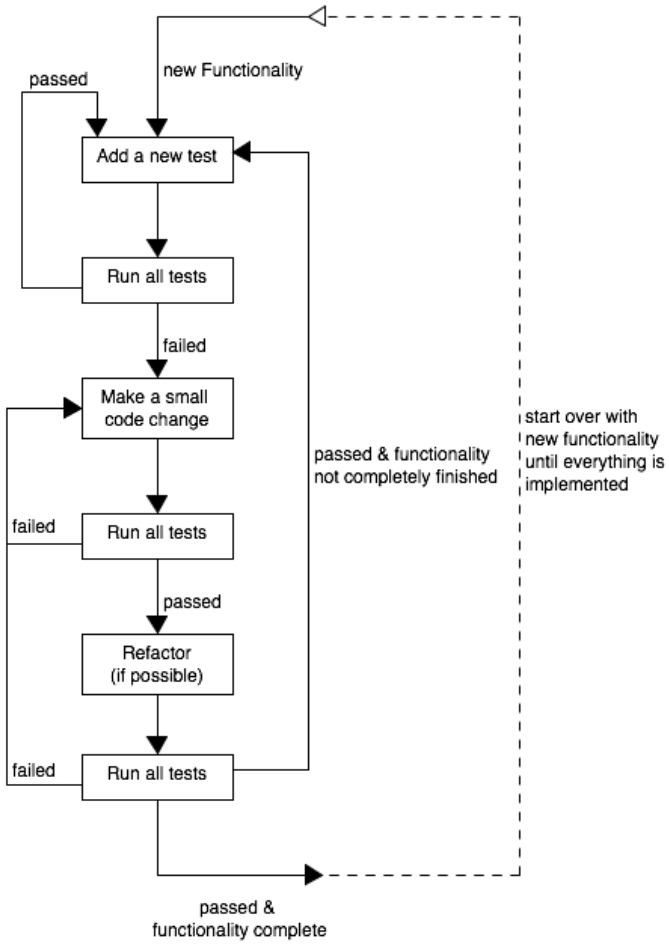

Figure 3.4.: Test-Driven development workflow used in the practical part.

imitate the role as a living documentation. Since tests are written before coding the implementation in test-driven development, a huge test base exists. All tests have to pass to reach a green build. Therefore all tests have to be up to date. Considering the tests as a documentation, then it is already up to date after every cycle and nothing needs to be done in addition. Of course, this kind of documentation is a real technical one. However, to understand the code it is really useful to consider the tests and see how the code parts are used properly. Combined with acceptance test-driven development (3.4) and it's emerged tests, the functionality will be easy readable, even for non technical-skillful customers.

In general testing will help to keep pace with a working and useful documentation, but it should not completely replace a written one. Global behaviour should be declared in written form and not only by the tests themselves.

3.3.6. Performance And Impact Of Test-Driven Development

Is test-driven development really as efficient as it is intended to be? This is the core question to answer and then to decide if TDD should be used or not. Koskela [14] listed a few advantages that actually exist and need no additional proof, because if following the cycle properly then it will behave as it claims. These advantages contain,

- Written code is testable, because otherwise it would not be implemented
- Effective against regression errors, because of the number of automated tests in the background
- Deployable version after each cycle
- Fixing defects is less time consuming because of the tests
- Time consumption is nearly equal, when comparing implementing the feature and writing tests afterwards $\rightarrow$ same amount of code
- Living documentation for *free*

The most important feature in the business would be that there is always something to deliver, which might not have all features, but something that is actually working. The structure and design of the testable code is

also given by just following the instructions of TDD. Fewer defects will be introduced and they are easier to locate due to the test suite [14].

Two independent studies have shown the impact of test-driven development in comparison to the standard models. The first one, accomplished by Thirumales [17], is about a one year long empirical study at IBM. Using the TDD approach about 40% fewer defects were found in comparison of the baseline prior product without impact of the productivity of this team. Another study showed that the quality of groups working with TDD was higher because 18% more black-box tests were passed instead of the control group, but they took 16% longer for the implementation, so performance was kind of decreased in the implementation phase [17].

The second study, which was accomplished by Janzen[18], investigated two different environments. One was at an industry environment and the other one in academics, more precisely at a university.

Janzen's study [18] was accomplished by forming three groups, where each group followed another technique (test-first, test-last, no testing). The results will be covered now:

For the industry, nearly the same results were discovered as Thirumales did. The produced code passed 18% to 50% more external tests and the productivity decreased by 16%, which is minimal according to him. These results were obtained by letting the same group of developers using both approaches to solve two different problems. In academic studies different empirical results were revealed. Two of the five hypothetical studies showed a big quality improvement and productivity improvement. One correlated with industrial results and surprisingly two did not show improvements in quality or productivity. According to him, the reason could be that the programmers had little or no experience in TDD programming [18].

Generally spoke, covering both environments, the test-first approach consisted of a lower complexity, fewer defects, fewer statements and more method parameters. The hypothesis of being more productive than with the other approaches is likely to be rejected, because the productivity in these studies were nearly the same or a bit lower in this phase. However, time savings afterwards will compensate this decrease of productivity. The test-first and test-last approach produced eventually more code and shorter methods than the *no testing* team. The internal quality was further increased by the test-first way [18].

Janzen [18] also provided a survey that had to answered by the developers

afterwards. These surveys showed that over 75% thought that the TDD produces a simpler design and would be the best approach. 70% thought that there were fewer defects when writing the tests first. The test first team was more confident about their code and indicated that they will choose this method again. The No-Test group would be more confident by using the Test-Last technique. According to Janzen [18], they are more comfortable with something they already knew [18].

Test-Last and Test-First provided a higher quality, however Test-First had some other positive by-products, for example coupling is higher, complexity is lower and of course the test coverage is higher, because in the Test-Last technique, tests are not always written, because of laziness. Conforming to Janzen, proper use of TDD would pay for itself over time and accordingly it should be used preferably [18].

3.4. Acceptance Test-Driven Development

According to Ambler [19] Acceptance test driven development(ATTD), sometimes also called Behaviour driven development, is based on the same intuition as test driven development, which consists of writing a test and changing the given construction till it works [19].

However, one big difference exists, because test driven development is about developer tests (unit tests) that are designed and written by the programmers themselves. In acceptance test driven development, tests are easier to be understood by the customer, who defines the specifications. Therefore, these acceptance tests cover the given specification to extract some specific requirements that are handed over to the developers, and see if the implementation covers those as well [14].

According to Koskela [14], "Acceptance test-driven development (acceptance TDD) is what helps developers build high-quality software that fulfills the business's needs as reliably as TDD helps ensure the software's technical quality".

According to Ambler [19] the main goal, as illustrated in the uml diagram 3.5, is, to get specific requirements, which are then implemented for example using test driven development. After implementing the functionality using test driven development, the result is checked against the given specification. The correctness of the implementation is then covered by TDD and the cor-

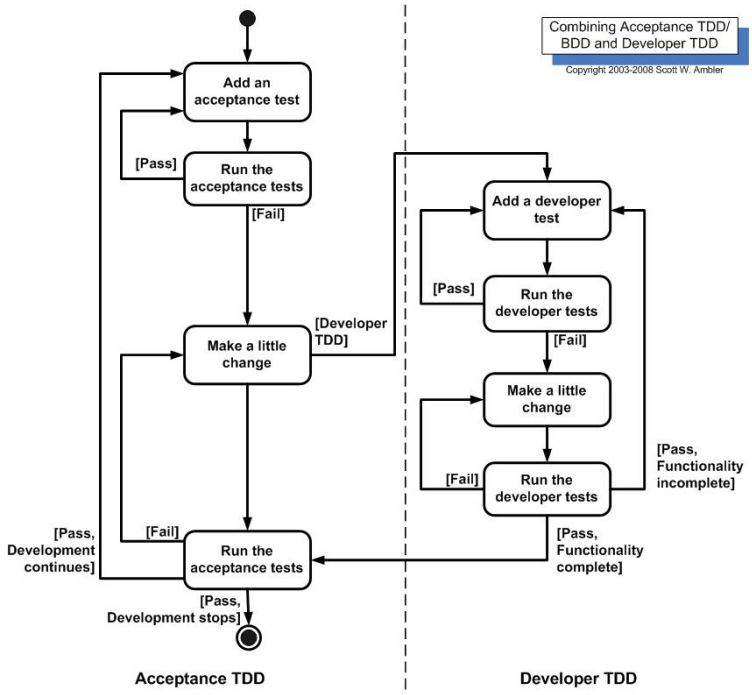

Figure 3.5.: A combination of acceptance TDD and developer test driven development [19].

rectness of the functionality itself is covered by acceptance TDD. Combining both techniques can be a good lead to satisfying the customer [19].

3.5. Testing + iOS Development

According to Graham [7] testing had a hard time to prevail in iOS development. In the early years, after the AppStore was introduced the testing process was a variation of the previously mentioned "Trail and error method". The testing part was only covering testing the application in the iOS simulator or on a development enabled iOS device to check if everything is working as expected. Nowadays, the importance of testing also reached the iOS developers. More and more instructions can be found to test different specific iOS structures and commonly used pattern to write unit tests evolved, since iOS development has a huge fanbase [7].

Salibra [20] mentions that even distributing beta versions to beta testers was a very hard task, because of the capsuled Apple environment that does not allow to install applications on devices, which are not downloaded from the own App Store. Therefore the device, on that the app should be running on, has to be added to the developer account. Then it was allowed to deploy the app on this peculiar device when it was connected to the Mac via USB connection. This changed over the years and also pushed programmers to develop with testing in mind [20].

Schroppe and Eggert [21] mentions that Apple also improved the way continuous integration is used in iOS projects. When using an Apple server, which can be installed on every Macintosh, the integration is straightforward. Within this server, triggers can be defined to check out the code and run tests automatically. This is all achieved with a simple, but yet powerful, user interface. After executing the tests, the developers can be informed in different ways, for example with the dashboard that is providing a good and clean overview [21].

Further, these tests do not complex the structure of the application code base, because they are handled in another test suite project, which is linked to the original one. Therefore a good practice would be to keep up the same hierarchy in both projects. One extra useful addition allows to exclude tests from running in different environments. For example, testing the actual built-in sensors cannot be tested in the simulator without mocking, therefore

excluding those in simulator test runs is a good choice. This also holds for every other connectable hardware if it is not connected. This feature is really handy for the Bluetooth implementation in the practical part afterwards [22].

3.5.1. Tools For Testing

Apple is providing useful frameworks to sufficiently test an application. The most important framework is *XCTest* [22].

XCTest provides the core functionality for unit testing. These tests are treated the same way as classes are. Each class subclasses the *XCTestcase* core class. This core class provides functionality for setup and tears down code to improve the performance of testing. Furthermore, since these tests are like methods in a class, helper methods can be written to avoid duplicating code parts in tests as well. So what is the difference between helper methods and test cases? There is a naming convention used to distinguish between those two types. Every method starting with "test" is considered as a test case and therefore executed in the test run. If it does not start with "test", then it will be excluded and XCTest will skip this method when running the tests [21].

As Apple [22] mentions in the Testing guide, since iOS 9, two more types of testing tools/testing improvements have been added. The first one is about simplifying the beta testing process. Testing devices do not have to be added to the developer account to run the application anymore. Another improvement is that distributing to beta testers is now able over the air with Apple's own app called "Test Flight". If a beta tester is enrolled in the testing process of an application, then the user gets notified to download every new provided build from the Test Flight app. So developers do not have to gather all devices, but only upload the build to the iTunes Connect account to deliver a new beta version to their testers. There is even a communication connection for beta testers to the developers, which can be very useful if bugs have been discovered. This is a genuine improvement and can save a lot of time for developers [22].

The second big enhancement in the iOS testing environment would be the possibility to create *User Interface Tests*. This is a giant step forward, because it was not possible to test the user interface without some third

party tools that were not officially allowed, before iOS9. Testing the user interface automatically can lead to a formidable increase in user experience, because not only the functionality is tested but also the use and interaction [22].

There are other tools as well, like the Apple server as mentioned before and additional third party tools to improve the testing process. However, when using the Apple provided tools properly then sufficient testing can be reached.

3.5.2. Writing Tests

Writing unit tests in iOS is like creating a class with methods. Each method starting with "test" is considered as a test case afterwards of the framework. Before each test case, the setup method is called to execute some setup functionality. After each test, the teardown method is executed to free some objects. Therefore you have control about what happens between every single test case.

According to Schroppe and Eggert [21] it is helpful to use a pattern to structure each test. They introduced a so called *Given-When-Then* pattern. Each test is divided into these three parts. The *Given* part consists of the environment setup, like creating objects or setting certain states. The *When* part is where the action happens. In this part the code to be tested is called, which is most likely one method call. The *Then* section is right after that call and includes the checking of the test result. Assertions are written to check if the expectation is fulfilled. With this pattern the reading and understanding of the tests is simplified [21].

It is really straightforward to "write" those user interface tests provided by an Apple user interface framework [22]. The only thing to do is to start the simulator and record clicks and typings on the user interface to create those tests. If the desired functionality has been recorded the developer has to add asserts to finish writing a user interface test case. That is why writing was set under quotation marks. There is just one prerequisite for recording the tests, which says that the graphical elements have to implement the accessibility methods. These are certainly pre implemented for all core user interface elements and therefore for all subclassed versions of it. But if own user interface elements are designed, one has to concern about the

accessibility on your own [22].

3.5.3. Creating Mock Objects In iOS

As mentioned previously when defining unit tests it is sometimes necessary to create mock objects. There is no built-in functionality for this case but several good frameworks exist including *OCMock*[1] and *Mockito*[2]. Thus far these frameworks only support Objective-c and not Swift. It is very easy to use those frameworks, because they are managing the encapsulating of classes on their own. Accordingly, creating a mock object is mostly just a line of code when using the frameworks properly [21].

For implementing mocks with swift, a commonly used way would be to implement every class with a class protocol and create mock classes on your own. This protocol pattern was also used in the following practical part.

3.6. Testing Especially With Hardware Connection

Software, which is communicating with other hardware, is exposed to many influences that the software will not work as expected, for example due to the communication. That is why it has to be guaranteed that both devices, both software on these devices and the connection are working as desired. Another point when speaking about communication would be that this should be working right at that moment when the action happens. Sometimes no influence is given to control the extra communication partner, because it is developed by another company or is running a standard software to offer the communication and control every feature. In the practical part, an Arduino device is used, running the standard Firmata software to enable access to all pins (see Section 4.1). Therefore no control is given to change the code or behaviour of this communication partner.

As mentioned previously, the second error prone component can be the communication part itself. The latter can occur in many different ways. The simplest would be a cable connection, which would be the most reliable

[1] **https://ocmock.org**.
[2] **http://mockito.org**.

solution. An additional one and perhaps most used solution nowadays would be the wireless communication. For smartphones and the purpose of connection to the Arduino, Bluetooth was chosen. Of course wireless is not that reliable as a wired connection, because messages can be lost due to the amount of traffic in the transmission medium or due to the distance between the two devices in that given environment. For example, if there is no line of sight, then the communication range will shrink accordingly.

So how should such a complex structure be tested? There are two controversial approaches to test, *Mocking* and *Real Environment Hardware Tests*. Both have advantages and disadvantages, which will be clarified and compared now.

3.6.1. Mocking

The first approach is about *Mocking*, which is covered in Section 3.2.4 in general and for specific iOS use in Section 3.5.3. In this special case mocking has a few different tasks to manage and this should be done separated, because as mentioned previously, only one specific functionality should be considered in one test case.

The first one would be to test the Bluetooth communication itself, more precisely the application programming interface of the Bluetooth communication. Therefore, mocking a peripheral and letting the software communicate with this faked outlying device should test the right part.

The second part would consist of testing the communication with the desired device. Accordingly the device is created as a class in the central device software and is answering with fake values. Together with this approach, it can be ensured that the software to test is handling the received values correct and is sending the appropriate commands to the peripheral device.

There are some advantages using this way, because no additional device and no real Bluetooth connection is needed. That is why these tests can be managed by everyone and even on a continuous integration server as well, since now the peripheral device is mimicked in code, to check if no regression errors were introduced. Another advantage is the case that these tests are reliable and are consistent in case of the outcome if not changed. However, there are some gaps in completely testing the feature. For example,

what happens if a message is lost over the Bluetooth connection. Mocking is only mimicking a seamless environment and connection.

3.6.2. Real Environment Hardware Tests

With mocking, communication is only mimicked as a function call. This is where *Real Environment Hardware Tests* comes in place. These tests use a few prerequisites. First of all, the communication partner needs to be in range and of course ready to connect. After the connection is established, the tests can be executed. For sake of independence the tests connection should be happening in the test setup phase and disconnecting in the test teardown method accordingly. Now these tests cover the complete structure, which is also used by the end user. This is the biggest advantage of this kind of testing. So this would be the perfect approach to test such a complex structure?

Certainly not only advantages exist with this approach. There is no consistent behaviour, because it is not guaranteed that the connection is set up, nor that messages are transmitted properly. This behaviour of test result inconsistency can lead to confusion and accordingly to debug sessions even though just some message got lost due to environmental changes. This would probably be the case when running those tests automatically on servers with the hardware in range. There is a high chance of some tests failing randomly. This is the wrong intuition of automated testing and can lead to more trouble. Another negative point would be that everyone, who wants to test, has to prepare the setup in advance. Thinking of Arduino, different hardware implementations exists. To cover every kind of these, one has to test with all different types of devices, which is certainly not possible.

3.6.3. Comparison & Conclusion

Summarised, both methods cover different parts. *Mocking* is a good choice when including the tests into an automated test suite, but it does not cover the real world behaviour. The double object only replies with mimicked values, initialised by the developer. Mocking calls the interface and sends the expected value. Therefore, the only possibility of error could be in the difference between how the device handles this data compared to how it

actually should behave. Hardware is never entirely reliable. That is why some additional error handling needs to be implemented.

Implementing tests only with Mocks is no trustworthy advice. One should have at least some weeks access to the peripheral device to test the interaction with the software. Afterwards mocking can be a good choice to ensure the functionality is offered at every time.

At this point, only unit tests were considered, but for hardware connections the value of integration tests should not be underestimated. These tests cover the whole user experience of the interaction with the other device.

So far, *Real Environment Hardware Test* were not mentioned, but they would cover all functionality and the user experience as well. Unfortunately diverse kinds of hardware and interferences complicate the testing process. This can lead to uncontrollable behaviour, which is not the right keynote of testing. This randomly failing of tests could lead to debugging sessions even though only an interference was blocking a message.

Since different types of hardware exist and not all of them can be tested, full coverage of *Real Environment Hardware Tests* is also not given. Accordingly, it is not tremendously better than *Mocking* speaking about test coverage.

When thinking about Apple and iOS testing, then the simulator is not in a position to communicate over Bluetooth, even though the server has Bluetooth build in, because Apple has blocked this hardware support a few years ago. That is why the setup for real hardware tests has to consist of an iOS device and the communication partner.

In conclusion *Mocking* would be the preferred solution for testing Bluetooth connection as covered in the practical part, because of the possibility to add those test cases to the automated test suite. However, the best solution would be to implement both approaches and exclude the *Real Environment Hardware Tests* of the automated test suite. So these tests could be executed every certain time (for example before each release) to check with actual conditions. Therefore, the Bluetooth setup only needs to be built if running these test and not every time the automated test suite is executed. Another advantage would be that the server has consistent test output and there is no need to install the setup near the continuous integration server, which can also imply difficulties. When combining the advantages of both approaches testing hardware can be achieved properly.

3.7. Test driven development in Catty

All sub teams of the Catrobat project strive for an agile development workflow. As mentioned in the first steps for new contributors at the Catrobat developer webpage[3], it is necessary to agree on the defined way of contributing. This includes strict usage of test driven development and clean coding. Of course, this is also practiced in the Catty (iOS) sub team. For new members, located at the university of Technology Graz, we provide kind of trainings together with senior members to learn how to be productive with test driven development and of course how to write useful test cases. The team is also trying to help new members by following the pair programming concept[4], where a senior member helps new members to deal with their first issues to get confident about the code structure, the workflow and the whole project. In my opinion, if everyone in the team has kind of fun developing new functionality or fixing bugs then the quality will rise and progress can be seen.

Now heading back to the development workflow. The standard workflow would be that the project has to be forked or a new branch of the core/master branch of the team has to be created. Afterwards, by using the previously mentioned techniques, the issue has to be solved, which can be a new functionality or a known bug. If the implementation is done, which is accompanied by the newly added tests and all other tests are passing on the test server, then one can ask for review via a pull request. This request will be checked by one of the senior members and if everything is working as expected it will get merged into the master branch.

This workflow uses many techniques of an agile development, starting with test driven development over continuous integration to double checking the code and its style by other developers. Accordingly, it is granted that the functionality of the master branch is always preserved. Such workflows, like the presented one, are commonly used in today's area of development.

[3] **http://developer.catrobat.org/first˙steps**.
[4] **http://c2.com/cgi/wiki?PairProgramming**.

4. Hardware Background

This chapter contains the theoretical background of Bluetooth and Arduino. Both components will be used in the practical part (Section 5) for the integration of controlling Arduino boards with Pocket Code for iOS. The theoretical background consists of technical details and protocols that are used later on to communicate with the Arduino board. To understand how the communication between the two devices is established and messages are exchanged, it is essential to know the basics about the Bluetooth technology as well as the differences between the diverse versions and how Apple is approaching the topic. Additionally, it is advantageous to know how the Arduino board is working in general. Furthermore, a special and widespread protocol for Arduino, called *Firmata*, is used to get access to all functionality of the connected microcontroller.

4.1. Arduino

> When we have an idea, we take a pen and we sketch it down on a piece of paper. Imagine if we could build things that interact with the environment just as easily [23].

This statement of Amariei [23] describes exactly the idea behind Arduino. It should be possible to test a hardware setup and connections to interact with the environment without big effort. The Arduino founders [24] at Ivrea Interaction Design Institute (IDII) go a step further and developed Arduino as an open-source prototyping platform. The goal was to allow fast prototyping with hardly any knowledge about electronics or in depth programming. These electronic microprocessor boards interact as an interface between the environment and the software implementation, because different types of sensors or other kinds of inputs can be interpreted. The system also allows to set a bunch of different outputs. This is achieved by interacting with

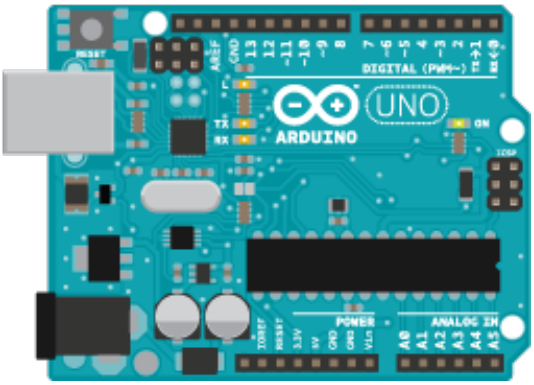

Figure 4.1.: Arduino schematic image [24].

the board using the Arduino programming language. The latter is based on C/C++ with special extensions for the purpose of microcontrollers. An Arduino Uno in its original design is illustrated in Figure 4.1.

What exactly defines a microcontroller? Techtopia[25] defines it as followed,

> A microcontroller is a computer present in a single integrated circuit which is dedicated to perform one task and execute one specific application. It contains memory, programmable input/output peripherals as well a processor. Microcontrollers are mostly designed for embedded applications [25].

As stated by Smith [26] the essential part of developing with Arduino is to understand that it behaves like a small computer with the built-in processor, memory and some in- and outputs. Nevertheless, all components are limited in their functionality for reasons like costs and size.

Since this project was designed as open-source hardware, many people contributed and therefore the community created different types of boards adapted to diverse needs, tutorials and hardware sensors especially for Arduino. As reported by Kushner [27] the project became the leading open-source hardware microcontroller board and really revolutionized do-it-yourself (DIY) in electronics. Dale Dougherty, publisher of Make[1], stated, that Arduino is “the brains of maker projects”.

[1] **http://makezine.com**.

According to the Arduino founders [24], the key to success was the wide range of Arduino, so that everyone can build interactive projects. It is feasible to learn for beginners, but very flexible for experts. The founder Cuartielles also mentioned in an interview to Kushner that "The philosophy behind Arduino is that if you want to learn electronics, you should be able to learn as you go from day one, instead of starting by learning algebra"[27].

Furthermore, it runs on Windows, Linux and Mac, so every major operation system is covered out of the box. Kushner [27] found out that this plug-and-play feature stood out from the competition which was further a reason for success. With this instant availability for creating things and the flexibility of Arduino the community used these boards in all kinds of fields starting with robotics over musical instruments to complete home automation. With these few examples, the power behind Arduino can be clearly seen. Consequently, it is very compelling to see what users can achieve with a smartphone, Pocket Code and a wirelessly connected Arduino device [27].

There are additional microprocessor boards around, nevertheless Arduino is state of the art for building electronic projects. The founders see some advantages of Arduino[24]:

- Inexpensive - compared to the other boards
- Cross-platform - as mentioned before it runs on all major platforms
- Intuitive programming environment
- Open source hardware - the plans of the boards are under Creative Common license
- Open source Software

The standard hardware is based on an Atmel AVR-microcontroller of the megaAVR-series (for example ATmega328P) operating with 5V. The clock frequency is 8, 16 or 32 MHz according to the built-in micro-controller. The boards contain a USB connection for programming with a personal computer and many inputs/outputs to connect sensors. This is only a small overview, because there are many extended versions, where some parts of the hardware are designed for special needs.

Firmata

The main idea behind working with Arduino was key to program the Arduino controller and let that program run in an infinite loop. For com-

munication between the microcontroller and software of another device, a protocol is needed. A commonly used protocol for this purpose is called Firmata.

According to Steiner [28], the creator of Firmata, the goal was to "make the microcontroller an extension of the programming environment on the host computer in a manner that feels natural in that programming environment". An extension is better integrated and therefore preferred by developers instead of an excluded system with own methods.

Design

As stated in the documentation of Firmata [29], it is using the midi-message format to create messages with integrated commands. A universal set of commands, containing

- analog I/O message
- digital I/O message
- report analog/digital pin
- sysex start
- set pin mode(I/O)
- sysex end
- protocol version
- reset system

is used for the Arduino Firmata implementation. An example for a command message would be as followed. In Arduino, each pin has to be set to a mode before using it. These modes represent different types, for instance analog input, digital in/output, PWM output. With Firmata, the developer can use the *set pin mode* command and set a pin to the required mode before acting with the pin itself. Given this example, the structure of the protocol can be analysed. There is always a command (a specified hexadecimal value) followed by one or more statements. These statements are kind of parameters in function calls. In Table 4.1 two examples of common commands are shown. In the first example the command of an analog message would be *oxEo* and the parameters are the pin number, and the value that is split in least and most significant bits.

Using these commands, it is allowed to control "...as much of the Arduino [...] as possible from the host computer"[30], which was a main goal of

type	command	MIDI	first byte	second byte
analog message	0xE0	4(pin)	LSB(bits 0-6)	MSB(bits 7-13)
report digital port	0xD0	0(port)	1/0 (enabled/disabled)	n/a

Table 4.1.: Example Firmata messages

the initiator Steiner [28]. Since this protocol is based on the MIDI protocol, messages are limited to 1 to 3 bytes. Conforming to the protocol [29], messages are byte sized with a 7 bit data resolution, 7 bit command space and 1 bit for the command itself. If there is need to transfer more bytes, then SysEx messages are used. The idea behind these messages is that a "start SysEx" tells the receiver that a longer message will be transmitted. After finishing sending this message an "end SysEx" is introduced to tell the receiver the end of a message. Therefore the receiver just parses the received data till an "end SysEx" appears and do not have to know about the length of the message in advance. For instance SysEx is used for configuration messages in Firmata (all kinds of queries, for example pin mapping query, where the state and mode of all pins are transferred).

According to Steiner [28], MIDI was chosen because it is efficient and comparatively easy to implement. With the new command set and data types of analog and digital pins the whole Arduino API can be interpreted with those Firmata-messages.

Usage

To use the Firmata protocol, both communication partners have to implement the Firmata protocol and therefore the ability to communicate over this protocol. For Arduino the *Standard-Firmata* is supplied in the Arduino integrated development environment and only has to be uploaded to the Arduino board, sometimes with small changes due to different Bluetooth modules, when using Bluetooth as a transmission type. As mentioned before, the other device has to implement the Firmata protocol as well. There are several Firmata implementations for clients including python, java and iOS. The whole list can be found on Github[2]. In listing 4.1 a python example of Firmata usage is shown, which is straightforward to use. Unfortunately,

[2] **https://github.com/firmata, [accessed 27.11.2015]**.

the iOS version is out of date and uses another outdated Bluetooth framework as well. Accordingly, a Firmata implementation for iOS in Swift was developed during this thesis. Further details of the implemented Firmata library are described in Section 5.2.1. The complete and up-to-date protocol definition is available on GitHub [31].

Listing 4.1: Firmata Python example

```
import processing.serial.*;
import cc.arduino.*;
Arduino arduino;

void setup() {
        arduino = new Arduino(this, Arduino.list()[0]);
        arduino.pinMode(5, Arduino.INPUT);
}

void draw() {
        if (arduino.digitalRead(5) == Arduino.HIGH)
                println("Digital pin 5 is HIGH");
        else
                println("Digital pin 5 is LOW");
        print("Analog pin 0 value is ");
        println(arduino.analogRead(0));
}
```

4.2. Bluetooth

Bluetooth [32] was introduced with the primary goal to avoid cable connections and exchange data between different devices without the necessary of a line of sight. It has been established in 1994 by Nokia and was then maintained by the Bluetooth Special Interest Group (SIG) [33] [34], a group of companies, who are creating and selling Bluetooth products. The wireless network service rapidly became a standard in the mobile device sector and due to the phenomenal increase of portable equipment devices in the last couple of years, it was adapted to special needs. Since the release of

Bluetooth, 4 major versions were published. In Table 4.2, a short overview is listed.

Version	Short description
1.x	initial release and some improvements afterwards
2.x	higher data-rate, better pairing
3.x	introduced a high speed channel
4.x	Bluetooth Low Energy, new IP connections, not compatible to < 4.0

Table 4.2.: Major Bluetooth versions [35].

In this document only Bluetooth Low Energy (BLE), also called Bluetooth Smart, is considered, because Apple is not generally allowing to connect to all devices that are only supporting Bluetooth version lower than 4.0. The second argument for only supporting Bluetooth 4.0 and higher would be that there are some differences in the implementation with previously defined specifications, called Bluetooth BR/EDR, and the current versions. This variant of the standard is optimized to low power with extended duty cycles containing geared limited maximum data size and low cost. Additionally it builds upon Generic Attributes to enable a more general, open use of the BLE protocol [36].

According to Townsend [37], after the release of Bluetooth Low Energy in 2010, there was a radical change, because the two wireless communication specifications are not fully compatible. Bluetooth version 3 is not out of date, because it is needed for data-heavy tasks like streaming, which would not be achievable with the new power saving specification [37].

Some manufacturers are implementing chips called *Bluetooth Smart Ready*, which are capable of communicating with both standards. Unfortunately "ready" is an unlucky name because these chips fully support both specifications details. These types of Bluetooth transceivers are built-in today's smartphones, so that they have the ability to communicate with all Bluetooth standards. However, if a connection between a Bluetooth Smart Chip, for example those in fitness devices, and another device should be established, then the specification of Bluetooth 4.0 or higher is required. That is why the Bluetooth receiver needs to be implemented as Bluetooth Smart or Bluetooth Smart Ready.

Apple has built-in Smart Ready Bluetooth transceivers in their iOS devices, but there are some restrictions on use, which is covered in Section 4.2.3.

4.2.1. Bluetooth Basics

As mentioned in Bluetooth technology basics [38], Bluetooth uses a radio and accordingly radio waves to communicate with another devices. Owing to the transmission power, the transmission is limited to a short range. *Piconets*, ad hoc networks, are built automatically in the size of two to eight devices in a certain radio proximity range. One of these devices in a Piconet acts as a Master, the others as Slaves. To connect more than eight devices, Piconets can be linked to each other via one slave (then called Bridge). This Bridge is then communicating with both Masters of the two networks and thus able to forward messages. These linked Piconets are called *Scatternets*. To find additional devices in the 79 different available radio channels, a hopping algorithm is used, where each device with no Master in range jumps randomly from one to another channel till a Master is found. When integrated in a network each device receives the next hop count from the Master to be able to not to leave the network connection unintentionally, because of a wrong radio channel. For communication the Master grants each Slave a certain time to exchange data [38].

4.2.2. Bluetooth Low Energy

As mentioned in Townsend book "Getting Started with Bluetooth Low Energy" [37], BLE is built upon power efficiency to support devices to be able to run for a long time on a single charge. This fact is important for the growing field of Internet of Things (IoT), where small devices do not have space for larger batteries. Nevertheless, Smart Bluetooth has additional features, including the connection modes and structure of managing data values as well.

Bluetooth devices that are only supporting versions lower than Bluetooth 4.0, have to pair each other by sending the pair code to be able to communicate. With the new specification 4.0 and higher, there are two different ways to reach a communication: *broadcasting* and *connections* (Figure 4.2). With broadcasting, data can be forwarded to any receiving device in listening range. However, this way only allows to communicate in one direction, therefore it would be a good choice if it is necessary to send messages to many device every given time. Connections use another approach, where data can be sent bidirectional between two devices. This mode is used

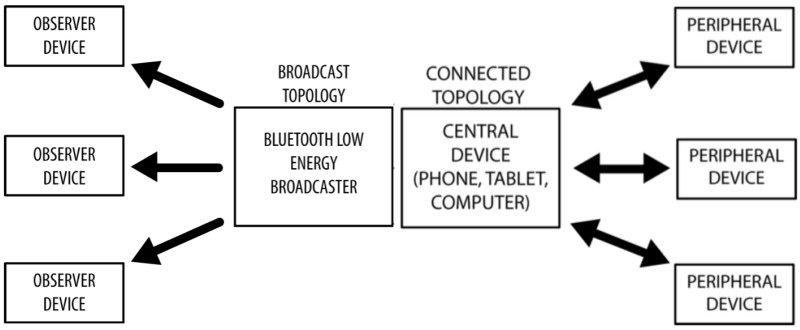

Figure 4.2.: Bluetooth Low Energy different communication types [37].

afterwards in the implementation phase (Section 5), because the application wants sensor data of the Arduino board and is also setting some outputs. Therefore, the ability of bidirectional communication needs to be provided. An overview of the connection mode will be described in the following sections [37]v:

Building Connection

Townsend [37] states that the Generic Access Profile(GAP) is in charge of the connection between devices. It makes peripherals visible to the others by sending out advertising packets in a duty cycle and managing, who is allowed to connect to each other. GAP defines roles for devices. These two roles are involved in a connection:

- Central
- Peripheral

The Peripheral, who acts as a slave, sends out advertising packets continuously. The Central is scanning for those packets and can initiate a connection to a Peripheral. After a connection is established, data is exchanged periodically as shown in Figure 4.3.

Both devices can be simultaneously connected to other devices if the Central/Peripheral roles are observed. One big advantage of connections is the availability of attributes that organise the data provided by each device. The latter can help to reduce power consumption, because it is not necessary to send all attributes if they are not needed by the application running on the central device [37].

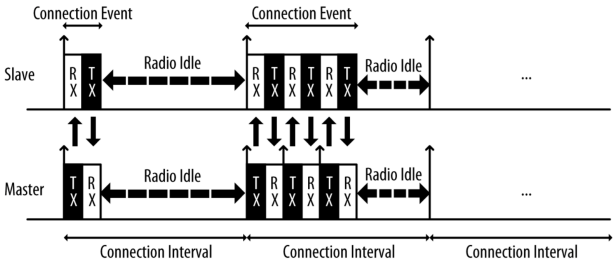

Figure 4.3.: Bluetooth Low Energy connection [37].

Concept

Townsend [37] also highlights that Bluetooth versions lower than Bluetooth Low Energy only support special attribute profiles, standardised in the specification. The revised specification is more exposed to individual profiles and therefore enabling a more general use of Bluetooth with own protocols. These profiles and embedded attributes are treated in a Generic Attribute Profile (GATT). GATT defines a kind of concept how data is exchanged. Before sender and receiver are able to use this feature a connection has to be established between them by the Generic Access profile (GAP) as mentioned before. This generic consists of three attributes which built each other. The hierarchy is shown in Figure 4.4 and managed as followed:

- **Service**: each service is a collection of belonging together characteristics.
- **Characteristic**: each characteristic has a type (a unique UUID identifier), a permission, a specific value and a descriptor if needed. Since the data consists only of bits, the descriptor or a convention helps interpreting them.
- **Descriptor**: a descriptor is an additional information of a characteristic value for example a unit but is not required.

Each characteristic can have different permissions to have control, which characteristic is editable or not. Permissions of a characteristic are structured as followed [37]:

- None
- Readable

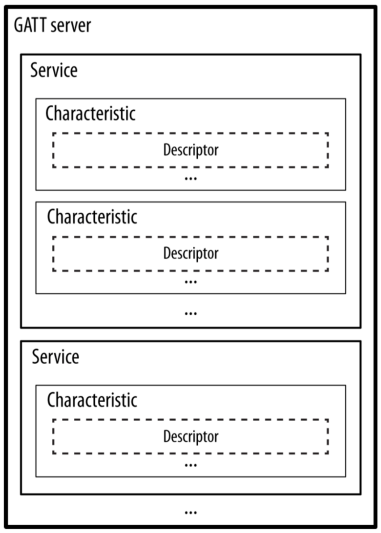

Figure 4.4.: GATT data hierarchy [37].

- Writeable
- Readable and writeable

GATT is using the Attribute Protocol (ATT) to save the attributes internally in a lookup table. Considering this fact searching an attribute is given in constant time. All attributes are retrieved by their universally unique identifier (UUID). Some UUIDs are used for unique attributes and are defined in the specification [37].

Example

For example, a device has a service to read the battery level, then the characteristic would have the battery identifier (0x180F [36]), the permission *Readable* and the value of the remaining battery life. In this example, the strength of the data structure can be seen. If the Master device wants to check particularly the battery life of its partner, the Peripheral just has to update the battery characteristic value and not sending it all the time or sending all the other Characteristic values, which are not required by the central at that specific moment.

The master is also able to limit its scan to devices with specific services. All these features are improvements to increase battery life.

Data exchange

Townsend [37] describes the two different approaches for exchanging data, which are defined in the case of the connection mode. The peripheral has to send out periodically an advertisement packet that consists of the available services. Further, a peripheral has to own at least one service and characteristic that a central can search for the specific service it and receive the characteristic value. Receiving a characteristic values is split into 2 approaches. First a central can send a request for a specific value to read out the value. The second possibility is kind of notification. The central sends a notification request for a specific characteristic. If the characteristic changes its value, the value is notified to the central, which is subscribed for this characteristic.

The value can also be written to by both contributors. The central has a write request, which can be with or without a response. This is dependable on the peripheral, which type of writing it is supporting [37].

Key Limitations

With the focus on saving battery life, other limitations arise with Bluetooth Low Energy. There is still a tradeoff between being very computationally efficient and power efficiency.

For implementing software for Bluetooth 4.0 and higher it is mandatory to know that one outgoing data packet is limited to 20 bytes of data. Theoretically 120 bytes can be sent per connection event and therefore about 0.125 Mbit/s can be reached. However, the central device can be connected to other devices and thus not capable of reaching this maximum rate. As a developer, one should bear in mind that the actual data throughput will be around 10 KB per second. As a result of small data packets low latency can be reached which is one target of the specification [37].

Another limitation is the operating range. For sending data wirelessly over a wide range, a high transmission power is required. Sending data is always very power consuming and the higher the transmission power, the higher the power consumption. Accordingly the specification of Bluetooth Low Energy

limits the transmission power to save battery life. Hence communication is restricted to short range. Hypothetically data can be transmitted in a range of 30 meters with line of sight. The functional range with a common indoor environment is about 2 to 10 meters [37].

4.2.3. Bluetooth And iOS: Core Bluetooth

Apple[3] is quite restrictive if it comes to connecting Apple hardware to hardware from other manufacturers, because this will break gaps in their closed environment and therefore they have no influence in the user experience. The latter is of great importance to Apple because the user experience is one of the leading arguments to buy Apple products. Before Bluetooth Low Energy was introduced, Apple only allowed to connect to devices through Bluetooth, which were registered in their "Made for iPod / Made for iPhone" (MfI) program. As a manufacturer you have to join this program for making it possible to connect Apple devices to your bluetooth device with some limitations, because not all Bluetooth services are accepted. For special details, Apple published a list [39], which is showing the supported profiles. This MfI program is subject to a charge and you have to sign some secrecy papers, so further details are not available.

With the new 4.x specification, Apple changed their mind because allowing this standard will not affect the user experience. This is based on the fact that the specification is built upon the low power usage and a small packet size, as mentioned previously. The small packet size ensures a minimal latency and as a consequence it leads to a fluid user interface (UI). Accordingly, a robust user experience can be reached. Apple introduced a Bluetooth framework called *CoreBluetooth*. This framework exposes an application programming interface (API) of the built-in bluetooth dongle to developers. Nearly every Bluetooth Low Energy functionality can be accessed, for example the iPhone can act as a central device, peripheral device or even both at the same time. Apple offers a guide for CoreBluetooth which is covering all implementation details and how to use the framework [40] efficiently. Nevertheless, the framework has some conceptual shortcomings in handling devices. Therefore, a new Bluetooth Manager will be introduced in Pocket Code for iOS, which will serve as an interface for all Bluetooth

[3] **http://www.apple.com**.

related communications. Details will be discussed in the Bluetooth Manager implementation covered in Section 5.1.

Part II.

Practical Part

5. Implementation details

The target of the integration of Bluetooth able Arduino boards into the iOS application of Pocket Code was encapsulating the feature and use the, as previously described, test driven development approach (Section 3) for the software interface as well as the hardware connection. Therefore, several steps have been executed to be able to communicate and control a connected Arduino board. First of all, the application did not support Bluetooth communication. Accordingly, the possibility to connect and communicate with a Bluetooth Smart able device had to be implemented. Furthermore, the interface for communication especially with an Arduino board had to be designed. Finally, the integration into Catty, Pocket Code for iOS, was achieved to be able to connect to the desired board within the application. Additional bricks were added to control the Arduino board with self made projects. In the next couple of sections, every distinct part is described in more detail, with comparison and references to other patterns, which were used in the implementation to reach a well designed structure. At some certain code places, it was hard to integrate this new feature because of the control flow of the application at this time. These problems and accompanying implemented solutions will be described later in Section 5.4.

5.1. Bluetooth Manager

The first part was to integrate Bluetooth into the existing application. The goal was the creation of an encapsulated Bluetooth manager, which is offering a simple application programming interface and is covering and managing all Apple CoreBluetooth functionality, starting with searching for devices or storing known devices to connect to a device.

The best way to achieve this encapsulated behaviour was to build a framework. The framework is called *Bluetooth Helper*. This helper is built upon the protocol based approach to be able to create Mock objects, which

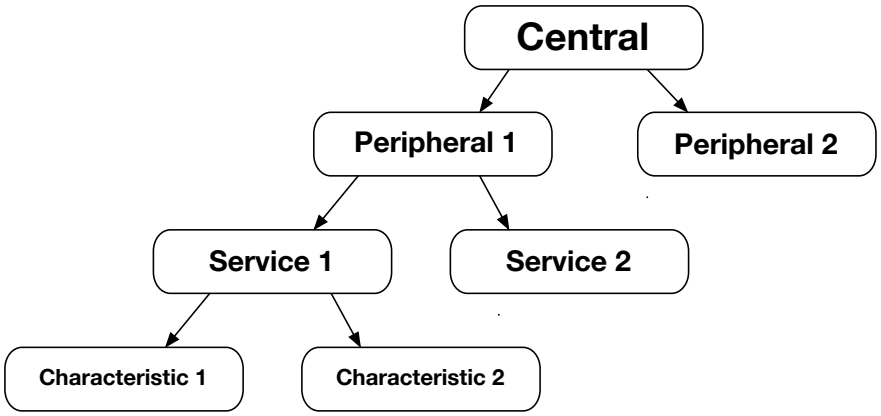

Figure 5.1.: Visualised structure of the *Bluetooth Helper* data model.

are very useful in the test driven development approach.

To be consistent with Apple's CoreBluetooth framework, the same types of classes were built. These classes cover the Bluetooth central, which is searching for devices and can be regarded as the master device in the communication, and the peripheral that is the communication partner. Furthermore, the peripheral owns services and characteristics. The shared central manager stores the scanned, known and connected peripherals, so that every single device is easily accessible. Each peripheral has an array of advertised services and those services are split into their provided characteristics, like the BLE specification demands. Looking at this structure, it is managed like a tree with the central manager at the root and the characteristics of each peripheral as leaves, visualised in Figure 5.1 .

Each class implements a protocol and contains a helper class for internal management to split the implementation and the API, to be in a position to have a quick overview of the provided features. Every function, which is available to the user of this framework, is defined in those protocols to make the use and maintainability easy. The implementation of the Bluetooth functions is not accessible for the user and therefore happening automatically, when using the provided protocol functions.

The Core Bluetooth framework is very powerful, but a straightforward API is missing. For example searching for devices just returns an instance

of the peripheral, but nothing about services or characteristics is known at this time. In the specific case of connecting to an Arduino board, it has to be assured that the peripheral has a characteristic to receive data from the smartphone and a second characteristic to send data to the smartphone, which are called RX and TX characteristics. If those characteristics are not provided, a connection would not be useful in our case, because of the need to exchange data bidirectional. Accordingly, the Bluetooth Helper framework wraps a few Core Bluetooth functionalities and provides the peripheral combined with its services and associated characteristics using a single function call (called *startScan*).

Another big advantage of using the Bluetooth Helper framework would be the connection process itself. Of course the Apple framework has a distinct call to connect to a specific device, but it only tries to connect to the device once and after a long time it just stops the connecting process. In this thesis a new connection process is introduced, which includes the option of retrying the connection phase without confusing the end user and it also introduces a fixed time for the connection timeout. For example, if the first message in the connection process is lost, then the smartphone tries to establish a connection again after a given timeout. This is quite convenient if located in a crowded place with a lot of Bluetooth devices around. Another useful feature would be the disconnected retry. If, for some reason, the connection is lost and it was not initiated by the user, the device tries to reconnect to the device once, without impact to the user experience. If this reconnection process is not successful the user will be notified, otherwise it has no effect on the user.

Finally the developer, who is using this framework, can easily decide, which action should be triggered for every single Bluetooth connection status. In the case of Pocket Code, this is a very essential feature. Considering a user is executing a project, which is using the Bluetooth connection, and the connection gets lost during executing a project. This project would not behave as it should, because for instance no sensor values are received from the disconnected Arduino board. Therefore, the user will be informed of the lost connecting during the project execution.

As mentioned previously, the framework was developed in Swift using protocols. This strategy was chosen to be able to create Mock objects for the test suite. As seen in Listing 5.1, it was easy to inject code to make the whole framework testable, even without real hardware devices. Accordingly, the

internal Bluetooth implementation can be automatically tested on the test server with this mocking approach. The following listing is showing a mock implementation of a peripheral with a fixed name and fictive services.

Listing 5.1: Bluetooth Helper Mock example.

```
class TestPeripheral : PeripheralWrapper {

    let helper = PeripheralHelper<TestPeripheral>()

    let state :CBPeripheralState
    let name : String

    let services : [TestService]

    init(name:String = "Mock Peripheral",
         state:CBPeripheralState = .Disconnected,
         services:[TestService]=[
         TestService(uuid:CBUUID(string:"..."),
                                 name:"Service Mock-1"),
         TestService(uuid:CBUUID(string:"----"),
                             name:"Service Mock-2")])
    {
                self.state = state
                self.name = name
                self.services = services
    }

    func connect() {
    }

    func reconnect() {
    }

    func cancel() {
        if self.state == .Disconnected {
            CentralQueue.async {
              self.helper.didDisconnectPeripheral(self)
```

```
            }
        }
    }

    func disconnect() {
    }

    func discoverServices(services:[CBUUID]?) {
    }

    func didDiscoverServices() {
    }

}
```

This framework is not automatically testing with actual hardware and the integrated Bluetooth dongle, because the iPhone simulator, which is running on the xCode server is not capable of Bluetooth interaction. There is another fact, why testing the Bluetooth dongle itself, is not necessary, because if the iPhone's Bluetooth dongle is not working properly, it is not able to be turned on in the device settings. That is why it is assured that the hardware is working properly when using the framework.

5.2. Arduino Connection Implementation

The Bluetooth able Arduino board is one type of Bluetooth device, which is now able to connect to the iPhone running Pocket Code. However, it is highly likely that other Bluetooth devices will follow. Therefore, the goal was keeping the integration of a new device into Pocket Code as simple and fast as possible.

The most suitable structure for this reason is inheritance. A parent class, called *BluetoothDevice*, has been introduced. This class is covering all the basic Bluetooth delegates and methods, which every BLE device requires. If a new device is added to Pocket Code for iOS, this new device class only has to be inferred by the mentioned parent Bluetooth device class and implement the special functionality, which is device specific. The standard

Bluetooth functionality is already covered and must not be considered.

For Arduino, our Catrobat team decided to use the *Standard Firmata* as described previously in Section 4.1. Since it can be possible that more devices will utilize this Firmata protocol in the future, a new class is introduced. This class is inherited from the root Bluetooth device class and covers all the Firmata specific functions, provided by the Firmata specification. Therefore, a further added Firmata device can inherit from this class and the Firmata functions are already implemented. Of course, the Arduino class is inherited from this class and the only thing to implement to properly add this new device, is to implement the protocol functions, which are covering setting and getting of analog/digital values. These functions are then used in Pocket Code to trigger diverse actions.

For the sake of consistency, an array is storing the actual values of the connected device to retrieve the state of the Arduino board at every certain time, without overloading the Bluetooth connection.

Very important to bear in mind is, that the Arduino board has to run the Standard Firmata as well. The Firmata code is integrated as an example in the Arduino editor. The board needs to be plugged in via USB to the PC or MAC and then the Standard Firmata has to be uploaded to the board to use it with Pocket Code afterwards. With some Arduino Bluetooth boards, this standard Firmata is not working, due to some different internal wiring. Therefore, these companies provide an own Bluetooth-Firmata code, which should be uploaded to the Arduino board instead of the Standard Firmata.

5.2.1. Firmata Implementation iOS

As indicated in the theoretical description of Firmata, it is a communication protocol to easily send instructions to the board to offer the ability to control it.

Unfortunately, the iOS version of this Firmata is out of date and so a new version has been developed using the programming language Swift during this thesis. This Firmata implementation is not entirely complete, but the most used functions are implemented. It was a little tricky to work with Swift and the required data modulations to receive the right instructions, because the strong typed features of Swift.

Firmata was implemented as an encapsulated component using the dele-

gate principle of iOS development. Every Firmata device has its own Firmata component, where it is registered as the delegate and it has to implement the delegate methods of the Firmata Delegate Protocol. This component can be seen as an en-/decoder of all the messages sent between the central device (iPhone) and the Arduino board.

Why is an en-/decoder needed? - First of all, as stated in the specification, each instruction has a unique prefix. This prefix needs to be added to the message. The second reason would be that in the CoreBluetooth specification of Apple only data, more precisely NSData, can be sent over the connection. Therefore the prefix combined with the message, could be a string, integer or something else. This has to be converted into this NSData format, before sending it to the Bluetooth device. Another factor would be that a message must not exceed 20 bytes of data, using Smart Bluetooth. If larger messages should be sent to the Arduino board, the splitting of the data has to be done correctly, so that the Arduino board knows when this message is finished. This also applies to the communication from the Arduino board to the iPhone. If the Arduino board sends a message, the Firmata component has to decode the message differently for each instruction type and then trigger the appropriate delegate function to pass the message to the internal Firmata device.

5.3. Integration in Catty

The most difficult task was to integrate the mentioned functionality into Pocket Code and do not overload the user with too many options and steps to use the feature. On the one hand, it should be very easy and straightforward to use this new functionality for the end user and on the other hand it should be easy to integrate other new Bluetooth devices into Catty with just a few lines of code.

The integration of this new feature should not change the user behaviour to control and use the application. Therefore, a way of minimal interaction with the user has been chosen to control the Bluetooth specific tasks. Only a few scenarios exist, where the end user has to apply some action to get the bluetooth communication working. Every time a project is started it is checked, which resources are required. If a connection with a Bluetooth device is needed to execute the project, then the user will be asked to select

a device, which is in range of the smartphone. For sake of convenience, previously connected devices will be shown first, because it is most probable that one device will be connected to very often. Accordingly, one does not have to think about connecting to the device in advance, because Pocket Code will notice if the iPhone is connected to a suitable device or not. After stopping the project the connection stays alive till the app is closed or the user force disconnects the device in the settings screen. This is a really handy feature for the user because restarting a program will not show the user the device selection process again.

Of course Bluetooth connection is not always perfectly reliable and even with the enhancement of reconnecting a suddenly disconnected device, there is a chance that the connection is completely lost, due to some environmental reasons (for example to large distance between the two communication partners). If this rarely case occurs, the user will be notified accordingly. For example, during an execution of a project that is using the Bluetooth feature, the execution will be stopped and the user will be asked to connect to a device, when starting the program again.

The other task was to make sure that the integration process of a new Bluetooth enabled device can be achieved without much effort. Since a device hierarchy exists, as shown in Figure 5.2, the new device has to be inherited from one of the existing devices according to the special needs. This created class has to implement the functions to control the device. Of course, new bricks have to be added to the data model and the script editor, so that the user has access to control these functions within the editing progress of projects. The final task would be to connect this device to the *BluetoothService* class, to be choosable if this device is required in a project.

First of all it has to be added to the requirements to be certain about that the new bricks are demanding this device to be connected. The next task would be to ensure that the device is inscribed in the *Bluetooth devices id* list. The connection function itself is only a function call with the peripheral device and the type of this device. Therefore, the concluding part to implement is to update the selection/connection process. This is very essential, because otherwise after selecting a Bluetooth device and calling the mentioned connection function the intern *BluetoothService* does not know how to behave with this kind of device type. In most of the cases of new

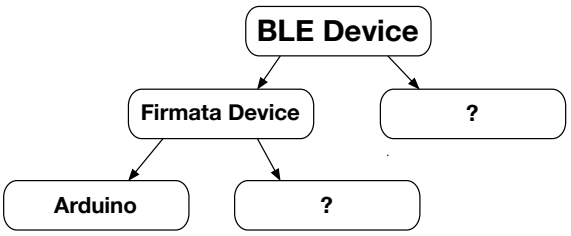

Figure 5.2.: Current Bluetooth device hierarchy in Pocket Code for iOS. The *?* are visualising that it can be extended easily.

devices, the implementation only consists of a new *switch*-case with the same instructions as the existing devices.

Summarising, the most difficult and time consuming part of adding a new device would be adding the bricks and providing a useful application programming interface for controlling the device by using the bricks. Everything concerning Bluetooth was intended to be straightforward.

In Figure 5.3 and Figure 5.4 the selection workflow is visualized. In the script editor, bricks that are requesting a Bluetooth connection are added. These bricks are of type *Arduino send digital* and are demanding an Arduino board to be connected to the smartphone. After triggering the play action, the selection/connection process will be initiated. As shown in Figure 5.4, the Bluetooth manager indicates which device is required to the user. Firstly, only the known devices are shown, but swiping to the right column called *Search* will reveal all connectable devices in the environment. After making a choice between them, the Bluetooth service is checking if the device advertises the required services. If everything discovered as expected, the connection process will be triggered. After a successful connection the project/program is started automatically.

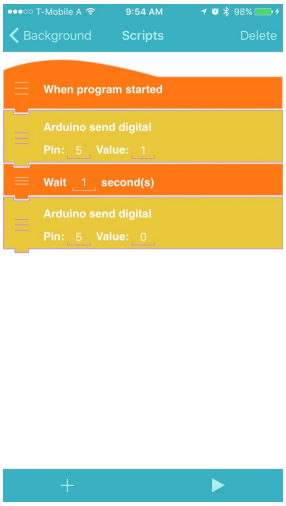

Figure 5.3.: Pocket Code for iOS script editor with Bluetooth bricks.

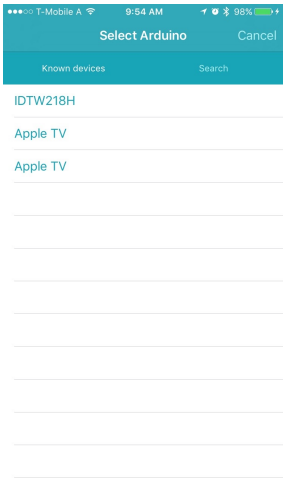

Figure 5.4.: Pocket Code for iOS Bluetooth device selection process.

5.4. Occurred Problems And Solutions

5.4.1. Bluetooth Sensors Blocking The User Interface

The first and severest problem that occurred in the integration of controlling the Arduino via Bluetooth with Pocket Code was about *threading*. To understand the problem, the player engine has to be described a bit in detail. It is based upon a commonly known strategy that is also used in compilers. The player engine consists of a front-end, back-end and scheduler. Furthermore, there is a broadcast handler, but this is not crucial for this specific problem.

When starting a program, the scripts are made available to the player engine. In the front-end, which is the first step, the bricks of each script are arranged to an abstract syntax tree, AST. With this hierarchical order, performance optimisations can be easily applied.

After a couple of optimisations the brick structure is taken over by the back-end. The back-end is then analysing each brick on his own and is creating a list (array) of executable instructions. In our case, these instructions can be SKActions or execution blocks. SKActions are part of the Apple gaming engine called Spritekit. The latter is offering the whole 2D gaming functionality, with automatic graphical updates and touching events. The SKActions have to be defined like an execution block, and then handed over to Spritekit at the desired time to run this peculiar action. Therefore, each brick, which needs a graphical update, is handled as a SKAction. After going through each script, we have several lists of these *executable* instructions. These lists are now handed over to the Pocket Code player engine own scheduler.

The scheduler gathers the bricks with the same instruction type, which should be executed at this specified time. For each type of instruction, another action will be initiated. SKActions are grouped together in an SKGroup and then transferred to Spritekit. After carrying out these actions, Spritekit calls the completion block, where the schedule method is triggered again. For own instructions, the execution blocks are executed without Spritekit and then the schedule mechanism also starts over again.

So where does the threading problem fits in this structure. For setting values to those bricks, the Catrobat language is using a type called *formulas*. These formulas can consist of numbers only, functions, variables, operators and sensor values. All types, except of sensor values, can be computed

instantaneously. Computing and interpretation of those formulas expect a return value in time, so that the execution of the brick instruction can be done immediately. Therefore, there is no time to wait for the value to be received. Sensor values of the device-own sensors can be read with just one getter method. Accordingly, this was no problem before integrating the Bluetooth service.

The core problem was waiting for the Bluetooth device sending the desired value. Since the smartphone has to send a request to the device and then receives the value, the lowest latency for one message would be about 10 milliseconds, but this time value is not reliable, because of different factors. It could also take a few tens of milliseconds as well. This time needs to be doubled to reach the waiting time to receive the sensor value.

Now two different solutions were composed. The first idea was to wait for the value and block the thread till the value is received. The second one was about asking every now and then for all sensor values and store them locally on the device. Of course, each approach has its pros and cons. In the end and after some tests, the decision was made to await the value to be able to work with the exact value at this time and not an outdated one. During the tests, the latency was quite on the lowest barrier, which also helped to decide between those two approaches. By choosing the first idea, the implementation consists of semaphores for the waiting process. That is why the thread is waiting for the semaphore to be signalled, after the sensor value is received.

In iOS programming, threads are handled with so called queues. Now The first problem arised, because the scheduler was running on the application main queue, which is also handling all the user interaction. Therefore, waiting on this queue would block the entire user interaction. This fact would be against the Apple guidelines and accordingly the solution was to run the scheduler on another queue with high priority. This helped to fix the blocking of the user interface. After some time of testing different programs, which were using Bluetooth sensors, two additional problems were recognised. The first and simpler one was the endless waiting on the semaphore if a Bluetooth message was lost and that is why the program stopped working. The simple solution was tantamount to add a timeout to the semaphore. This timeout triggers the signalling process of the semaphore if there was no answer in given time and the value zero will be returned. The second and more confusing one was that the user interface was blocked again. After

hours of debugging the reason for this behaviour was discovered. Spritekit is also using the same main application queue for executing the SKActions internally. This fact clarifies that under some conditions the SKActions are blocking the user interface again. Unfortunately, it is not feasible to dispatch these events to another queue. Since the Spritekit documentation is not that sufficient to specific questions or problems, an individual solution has been taken up. This solution is described in the following paragraph.

Since every brick is analysed and the according instructions are created in the back-end, this is the entry point of the provided solution. Every formula of a brick is checked and if it requires a Bluetooth connection. If a connection is needed, because the formula includes a Bluetooth sensor, then another Brick is inserted right before this certain brick. This newly inserted brick, which is *invisible* for the user, is not stored or has an influence on the user experience. It is just needed as a "helper brick". What is this special brick about? This brick has an own instruction called "FormulaBuffer", but holds a reference to the formula of the other brick that is requiring the Bluetooth device. If the scheduler wants to schedule such an instruction, the formula of the brick is computed on another standard priority queue, so that the user interface is not disabled. Now waiting on this queue is possible. This brick is always executed, no matter which instruction (SKAction or individual instruction) it is, because only the formula content is important. After the return value of the Bluetooth device is received, it is saved in a local formula own variable. Afterwards, the entire instruction is finished and the scheduler moves on. Nothing of the real brick instruction is executed till now. The next brick is the brick that is demanding the information that is stored in the formula own variable during executing the helper brick. The instruction is executed as there would not be any bluetooth sensor in it, because during evaluating the formula the local variable is checked, if it contains a buffered result or if it is empty. If it stores a value, then this one is returned and the local variable is set back to nil, if not then the formula is evaluated and we can be sure that no Bluetooth sensor value is required. Since every time the buffered value is read, it is set back to nil, it can be assured that the value is always refreshed properly and accordingly, only used once by the brick after the inserted formula buffer brick.

Of course, this value is not that up to date like evaluating it directly in the right brick, but the delay is acceptable and far better than overloading the Arduino board with scheduled requests in terms of efficiency, battery

life and up to dateness. By introducing this helper brick the user interface is not disabled anymore, no matter which instruction is called. However, the scheduler has to manage one brick more, which will lead to a slight slow down, but this is tolerable in comparison with a disabled user interface.

For reason of understanding, a simple example will visualise how this buffering is working. In Figure 5.5 on the left side one can see a *When Started*-Script with a *Wait*-Brick and a *Set Y to*-Brick. Both bricks contain a formula. In case of the Wait-Brick, the formula is just a number, in this case *1*, and therefore it can be evaluated immediately. The *Set Y to*-Brick holds a formula, which consists of a Bluetooth sensor value. Since this value cannot be evaluated in time and requires the requesting and receiving of the value via the Bluetooth connection. According to the previously presented solution a *Helper*-Brick is inserted right before the *Set Y to*-Brick, as presented in Figure 5.5 on the right side. So what does this *Helper*-Brick look like? In Figure 5.6 it is visualised that the *Helper*-Brick holds the reference of the same formula as the *Set Y to*-Brick, but has another instruction. Therefore, it is possible to store the pre-evaluated value in the Formula own "FormulaBuffer". The flowchart 5.7 is presenting the sequence, how the player engine scheduler is handling those bricks. The *Helper*-Brick is talking to the Bluetooth device on a different queue than the main queue. After the result has been received it is stored in the *Set Y to*-Brick formula buffer. Then the scheduler moves on, because the whole instruction of the *Helper*-Brick is finished. The next brick is the *Set Y to*-Brick. After checking the formula buffer, which is not empty, the SKAction is handed over to Spritekit with the buffered value and the formula own buffer gets cleared. After Spritekit has finished this action the next brick would be on the series.

5.4.2. Known Bluetooth Devices

Covering known Bluetooth devices, my first intention was to store the UUID, Universally Unique Identifier, of every Bluetooth device, which was connected to the smartphone through the Pocket Code application. Having this information, Apple's CoreBluetooth framework is providing the device instances for those UUIDs. These devices are shown to the user, if the Bluetooth device selection is presented. This should happen without searching for other devices. Tapping on one *known* device will start the

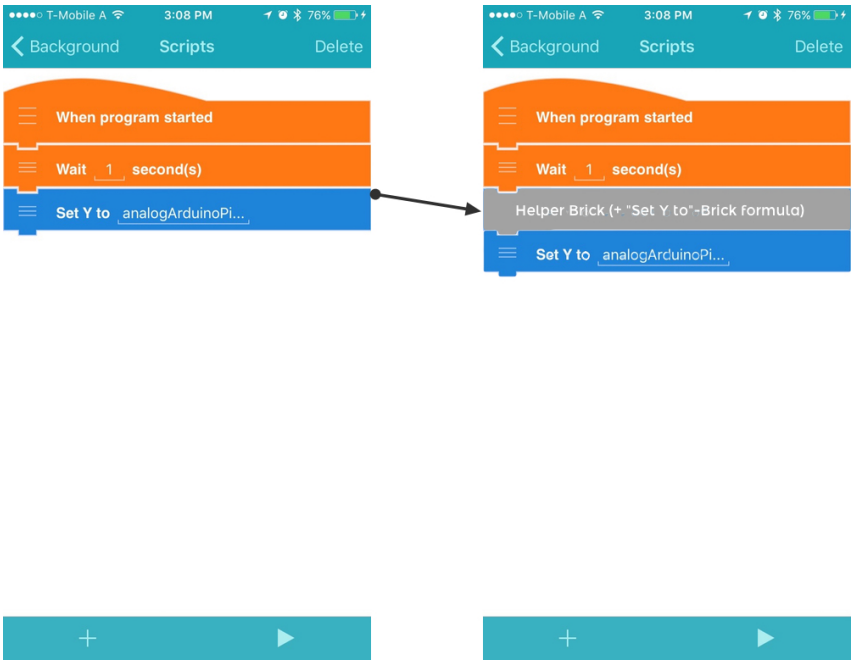

Figure 5.5.: Pocket Code for iOS Bluetooth device selection process - Script editor

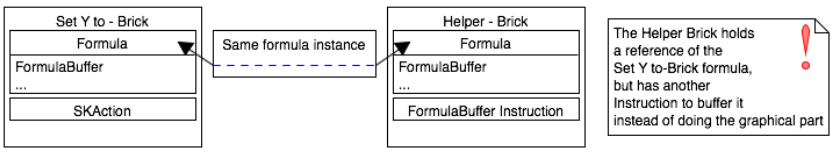

Figure 5.6.: Pocket Code for iOS Bluetooth device selection process - Data structure

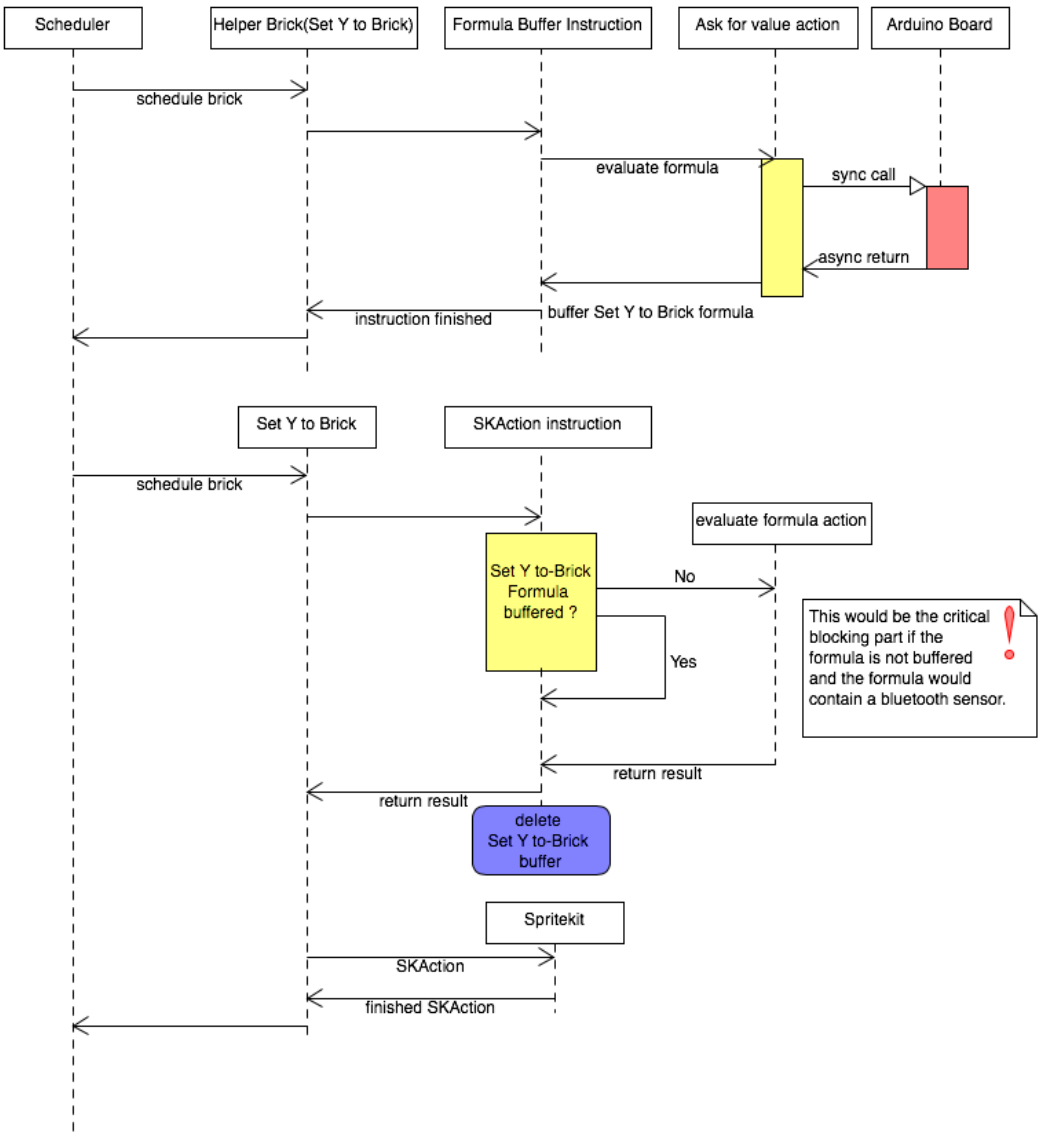

Figure 5.7.: Pocket Code for iOS Bluetooth device selection process - Flowchart.

connection process to this device and after a successful connection starts the project. No searching for devices means that the application is more battery efficient and of course faster than searching every time for all devices in the environment, since it is very likely that the end user will connect to only one and the same device very often.

Unfortunately this is not working as expected. Everything works fine until the connection process is triggered. Even if the device is in Bluetooth range it cannot be connected to. I also did not find a proper solution for this, therefore I implemented a little workaround, which will be addressed in the next paragraph.

The first few steps remain the same. The UUIDs are stored in the User-Defaults and the CoreBluetooth central manager is returning the devices for those UUIDs. Then the connection status of these devices will be controlled, because if one is already connected and it is the right type of device it will be chosen and the project starts. If all known devices are disconnected the selection view is shown with the known devices on the starting table view. Swiping to the left will start the scanning process and all devices in the environment are listed. However, as mentioned before it is highly probable that the user will choose one of the known devices. So there is a little workaround after choosing a device. Since just connecting to the device is unfortunately not possible, the central manager then searches for devices in the environment only with those services provided by this device. This method is more efficient than searching for all devices, according to Apple. If no device is found, the connection process is stopped and the user will be made aware that a connection is not possible. If one or more devices are found they are analysed. Since every device has a unique identifier, the devices are checked against the *known device* according this identifier. If the identifiers match then this device is exactly the *known device* and will be connected to like all other devices.

This workaround slows down the connection process of known devices a little bit, but it is more power efficient and more user friendly than do not provide known devices, because one does not have to scroll through the scanned devices list to find the right one. This list can be quite long in a crowded place, because many devices support Bluetooth nowadays and a

lot of people have this feature turned on all the time.

5.4.3. Resetting The Arduino Board

When working with Arduino boards, the pin states are always preserved till the Arduino board is resetted, which can be reached by clicking the reset button on the device or disconnecting the power outlet. After the new boot, all pins are initially set to 0. In our case, we are working remotely with the Arduino board and restarting a program in Pocket Code will not reset the Arduino board in general. This could result in different behaviour of the executed programs. A simple case would be that some pins are "working" together to trigger an action to a connected electronic curcuit and then this could lead to trigger it at a wrong time, because for example a digital pin was set to 1 in another program before.

Of course, a simple solution would be to advise the user to reset the Arduino board after each restarting of a program in Pocket Code, but this will lead to a complete reboot and therefore the Bluetooth connection will be lost. Accordingly, this does not constitute an optimal solution. A second solution would be that the user has to take care of resetting the used pins at the beginning of each program in a *Start script*. However, this is not a clean solution, because it is likely that users forget to "program" the resetting process in Pocket Code.

The approach in Pocket Code for iOS for solving this issue is to reset the board after leaving the scene. This could be the case by resetting or quitting the program. If one of these two actions is triggered by the user, the Bluetooth manager runs a resetting procedure. The first intuition was just to set all pins of the Arduino board to zero, but this revealed another error. For some boards this canceled the Bluetooth connection. After some debugging the cause of this behaviour was unveiled. By setting all pins to zero it could be the case that one pin, which is responsible for the Bluetooth dongle, was set to zero in an inappropriate time and that is why the connection was canceled. The solution for this issue is to analyse each pin to see if the setting mode is allowed. Every pin, where the setting mode is allowed, is now set to zero. After this procedure the Arduino board is reset. This means, all settable pins are on state LOW (zero), and the user triggered action will

continue. This slowdown is nearly not recognisable for the user.

There is only one problem with cheap versions of Arduino boards, which have this Bluetooth pin also made available. But combining this fact that there are only a small percentage of these boards and the situation where this pin is currently talking to the Bluetooth dongle, it is very unlikely that the connection will be lost. If it is the case the user just has to select the device of the known devices in the selection view and it will be connected again.

Another case where this approach is not working as expected would be if the connection is lost during executing a program in Pocket Code. If there is no connection, then the mentioned resetting procedure cannot be carried out, because the connection is essential for it. That is why the user is informed that the connection is lost and for a clean start the user should reset the Arduino board by clicking the reset button on the board. There would be one solution to run the reset procedure right before starting a program, but this would lead to a delay of every start. Since this not desired disconnection is a rare case the slowdown is not appropriate. Unfortunately there is no better solution to tackle this certain problem.

5.4.4. Arduino Pins, Pin Modes & Pin Naming

Since many different types of Arduino boards with different specifications exist, the implementation to support those devices should be very general and open. As mentioned previously in the Arduino hardware section, each Arduino board has digital and analog pins. The digital pins are split into digital (I/O) only and PWM, pulse width modulation. Every type of Arduino board has a different number of those pins, so it cannot be assumed that the board has the standard number of 5 analog pins. The Android version of Pocket Code is just supporting the standard configuration of a common Arduino board. When integrating Arduino in the iOS version, thoughts were given to being more flexible. Therefore, the iOS version is able to fully support different types of Arduino boards. But the problem was how to achieve this flexibility.

This is possible because of the use of the Standard Firmata own *PinState Query* and *Analog mapping query*. After connecting to a device these queries are forwarded to the Arduino board. First the answer is given to the analog

mapping query. It tells the smartphone how many pins the Arduino board consists of and which ones of type ”Analog” are. After this is analysed, the pin state query or capability query is transmitted to the Arduino board. The answer is very complex, but it includes all the information needed to support all pins with the matching pin modes. This answer contains each pin and the matching pin modes. After all pin modes of a pin are read, there is a 127 byte to signal the end of the first pin. This is repeated till all modes of all pins are known. This parsed and encoded information is then stored in an array of dictionaries, where each pin has its own dictionary, filled with the pin modes that are supported by this specific pin. Now the smartphone has complete knowledge about the Arduino board.

If the user wants to send a PWM value to a pin, then right before sending it, Pocket Code is checking if this pin exists and if it has the availability to process PWM values. If it supports this mode, the message will be transmitted, if not it will be ignored. This is working for all pins and therefore, it is guaranteed that pins do not receive wrong commands.

Unfortunately there is the same problem with cheap Arduino versions as mentioned in the section on resetting the board. It could be the case that sometimes the pin state query is not working with those Arduino boards. If this is the case, we pretend that this current connected board is a standard Arduino board, like the Android version does.

Another problem was detected, which is about the Arduino pin naming. On some devices, the analog pins are starting to count at zero again. So there are digital pins with the preface ”D” and analog pins with the preface ”A”. Other boards are counting on, accordingly A0 could be the same as A14 for example. So what is the user supposed to type in Pocket Code. In the iOS version, we are handling the analog pins separated from the digital ones. Therefore we start counting at zero. However, we cannot say that the user has knowledge about that or typing it as we suggest to. That is why Pocket Code for iOS is converting the user input of analog pins if there is a need to. Some cases exist, which will be covered now:

The simplest case would be if the user wants to set an analog Arduino pin with a number lower or equal than the existing number of pins. Then this input must not be converted. For example, if there are five analog pins and the user types in number three everything is ok.

The second case would look like if the user types in a number which exceeds the total number of pins. Then the input will not be taken into

account.

The third and most interesting case would be if the user types in a number which exceeds the number of analog pins but is lower or equal than the total amount of pins. We know the total count of the pins and the number of true analog and true digital pins. Therefore, we are just subtracting the number of digital pins of the input to gain the right analog pin. This method could lead to another case where subtracting leads to a number smaller than zero. Then the user has typed in an illegal input, because the input exceeds the number of analog pins, but is smaller than the total amount of pins and smaller than the number of digital pins. Accordingly the user wants to send an analog value to a digital pin, which is not compatible.

With this conversion, it can be guaranteed that both inputs methods of the user will be accepted and interpreted in the right way. An extension could be to tell the user that there was a wrong input, so that errors in programs can be recognised more easily. However, this feature is not implemented by now.

Part III.

Conclusion

6. Conclusion & Outlook

6.1. Test Driven Development In Usage

To summarise, testing will help to improve the product quality, but there is no way to deliver error free software due to the infinite number of different inputs and extremely large number of logical ways. Everything cannot be tested in reasonable time and without spending too much money. So generally said, there is always a chance that the user will find an error in the product. Testing can be seen as decreasing the chance that a user will bump into an error.

No general way exists, which will lead to a good tested product, because every project is special on its own and has other core parts that should be tested in more detail than other parts. Different approaches exist to tackle this certain problem and they evolved over the time, but it cannot generally be testified that one is better than the other. Every approach gets its pros and cons.

To successfully write unit tests for a whole product, it must be assured to write code that is designed for tests. Most of the design will grow as it should be if development is done in a test first way. Overall it is very important to design an interface instead of the underneath implementation, because the interface will be used and the concrete functionality should be working (Tests will check the functionality). Owing to this fact, it will be possible to replace dependencies by fake objects and to be able to test the "whole" project. Another advice would be to keep the methods small and focused on one precise functionality. If a test fails, it should not be difficult to locate the error, because this is why the tests are written. If a method would cover more than one task, then the developer has to debug again with just a little time saving.

In my opinion, the test driven development is a very straight forward method with excellent results if everyone follows the principles. It is also

covering the mentioned aspects covered in the previous paragraphs. However, in my point of view it is hard to integrate this workflow into an existing project by cause of the "untested" code base. While implementing one functionality after the other, sometimes the point of testable design is completely left beside. To reach a good code coverage, the existing code has to be tested afterwards, which is very time consuming and can lead to a lot of refactoring. I noticed this once, when working on a project and then tried to adapt to the test driven development cycle. It was very exhausting and therefore a hybrid workflow had been created for this project.

Such a combined workflow can be a good solution as well, but it has to fit exactly the products needs, therefore one does not have a single workflow for different projects, which can lead to a confusion.

Talking about test driven development, it is not just about the three main principles:

- Test
- Code
- Refactor / Design

These three steps would be easy to adapt to, but the time consuming part is the iteration of these steps when implementing just one requirement. Following the cycle strictly, one warning or one error in the test will lead to one iteration. Accordingly, for one test, dozen of iterations is quite normal. Of course, the cycle and the thought behind the test driven development are just guidelines, but the question arises if an amended cycle will lead to the same results. Personally, I think one should see this strategy as a rush lead, but it would be perfectly decent to write the whole test, then implement the functionality to pass the test and afterwards refactor without failing the test again. The refactoring is of course an iterative phase, which is fine, because the iteration is just running the test cases to see what happens.

From my point of view, after some time of working with test driven development, it is my favourite approach to tackle the testing issue. Before using this strategy, I caught myself after finishing a requirement thinking about,"Oh it works, so why should I write a test now? ". I guess, I'm not the only developer with this thought in mind, that is why I think a lot of test are not written, because of similar thoughts of certainty that it works at

this time or of course laziness to write lines of code, which are not adding some new functionality. Writing tests afterwards, no matter how much you believe it will work, it does not. Following the test driven development cycle, no thoughts are given and writing the tests are part of the implemented feature. Do not stop following this principle throughout the project, even if the feature is that small and easy, but it will have a high chance to fail the whole project. Testing is part of the progress, so there should be enough time for doing it, do not forget about it even if you have time constraints.

In my opinion, this approach with its core principles should be the preferred one to teach, when someone is starting to learn to program. The following paragraphs summarise some of the previously mentioned advantages: (3.3)

At first sight using Test Driven Development seems really annoying and just a waste of time. For the sake of convenience a small method is written and tested with the previously mentioned trial and error tactic. Nevertheless, it could be the case that this method is growing over time and different persons are extending or refactoring it. After some changes, it is not guaranteed that the initial functionality is furthermore given, because only the added use case is tested in this *Code and Fix* model. This is precisely the point where the test driven development strategy is showing it's point of perpetually using it.

Not only this fact but also being sure that one mistake cannot happen again, because of the tests in the background, should be enough conviction to use this model or a fitted version for the own developing workflow.

Personally, what really convinced me was that there is always a working version available. I worked on different projects and the most annoying thing would be if one wants to add a feature and checks out the current base and this one is not working. So other people have to be contacted or one has to fix those errors first, which is really time consuming and therefore decreasing motivation.

Test driven development is only covering the incessantly correctness if the requirements are implemented correctly, but it is not meant to show that it really behaves as the customer wishes. Accordingly, acceptance test driven development should be used. With this strategy, it could be possible that tests and methods have to be rewritten, which is resulting in a very large time investment. To tackle this problem one should have at least a general overview of the project, as Graham [7] said. If the test driven development

cycle is stiffly applied, the project could end up in a miscellany of small methods, which are tested, but do not satisfy the global need.

With this fact in mind, test driven development can help to improve the development workflow and direct the project to a good code base and less error prone, but it should be utilised intelligently. Wisely use was only noticed on the edge but in my opinion this is the core fact to succeed in using test driven development.

Before joining the Catrobat project, I just heard about the basic details of test driven development, but it was never applied in one of the courses with real examples. After reading through the code base and starting to develop with some help of senior members it was quite challenging for me to write the tests first. The rethinking of the own development workflow takes some time, because writing tests with errors and warnings that something is not available is very confusing and difficult at the beginning. However, after some time, it really helps to think about the interfaces of each method and how it can be improved before coding it and then rewrite the code again. The train of thought is much more precise and focused.

6.1.1. iOS Test Driven Development

In some parts of the iOS's development, it is very hard to integrate the test driven development cycle. This applies to tightly coupled View and ViewController handling in the iOS development. Some Views are built with the internal *Interface Builder*. One can add subviews, buttons, etc. set their position with constraints and a lot more. However, this is not really testable. Since iOS 9, there is a feature to test the user interface (UI). The screen can be record and clicked on to find out if there is a button and which action is called. Unfortunately, this feature is not fully reliable till now. I hope that it will be improved in the future.

But not only the UI is hard to test, when talking about iOS development. Since Apple is providing a lot of frameworks, which are used frequently, this is why so many delegate functions exist, which are really hard to change into a testable design. Mocks are needed for each framework class like mentioned in the implementation of the Bluetooth Manager in section 5.1. Apparently this dependency is enforced by the principle of frameworks

design but should be converted to a testable design.

Generally, there are some places in the iOS design, where test driven development cannot be used out of the box. Framework delegates can be designed and tested with Mock objects, which are a lot of work, but it would be possible. Test first design cannot be applied to everything covering the UI and UI related parts. So another approach needs to be practiced then. In my special case I did fall back to the *Code and Fix*-model for UI parts and then tried to add user interface test, but I was not able to cover all user interface functionality because of the lack provided UI test features.

6.2. Future Work

6.2.1. Support other Bluetooth devices

The Pocket Code for iOS application will be extended throughout the years. New bricks, scripts and features will be added because smartphones will change and provide more and more sensors or other useful elements. Since this thesis is not covering the application itself, but only the special part concerning the Bluetooth section of the future work will be to catch up with the Android team and support other Bluetooth devices or in general wireless connected devices as well.

As mentioned in Section 5, the Bluetooth manager has a fixed application programming interface to easily integrate and support other Bluetooth devices. Therefore, it will be straightforward to integrate other BLE devices into Pocket Code for iOS.

The only thing to do is tantamount to add it to the "Selection Manager" and implement the actions of the device itself. Everything else is already implemented by the Bluetooth service.

Concerning other Wireless device, for example over WiFi, one can take the Bluetooth Manager as a lead and implement an *Wifi manager*. This Wifi manager would be very interesting for adding a drone to Pocket Code for iOS.

6.2.2. Further Arduino-Firmata extensions

Additionally to this thesis, which covered implementing a Bluetooth manager and the standard Arduino Firmata integration into PocketCode for iOS, an integration of further commands can be implemented. In this thesis only *set digital* and *set PWM* commands were covered. Firmata is providing several more commands to for example control servo or stepper motors. These commands can be used to create new bricks in Pocket Code and to be added to the own iOS Firmata implementation.

Appendix

Bibliography

[1] Catrobat, *Catrobat*, 2015. [Online]. Available: http://www.catrobat.org (visited on 11/21/2015) (cit. on p. 5).

[2] MIT Media Lab Longlife Kindergarten, *Scratch*, 2015. [Online]. Available: https://scratch.mit.edu (cit. on p. 5).

[3] M. U. Bers and M. Resnick, *The Official ScratchJr Book*. 2005, ISBN: 9781593276713 (cit. on p. 5).

[4] MIT, *Mit media lab longlife kindergarten*. [Online]. Available: https://llk.media.mit.edu (cit. on p. 5).

[5] Catrobat, *Pocketcode*. [Online]. Available: www.pocketcode.org (visited on 11/21/2015) (cit. on p. 5).

[6] G. J. Myers, C. Sandler, and T. Badgett, *The Art of Software Testing*. 2004, p. 200, ISBN: 9781118133156. DOI: 10.1002/stvr.322 (cit. on pp. 11, 16, 17).

[7] L. Graham, *Test Driven iOS Development*. Addison-Wesley, 2012, ISBN: 9780672329166 (cit. on pp. 12–17, 28, 30–35, 41, 91).

[8] B. Meyer, "Seven principles of software testing," *SoftwareTechnologies*, no. August, pp. 99–101, 2008. DOI: 10.1109/MC.2008.306 (cit. on pp. 12, 18).

[9] D. W. W. Royce, "Managing the development of large software systems," *Ieee Wescon*, no. August, p. 2, 1970 (cit. on p. 14).

[10] R. Osherove, *The art of Unit Testing*. Manning Publications Co., 2014, p. 294, ISBN: 1861891059 9781861891051. DOI: 10.1038/nchem.1109 (cit. on pp. 18, 20–23, 25, 26, 35).

[11] L. Koskela, *Effective Unit Testing - A Guide for Java Developers*. 2013, vol. 2001, p. 249, ISBN: 9781935182573. DOI: 10.1145/358974.358976 (cit. on pp. 19, 20).

[12] J. Reid, *Dependency injection*. [Online]. Available: https://www.objc.io/issues/15-testing/dependency-injection/ (visited on 12/02/2015) (cit. on pp. 23, 24).

[13] M. Lazer Walker, *Test doubles: mocks, stubs and more*. [Online]. Available: https://www.objc.io/issues/15-testing/mocking-stubbing/ (visited on 12/02/2015) (cit. on pp. 24, 25).

[14] L. Koskela, *Test Driven: Practical Tdd and Acceptance Tdd for Java Developers*. 2007, p. 470, ISBN: 1932394850 (cit. on pp. 26–30, 32–34, 37–39).

[15] K. Beck, *Extreme Programming Explained: Embrace Change*, c. 1999, p. 224, ISBN: 0201616416. DOI: 10.1136/adc.2005.076794. arXiv: 0201616416 (cit. on pp. 27, 31, 32).

[16] ——, *Test-Driven Development 'by Example'*. Addison-Wesley, 2003, ISBN: 0321146530 (cit. on pp. 27, 33, 34).

[17] N. N. Bhat, Thirumalesh, "Evaluating the efficacy of test-driven development : industrial case studies," *Isese'06,*, pp. 356–363, 2006. DOI: 10.1145/1159733.1159787 (cit. on p. 38).

[18] D. S. Janzen, "Software architecture improvement through test-driven development," *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications - OOPSLA '05*, p. 222, 2005. DOI: 10.1145/1094855.1094945 (cit. on pp. 38, 39).

[19] S. W. Ambler, *Introduction to test driven development (tdd)*, 2013. [Online]. Available: http://agiledata.org/essays/tdd.html (visited on 11/30/2015) (cit. on pp. 39–41).

[20] L. Salibra, *A brief history of ios beta testing*, 2015. [Online]. Available: https://blog.pay4bugs.com/2015/02/04/a-brief-history-of-ios-beta-testing/ (visited on 02/26/2016) (cit. on p. 41).

[21] A. Schroppe and D. Eggert, *Real-world testing with xctest*. [Online]. Available: https://www.objc.io/issues/15-testing/xctest/ (visited on 12/02/2015) (cit. on pp. 41–44).

[22] Apple Inc., *Testing with xcode*. [Online]. Available: https://developer.apple.com/library/ios/documentation/DeveloperTools/Conceptual/testing%7B%5C_%7Dwith%7B%5C_%7Dxcode/chapters/03-testing%7B%5C_%7Dbasics.html (visited on 02/26/2016) (cit. on pp. 42–44).

[23] C. Amariei, *Arduino Development Cookbook*. Packt Publishing, 2015, pp. 1–2, ISBN: 9781783982943 (cit. on p. 49).

[24] D. Cuartielles, G. Martino, T. Igoe, and D. Mellis, *Arduino.cc*, 2015. [Online]. Available: www.arduino.cc (visited on 11/23/2015) (cit. on pp. 49–51).

[25] Techtopia, *Microcontroller definition*. [Online]. Available: https://www.techopedia.com/definition/3641/microcontroller (visited on 03/16/2016) (cit. on p. 50).

[26] A. G. Smith, *Introduction to Arduino*. 2011, p. 1, ISBN: 9781463698348 (cit. on p. 50).

[27] D. Kushner, "The making of arduino how fi ve friends engineered a small circuit board that ' s taking the diy world by storm," pp. 1–6, 2011 (cit. on pp. 50, 51).

[28] H.-c. Steiner, "Firmata: towards making microcontrollers act like extensions of the computer," *New Interfaces for Musical Expression*, pp. 125–130, 2009. [Online]. Available: http://archive.notam02.no/arkiv/proceedings/NIME2009/nime2009/pdf/author/nm090182.pdf (cit. on pp. 52, 53).

[29] D. Mellis, J. Gautier, P. Stoffregen, F. Gulotta, and J. Hoefs, *Firmata protocol*. [Online]. Available: https://github.com/firmata/protocol (visited on 11/25/2015) (cit. on pp. 52, 53).

[30] Arduino, *Firmata*, 2013. [Online]. Available: www.firmata.org (visited on 11/23/2015) (cit. on p. 52).

[31] D. Mellis, J. Gautier, P. Stoffregen, F. Gulotta, and J. Hoefs, *Firmata arduino*. [Online]. Available: https://github.com/firmata/arduino (visited on 11/24/2015) (cit. on p. 54).

[32] I. Bluetooth SIG, *Bluetooth*. [Online]. Available: http://www.bluetooth.com/what-is-bluetooth-technology/bluetooth (visited on 11/25/2015) (cit. on p. 54).

[33] ——, *Bluetooth special interest group*. [Online]. Available: https://www.bluetooth.com (visited on 03/16/2016) (cit. on p. 54).

[34] ——, *History of the bluetooth special interest group*. [Online]. Available: http://www.bluetooth.com/Pages/History-of-Bluetooth.aspx (visited on 03/02/2016) (cit. on p. 54).

[35] ——, *Bluetooth le specification*, 2015. [Online]. Available: https://www.bluetooth.org/en-us/specification/adopted-specifications (visited on 11/25/2015) (cit. on p. 55).

[36] BluetoothDeveloperPortal, *Gatt specifications*. [Online]. Available: https://developer.bluetooth.org/gatt/services/Pages/ServicesHome.aspx (visited on 11/25/2015) (cit. on pp. 55, 59).

[37] K. Townsend, C. Cufi, and R. Davidson, *Getting Started with Bluetooth Low Energy: Tools and Techniques for Low-Power Networking*. O'Reilly Media, Inc, 2014, p. 180, ISBN: 9781491949511 (cit. on pp. 55–61).

[38] I. Bluetooth SIG, *Bluetooth technology basics*. [Online]. Available: http://www.bluetooth.com/what-is-bluetooth-technology/bluetooth-technology-basics (visited on 01/01/2011) (cit. on p. 56).

[39] Apple Inc., *Ios: unterstützte bluetooth-profile*, 2013. [Online]. Available: https://support.apple.com/de-de/HT204387 (visited on 12/20/2015) (cit. on p. 61).

[40] ——, "Core bluetooth programming guide," 2013, [Online]. Available: https://developer.apple.com/library/ios/documentation/NetworkingIntern
Conceptual/CoreBluetooth%7B%5C_%7Dconcepts/AboutCoreBluetooth/Introduction.html (cit. on p. 61).

Appendix A.

Acronyms

- **TDD** Test-driven development
- **UI** User interface
- **XP** Extreme Programming
- **CI** Continuous integration
- **BLE** Bluetooth Low Energy
- **PWM** Puls width modulation
- **I/O** Input/Output
- **UUID** Universally Unique Identifier
- **API** Application programming interface
- **MfI** “Made for iPod” / ”Made for iPhone”
- **GATT** Generic Attribute Profile
- **GAP** Generic Access profile
- **IoT** Internet of Things
- **IDE** Integrated development environment