Miloš Pokrievka, BSc

Platform Independent Hardware Tests with Behaviour Driven Development

Master's Thesis
to achieve the university degree of
Master of Science
Master's degree programme: Computer Science

submitted to

Graz University of Technology

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

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

Graz, July 2019

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.

Date	Signature

Abstract

Catrobat is a visual programming language designed to create programs solely using mobile devices. It is developed for multiple platforms. In addition to other features, it allows users to use device sensor values when writing programs.

This thesis deals with creating a system for automatically testing these hardware features independently of the platform. Measurements of several mobile device sensors are directly dependent on its position and inclination. Therefore a robot arm was constructed to enable moving and rotating the mobile device during automated tests. As platform independent tests serve feature files of the behavior driven development tool called Cucumber. For this purpose Cucumber was integrated into the Catrobat integrated development environment for Android called Pocket Code.

The robot arm is capable of partially rotating on all three axes. This enables to change positions and inclinations and manipulate acceleration of the mobile device enough to achieve a sufficient change of mobile device sensor measurements for practical testing. Cucumber feature files are written in the plain text language Gherkin, which was used to abstract the tests into a platform independent form. When using Cucumber with the testing framework currently used in Pocket Code several problems with compatibility emerged. The results of using Cucumber did not completely meet all of the expectations. However, the created robot arm with its simple control mechanism is an invaluable tool which enables developers to incorporate automated hardware tests in the test suites. In that way any app functionality relying on sensor values in relation to location and movement of the mobile device in space can now be easily tested.

Contents

List of Figures

List of Listings

1. Introduction

The high demand for high quality and fast software projects drives the constant changing and upgrading of modern software development practices. A lot of software development approaches have emerged over the past 70 years: from structured programming in the 1950s and the waterfall model in the 1960s to agile methods including test-driven development. The goal of all these methods is to create a product that will bring value to the customer. Agile methods developed in recent decades have become very popular in software development. Automatic tests covering program code guarantee software quality in agile methodologies. One of the newer agile methods is behavior driven development. Behavior driven development, like its predecessor, test driven development, is a software design method that when properly applied also guarantees coverage of all written code by executable specifications that can be used to test this code.

Catrobat is a visual programming language inspired by Scratch. In a block-based environment called Pocket Code, developed for mobile devices, Catrobat programs can be created by properly arranging visual elements, called blocks, instead of writing program code. Designed for use on mobile devices, Catrobat also includes the use of features unique to mobile devices, such as various sensors and built-in cameras.

Pocket Code is developed by agile methods with an emphasis on test driven development, leading to the need for full program test coverage. This also includes the program code used to communicate with the sensors of the device on which Pocket Code is running. However, sensor values depend on external factors such as the device position.

The goal of this thesis is to create a system to allow testing of Pocket Code parts that work with external factors, measured by device sensors and introducing behavior driven development into Pocket Code to make these tests platform independent. The following chapters present the concept of block-based programming, the Catrobat language, and offer a description of basic behavior driven development concepts. In the next section, a robot arm is introduced, designed to enable the execution of sensor tests. Finally, the control and communication between the robot arm and the Pocket Code are described along with new sensor tests created as a proof of concept.

2. Visual Programming Language Catrobat

The idea of visual programming is based on using visual elements with certain colors and shapes, each of which represents a different functionality and category a block belongs to. Covering the program code with graphical elements eliminates the need to learn the syntax of the underlying programming language. It is sufficient to understand how to connect and organize the building blocks to achieve the desired program functionality. It is similar to combining Lego® bricks where their shapes indicate whether the blocks fit together. In this way visual languages obtain their simplicity and intuitiveness. This gain comes along with the cost of losing some versatility, since there is a limited amount of blocks and categories to choose from. Nevertheless, this visual approach makes them a very good tool for teaching programming to beginners, especially children (Slany, 2012b).

One such visual programming language is Catrobat, it is inspired by Scratch and developed to be used to directly program on mobile devices. By using building blocks called *bricks* which are combined to scripts, games, interactive animations, and other programs that can be created easily on the mobile device and can be executed on Android and iOS. This chapter describes the Catrobat project and its interpreted programming language, which is used in later chapters.

2.1. Block-Based Programming

Wah (2007) defines visual programming languages (VPL) as languages where a significant part of the program structure is represented in pictorial notation (for example, as icons, connecting lines representing relationships, or colors). While text may appear in such a notation, it has a secondary role, such as naming program entities. Burnett et al. (1995) list four properties of

VPLs:

- Fewer concepts needed in the program (like pointers, declarations, scope, or storage allocation).
- Concrete programming process.
- Explicit depiction of relationships.
- Immediate visual feedback.

These features make VPL often more beginner friendly than textual programming languages which in turn is interesting for educational purposes. One of the oldest educational programming languages is Logo, offering turtle graphics, visual representation of output from user commands. Even with its friendliness to novices, the use of textual code led to difficulties for some learners such as the difficulty to understand pointers. This has led to the creation of a new variety of VPL called block-based programming that takes on a number of characteristics from the Logo such as using onscreen avatars and the egocentric commands they perform (Kurihara et al., 2015).

In block-based programming environments, programs are divided into blocks representing small sets of instructions that can be combined into full programs. The color and shape of blocks, or other visual cues, are used to indicate valid connections and explain their individual functions.

Repenning (2017) specifies the minimum set of affordances that VPL must provide to be considered block-based:

- The end-user must be able to compose blocks into programs via simple manipulation like drag-and-drop. A mechanism must be present to ensure the composition is syntactically correct. (For example context aware menus or block shape/color).
- Blocks as interactive objects, contain end-user editable information.
- Blocks can be nested to represent tree structures (For example loop blocks containing more blocks).
- Blocks are arranged geometrically to define syntax(block geometry) without the use of additional explicit graphical connectors (block topology).

Weintrop and Wilensky (2015) point out that block-based programming combines two branches of research on how to scaffold novice programmers: directly manipulatable interfaces that visualize concepts and objects, and structured editors. For textual languages, structured editors provide help by using information on programming language grammar to offer code-complete suggestions, syntax highlighting, and real-time compilation checking. In block-based programming environments, grammar is built into individual blocks through their properties and appearance (such as shape or color).

Bau et al. (2017) link the learnability of block-based programming languages with six learning barriers identified by Ko, Myers, and Aung (2004). They believe that this learnability is the result of how these languages address usability challenges based on three of these barriers resulting from the difficulty of compiling the program:

- Blocks are based on recognition and thus circumvent the problem of learning programming vocabulary.
- Blocks reduce cognitive load to new programmer by chunking code into fewer elements.
- Blocks prevent basic errors by providing constrained direct manipulation of the structure.

In recent years, the global increase in interest in computer science has led to the development of many block-based programming environments such as Alice and Scratch on personal computers or MIT App Inventor and Pocket Code on mobile devices.

2.2. Scratch, an Example of a Visual Language

Scratch is a project developed by the Lifelong Kindergarten Group at the Massachusetts Institute of Technology (MIT) Media Lab under Mitchel Resnick. It is a graphical environment and a visual programming language based on the constructionist ideas of Seymour Papert, the co-inventor of the Logo programming language. The primary goal of Scratch is teaching younger users in programming. That is why it was designed with the emphasis on simplicity and intuitiveness. It was released in 2007.

It allows users to easily create programs and animations "by snapping" together command blocks that control 2D graphical objects called *sprites*. These sprites move on a background called *stage* (See figure 2.1).

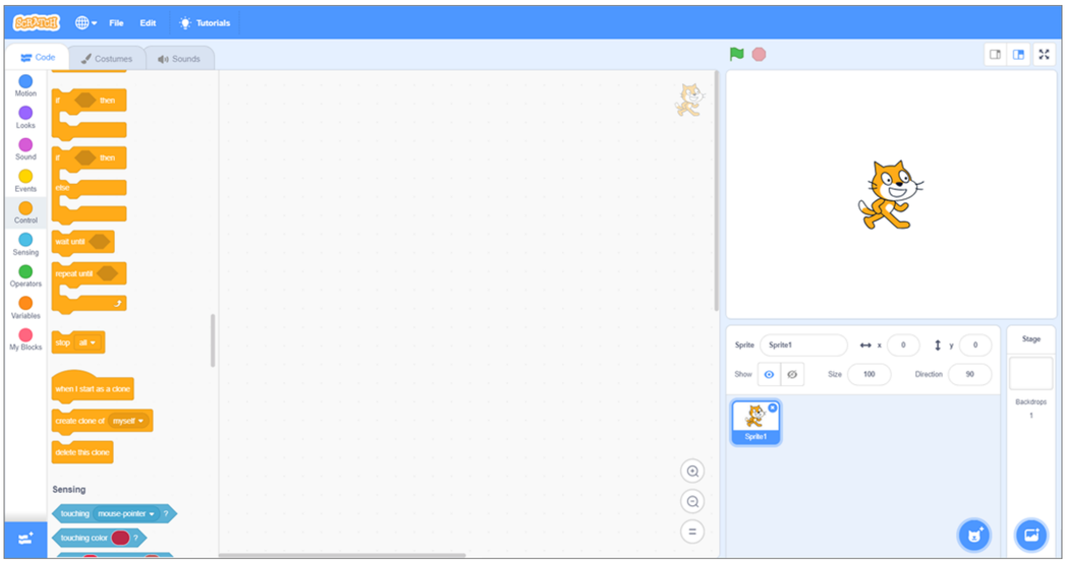


Figure 2.1.: Scratch Desktop (Version 3.0).

Sprites encapsulate state and behavior, but they cannot communicate directly with each other. Scripts of one sprite can't directly call a command on another sprite. Inter-sprite communication and synchronization takes place using a broadcast mechanism where the sprite broadcasts user defined string message that activates all scripts starting with block *When I receive ⟨msg⟩*.

Each sprite can have more than one script running concurrently, allowing multithreading. As Resnick et al. (2009) explain, in this way Scratch tries to make parallel execution more intuitive. Most of the common race conditions that would result from multithreading are eliminated by the thread switch being able to occur only at the end of a loop or by an explicit wait command.

The shape of the individual blocks, limit the way they can be combined, i.e., they can only be connected in executable combinations, ensuring syntactic correctness. They have a bump on top or a notch on the bottom indicating the possibility of connecting to another block, or a slot for another block in the block body. Maloney et al. (2010) point out that in this way the visual grammar prevents syntax errors from being introduced to the program.

The blocks are divided by their shape, into 6 categories as shown on the figure 2.2:

- *Hat* blocks indicating the beginning of a script and contain a trigger that starts the script,
- *Stack* blocks, category with the most blocks, performing main commands,
- *Boolean* blocks that contain conditions,
- *Reporter* blocks that contain values,
- *C* blocks, also called Wrap blocks that control program flow by executing conditions and loops,
- and *Cap* blocks, used to terminate scripts or programs.

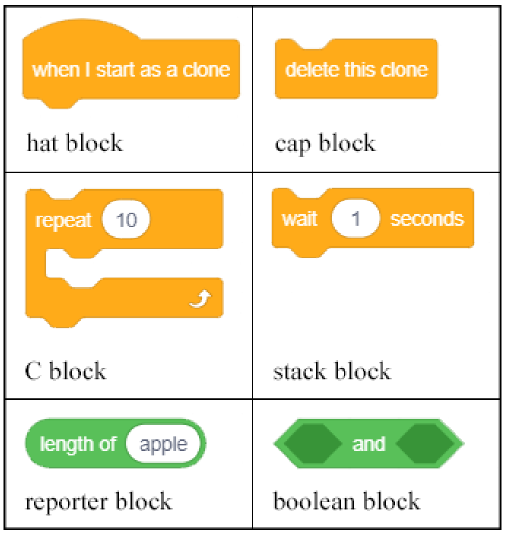


Figure 2.2.: Six types of Scratch blocks (adapted from *Scratch Wiki* 2019).

In addition to the distinction by shape, the blocks are divided by type into 9 categories (plus 10th category for custom blocks). Each category has its own color.

2.3. Project Catrobat

The Catrobat project is a visual programming language and a set of *creativity tools* inspired by Scratch and developed by the Catrobat team under Wolfgang Slany. It was published as a public beta in spring 2010. The project aims at creating applications on various mobile platforms for the main purpose of helping beginners and younger users to acquire coding skills (Luhana, Schindler, and Slany, 2018).

The Catrobat visual programming language was heavily inspired by Scratch. However, while Scratch is an application for personal computers, Catrobat is specifically designed for mobile devices (Slany, 2012a). An HTML5 version was also in development which made it also available for HTML5-capable browsers on any platform.

On Android an integrated development environment (IDE) and interpreter called Pocket Code, developed from a previous version, called Catroid, is available for the Catrobat language. Other tools integrated into the Pocket Code application include Pocket Code webshare, allowing users to upload their creations to the community website as well as to download projects from other users from that sharing platform. A Scratch2Catrobat converter is available and can be accessed from within Pocket Code. It enables access and conversion of all Scratch projects into the Catrobat language. It is still in development therefore some of the converted projects are not 100% functional since Catrobat is not yet 100% Scratch compatible - there are still some missing bricks in Catrobat.

In addition to the Pocket Code application, a sophisticated graphical editing software called Pocket Paint is available, allowing users to draw and edit all graphic objects in Catrobat projects within Pocket Code (Slany et al., 2018). It is available either as a standalone application or as a part of the Pocket Code app.

Similarly to Scratch, the Pocket Code IDE conceals the underlying code behind the simple functional bricks that can be snapped together to create programs. The necessary Catrobat bricks can be selected from a list of bricks and connected via drag and drop into scripts that can run concurrently.They can communicate with each other using broadcast messages. Since Catrobat is specifically designed on mobile devices, it offers the ability to use mobile device sensors, for example, to determine device inclination.

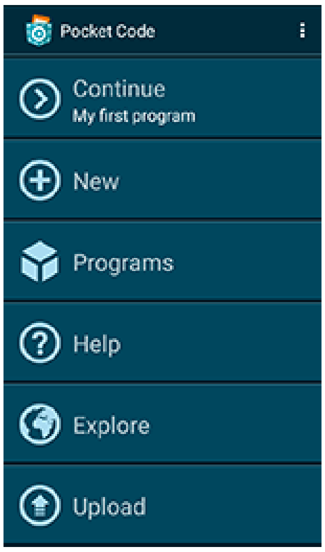


Figure 2.3.: Pocket Code main menu.

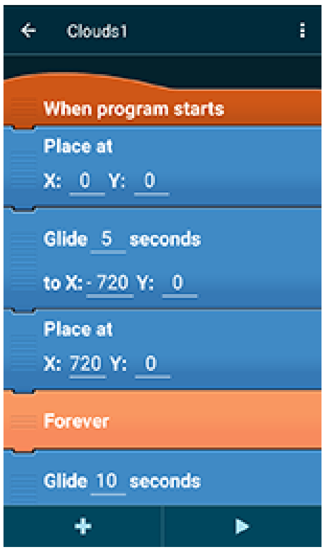
2.4. Catrobat Language Syntax

Catrobat programs consist of scripts, virtual containers for blocks of code called bricks. Each script belongs to one object or to the background of the application and refers to it. Each object (or the background) can contain multiple scripts which can be designed to run concurrently. Each Catrobat project contains at least one background and can contain multiple objects in addition to it. Each script consists of an event brick (a starting point) that determines when the script should run in the program which can be followed by one or more bricks with program logic. The individual Catrobat bricks fall into one of the following categories:

- *Event* - bricks used as entry points for the execution of scripts or to broadcast messages.
- *Control* - bricks to control the program's flow such as conditionals, loops and waits.
- *Motion* - bricks concerned with the movement of objects, determining their position, speed and direction of their movement along with some physical properties.
- *Sound* - bricks adding sound files to the programs.

- *Looks* - bricks used to change the appearance of an object and to control the camera.
- *Pen* - bricks enabling drawing by letting the object leave a trace behind as it moves.
- *Data* - bricks used to create, store and change variables and lists in programs.

Beside these categories, Catrobat extensions can be unlocked to add categories of bricks to control various devices such as Lego Mindstorms Robots, Arduino or Raspberry Pi Boards.



```
When program starts
Place at
X: 0 Y: 0
Glide 5 seconds
to X: -720 Y: 0
Place at
X: 720 Y: 0
Forever
Glide 10 seconds
```

Figure 2.4.: Catrobat script.

Catrobat uses a different approach to working with formulas than Scratch. Instead of using bricks specifically for equations, there is a formula editor in Pocket Code to create mathematical and logical equations that can then be used directly as parameters of individual bricks. This hybrid system has been shown to be faster and less error-prone among users than Scratch brick equations or text equations in common programming languages (Slany et al., 2018; Harzl et al., 2013) .

The formula editor (figure 2.5) offers functionality similar to a basic calculator along with providing access to more advanced possibilities divided into five categories. The *Object* category provides access to the object or

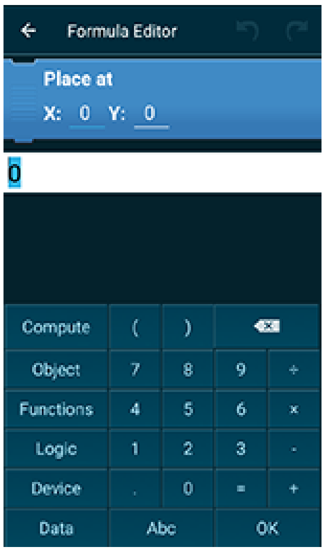


Figure 2.5.: The formula editor of Pocket Code.

background specific variables defining its look or movement properties. More advanced functions are grouped into the *Functions* category, providing mostly mathematical functions such as exponentiation and trigonometric functions. The *Logic* category extends the operator list by adding boolean and comparison operators. Variables in the *Device* category (as shown on the figure 2.6) include time values, screen touch and face detection data, and sensor values recorded by the device such as inclination.

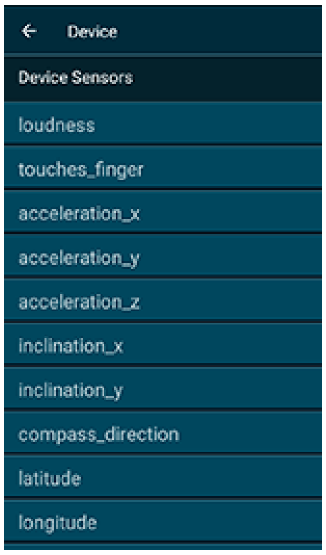


Figure 2.6.: Device specific variables.

3. Behavior Driven Development

Testing is a critical factor in determining software quality. Patton (2000) defines its goal as find out the existing bugs and making sure the get fixed. Dijkstra et al. (1970) says:

> Program testing can be used to show the presence of bugs, but never to show their absence!

Patton (2000) also points out that the cost of repairing an unrecognized bug in the system increases logarithmically with the time it is not found. A large number of errors occur before the start of writing program code as a result of misunderstanding between the customer and the development team, so it is important to find them as soon as possible.

Therefore, a technique is needed that will ensure not only the validation (checking if the product meets the needs of the customer to see if the right product is being built), but also the verification of the software in development (checking against the specification to see whether the product is being built right). It must ensure that these two conditions are met throughout the development and that each member of the team, including the customer, is involved.

One such technique is behavior driven development that, as Wynne and Hellesøy (2012) say, brings a ubiquitous language understood by the whole team to facilitate communication between customers, developers, and testers.

3.1. Agile Software Development

The Catrobat project is developed according to agile principles. Agile software development is a set of frameworks and practices that was created in the early 2000's by a group of software developers calling themselves *Agile Alliance*. These practices are based on the values expressed by the agile manifesto and 12 principles behind it. The four values of agile manifesto are:

> Individuals and interactions over processes and tools
> Working software over comprehensive documentation
> Customer collaboration over contract negotiation
> Responding to change over following a plan[1]

All agile methodologies apply these four values, although the way they are applied varies between methodologies.

Individuals and interactions over processes and tools - A functional team with good communication is more important to the project than a good process, because it is the people who drive development and team skills increase the chances of success more than individual programming skills or the right development environment or compiler, though even these can have a big impact on success.

Working software over comprehensive documentation - Although code documentation is important, Martin (2003) says that too much documentation is more damaging to the project than too little. This leads to a new set of problems, where much time is spent on creating software documents and even more time is being lost while keeping this oversized documentation in sync with the code that is being developed. Martin (2003) further points out that the best way to transfer information to new team members is the code itself and other team members anyway.

Customer collaboration over contract negotiation - The requirements of the customer at the beginning of a project are almost never exactly identical to the requirements at the end. The requirements evolve organically as the project progresses and the customer should therefore work closely with the development team to deliver regular and frequent feedback.

Responding to change over following a plan - The fourth point is closely linked to the third, as the requirements and other factors change and adapt over the lifetime of the project, the ability to respond to these changes is very often one of the most important factors determining the success or failure of the project.

[1]https://agilemanifesto.org/

Larman and Basili (2003) explain that agile development models differ from the traditional waterfall model (as shown on figure 3.1), in them being based on an incremental and iterative approach. The traditional waterfall model uses a sequential approach where each phase of the software development process contains one fundamental activity and the individual phases follow linearly.

In contrast, Stoica, Mircea, and Ghilic-Micu (2013) say that incremental models divide software development into smaller units and add new functionality over small increments. By doing so, they offer greater flexibility in development as the introduction of change into the system during development is cheaper. Each "increment" represents a piece of functionality that is required by the customer, so it is possible after each increment to introduce the given feature to the customer for feedback, with feedback only affecting the work done during the last increment. This way:

- The cost of changing requirements is reduced.
- The customer can see a functional feature and give a more accurate feedback.
- The development can focus on important functionality first and less critical parts of the software can be added in later increments, making software delivery faster.

Iterative development aims to develop the system in small parts with the rapid fulfillment of the initial scope for fast release and feedback. The scope is then increased with each iteration.

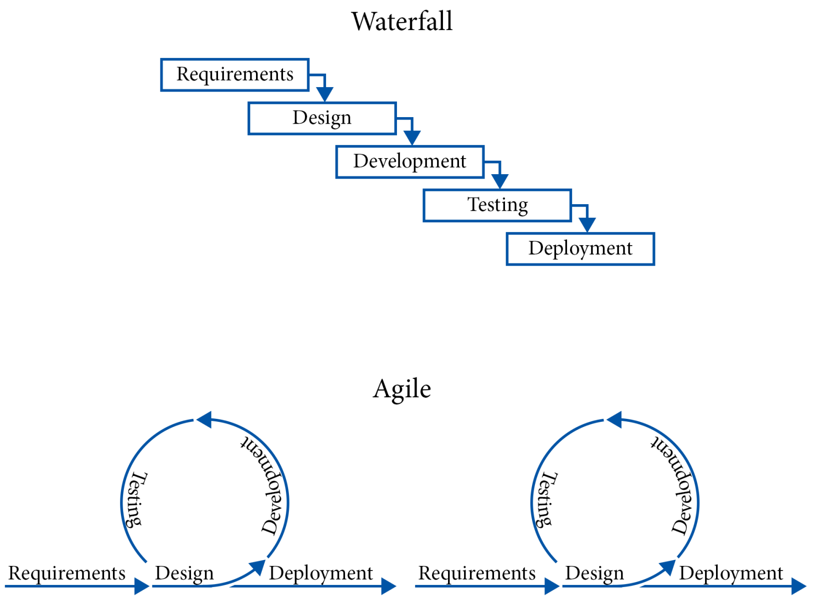


Figure 3.1.: The life cycle in waterfall model and agile.

These two processes (iterative and incremental development) combine in the agile in a model where the development cycle is iteratively repeated with incremental addition of new functionality in small chunks and each iteration produces deliverable software (Alshamran and Bahattab, 2015).

The Catrobat project uses several agile practices to develop, with test driven development having the strongest influence on the development process.

3.2. Test Driven Development

Test driven development (TDD) is one of the essential methods of agile programming. In spite of the name, it is not a testing technique, but a process of agile design and software development, the basic idea of which is to write unit tests before the program code. Since the code is written after the tests, using TDD automatically implies that all code written is testable.

The primary purpose of writing tests in TDD is their function as a tool for creating the resulting system. Freeman and Pryce (2009) explain that it adds executable description of what the code does and a complete regression suite. Writing tests before the development of a feature forces the developer to first thoroughly think about what needs to be done and how the feature will be used and only then to start writing its code. Freeman and Pryce (2009) describe this phenomenon as a separation of logical design from physical.

Subsequent running of these tests at the end of each phase of the TDD cycle results in error detection soon after they are introduced. The developer still has fresh knowledge of the code segment that led to the bug and can more easily pinpoint the source of the problem using the tests that already exist. In this way the development cost is reduced because the cost of error correction during the testing phase of a conventional non-TDD methodology is according to Koskela (2007) usually two magnitudes higher than if they were found as soon as they originated.

David Astels (2003) explains that TDD is based on repeating of a short cycle that turns the traditional methodology around - firstly, automated unit tests are written, then comes the production code development and finally the code is refactored – which is changing the code without altering its external behavior.

The TDD cycle is also called Red-Green-Refactor. Red represents the state of the program when a new feature is to be added. According to Koskela (2007) this is the part of the cycle where the design decisions are made. First a test is written, defining the behavior of said feature (its logical design). This test understandably fails as the feature does not exist yet – thus the name Red. Green phase of the cycle covers writing the production code until all original and newly added tests run successfully. The Refactor phase serves to simplify the code, without breaking the tests. During Refactor phase duplications are removed from the code, its structure is simplified and other changes are performed to obtain better and more readable code.

The goal of TDD is, as Beck (2002) describes it, to create *clean code that works* by driving the development with automated tests: the new code is written only when a test fails and no more code is written than is needed to pass the test. With this approach, all code is testable and tested by definition.

Koskela (2007) talks about the three great benefits that are brought by the use of TDD. The first is fewer bugs in the code, because the development is split into small steps, so that the developers can focus on the problem at hand and because the new code is covered by tests before it is even written. The second benefit of TDD is saving time which is based on the fewer bugs present and because the development stops when all tests run successfully, so no additional and unimportant functionality is introduced in the code. Third, the test coverage enables the developers to see that at the end of the cycle new code did not break old functionality, which increases their self-confidence.

As a summary, according to David Astels (2003), the TDD can be described as a development style, where:

- An exhaustive suite of Programmer Tests is maintained. This builds up a test harness which allows regression testing and checks whether the refactoring did not change functionality.
- No code goes into production unless it has associated tests. This implies a higher production code quality.
- Tests are written first. The code is per se designed to be testable since one or more tests are written before the production code.
- The tests determine which code is needed to be written. The tests are an executable specification and better describe the requirements than any comments.

3.3. Acceptance Test Driven Development

Unit tests, on which the TDD approach is based on, test only small isolated pieces of code that somehow affect the behavior of the system and are therefore not suitable to be the only tests in the systems. Unit tests and TDD can test whether the ″code was written correctly″, but it is not possible to evaluate the correctness of the problem solution. Therefore, acceptance tests are used to indicate whether the ″correct code″ has been written. Koskela (2007) defines acceptance tests such as:

- Owned by customers. Since their purpose is defining the acceptance criteria of a feature, the customer knows best what they want from that feature.
- Written together with the customer, developer, and testers. A customer in the role of the domain expert and a developer as a technical expert write the acceptance tests together to encourage communication.

- About *what* and not about *how*. Acceptance tests describe the source of value for the customer, not how the value is delivered.
- Expressed in the language of the problem domain. Acceptance tests are written in a language that the customer can understand.
- Concise, precise and unambiguous. An acceptance test verifies a single aspect or scenario relevant to a feature.

Managing development by defining acceptance tests before writing code is similar to writing unit tests before implementing a new program logic in TDD. This methodology is called the acceptance TDD (ATDD) and complements the TDD. Freeman and Pryce (2009) note, that while TDD assists with internal code quality, ATDD ensures its external quality – that is how well the system meets the needs of customers and users.

The ATDD cycle (shown on figure 3.2) is very similar to the TDD cycle - when the implementation of a new feature starts, first an acceptance test is written. It fails, as the feature does not yet exist. Then the code is written until all acceptance tests are successful. This also serves as information that the feature is done. During one ATDD code development phase, usually several TDD cycles (at least one) take place.

As the ATDD deals with the customer's needs, unlike the TDD, it is a whole-team technique. Larman and Vodde (2010) even claim that if the whole team is not involved, including the product owner or representative, the development process is not a ATDD process. In contrast to TDD in ATDD the stakeholders themselves should participate in the definition of acceptance tests, even create a part of the test suite themselves. Koskela (2007) defines the main objective of ATDD in supporting this close collaboration between customer, developer and tester parties.

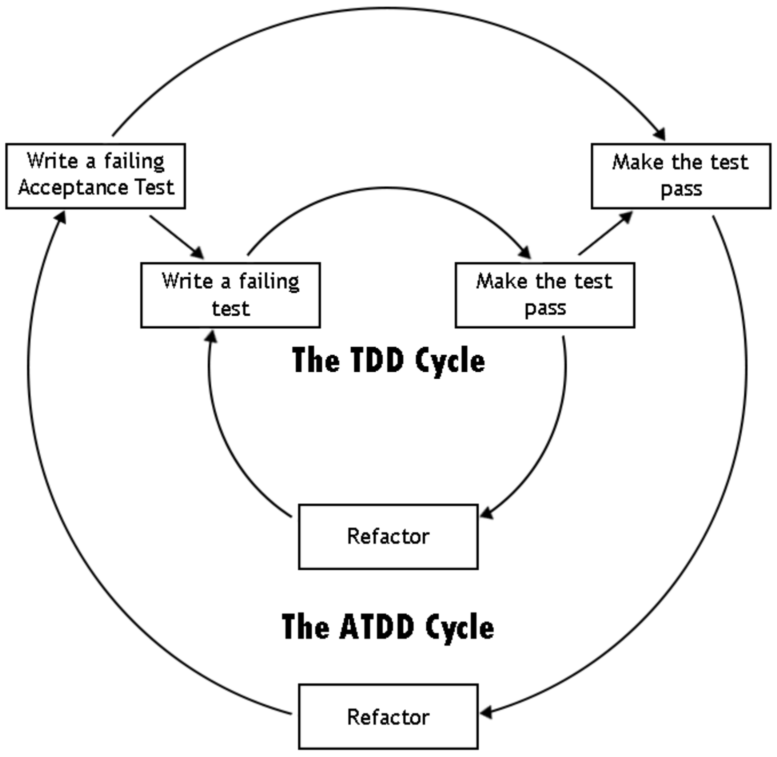


Figure 3.2.: ATDD Cycle

3.4. Behavior Driven Development

Despite the advantages of TDD, this methodology has some problems. Dave Astels (2006) sees the largest of them in the language used by TDD. Because of the common use of terms like test and unit, many developers see TDD just as writing tests before the code itself, while they should be looking at how to write a test in a concise, unambiguous and executable form. He bases this statement on Sapir-Whorf's hypothesis, stating that the way an individual thinks is shaped by the language they use. Therefore to change the way people think about TDD, the language used to explain TDD must be adjusted first. In addition, the idea of testing often encounters negative responses from developers and project managers, therefore a change of terminology is also useful in this regard.

Similarly North (2017) saw problems with the effective adoption of TDD. Developers often did not know where to begin writing tests, which tests to write next, or focused too much on details, losing in the process a wider view of the original business goals. North has therefore developed behavior driven development (BDD) as an attempt to better understand and explain TDD. It is an evolution of TDD and ATDD, that takes the ATDD process and replaces the terms like unit and test with behavior and specification. BDD focuses on properly translating the requirements of stakeholders into features. From TDD it takes the emphasis on automated software testing. This gives immediate feedback on the state of the application. ATDD respectively can be seen in the orientation of testing at the acceptance testing level.

Smart (2014) explains that BDD combines the test driven development with a domain-driven design, from which it takes the concept of a common dictionary, also known as the ubiquitous language. In BDD this dictionary is based on simple structured sentences in English, or in another language understood by all stakeholders. It is developed by the whole team and is designed to create a communication bridge between development and business and helps meet the most important BDD principle that says business and technology people should talk about the same system in the same way.

The common language then allows the customer to specify requirements from a business perspective, the business analyst to add concrete examples to specify the behavior of the system, and the developer to subsequently implement the required system behavior in the TDD way. Lazăr, Motogna, and Pârv (2010) say BDD is often described as "TDD done correctly".

There are three principles in the BDD activity framework, according to North (2014). First of all, the system should be defined in the concepts of its behavior so that both development and the business side of the team can use the same language independently of the granularity level. In short *everything is behavior*. Second principle is *where is the business value* meaning that if something is to be added to the system, it must bring business value to the stakeholders or be a prerequisite of some other feature that does. Behavior that has no current business value for the project is unlikely to be added to the system. Lastly *enough is enough* says that the usual processes that take place at the beginning of a project, like planning, analyzing and designing of the system should only be done to such an extent that the development can be started. The reason is if the requirements change, all work above that would have to be repeated and therefore can be regarded as superfluous.

Solis and Wang (2011) extracted 5 basic BDD properties:

- A ubiquitous language is used.
- An iterative decomposition process - analysis begins with the need of a customer, divided in stories and specific examples; this approach is also referred to as an outside-in.
- Plain text descriptions with user stories and scenario templates are used. User stories and scenarios use a language and a structure which is understood by the whole team.
- Automated acceptance testing with mapping rules. The specification of behavior in BDD plays the role of executable acceptance tests.
- Readable behaviour oriented specification code. The code is part of the system documentation.

From a mechanical point of view, BDD is almost the same as ATDD - tests are written before the code, the Red-Green-Refactor cycle is used, and during one iteration of "Green" there are likely to be several TDD cycles involved. Unlike ATDD, however, it does not start with a test but with a set of requirements that the new feature should meet. The requirements are defined by the entire team, including users and other stakeholders in conversations. The result of this process are user stories and scenarios describing user requirements for the feature in the form of concrete examples using the common language offered in the BDD. This concept of using specific examples, called specification by example, comes from Parnas (1972) who specifies the rules for such specification:

- The specification should contain all the necessary information for the user to use the program and nothing more.
- The specification should contain all the necessary information for the implementer about the intended purpose that is needed to complete the program and nothing more.
- The specification should be formal enough to be machine testable.

- The specification must use a language understood by both the user and the implementer.

Specification by example also creates living documentation. Adzic (2011) describes it as an authoritative source of information about system functionality that everyone has access to and which is as reliable as the code but more comprehensible. It serves as documentation of requirements as well as technical documentation. It is made up of the requirements themselves. They are executable, so there is always an overview of all the requirements that are already implemented. Living documentation can not become obsolete and outdated like standard documentation.

3.5. User Story

BDD offers a specific way of communication between stakeholders and developers by using ubiquitous language: In this language it is defined when the program is finished and functional. The basic element of this communication is a user story that acts as a basic unit of functionality and delivery in BDD. It describes one discrete piece of functionality and its main task is to describe the requirement in such a way that all interested parties (management, analysts, testers, developers) understand it. North (2019) lists the basic elements of a user story as the title, narrative and acceptance criteria in the form of scenarios.

The title is a short and concise description of the behavior that the story implementation will bring. The narrative of a user story shows who requests what functionality and what benefits it will have. North (2019) specifies the template for writing a user story narrative:

> As [role]
>
> I want [feature]
>
> So that [benefit]

North also provides a template to serve as minimum requirements of a user story.

```
Title (one line describing the story)

Narrative:
As a [role]
I want [feature]
So that [benefit]

Acceptance Criteria: (presented as scenarios)

Scenario 1: Title
Given [context]
And [some more context]...
When  [event]
Then  [outcome]
And [another outcome]...

Scenario 2: ...
```

Listing 3.1: User Story template (adapted from North (2019))

Acceptance criteria are presented based on specific examples of behavior described by the user story. Each of these scenarios has its own title and a description. The title describes what differs from other scenarios. The description is divided into the context in which events take place, events that happen during the scenario and the consequences of these events. According to North (2017) the acceptance criteria form the body of a user story, because they define the extent of the behavior and show when the story is finished. Their individual components are small enough for them to be represented directly in the code, making the scenarios executable specification.

The scenarios interact directly with the code, but are also written in a language that business stakeholders understand. They create a place where both sides of the linguistic division, as it is called by Evans (2003), meet.

3.6. Cucumber

Cucumber is a BDD framework, originally created for Ruby and subsequently ported in many languages. Dees, Hellesøy, and Wynne (2013) describe its goal as automation of evaluating whether an application behaves according to specified behavior using automated tests that are understood by both the developer team members and all other stakeholders, which it shares with JBehave.

Chelimsky et al. (2010) explain its origin as originally a part of RSpec. When Dan North came up with the BDD concept, he created the JBehave tool in 2003 as probably the first tool to deploy the BDD in development. Later in 2005, when Steven Baker began developing the BDD framework for Ruby called RSpec, JBehave was ported to Ruby as RBehave, refactored under the new name "Story Runner" to use plain text and merged with RSpec. In 2008, Aslak Hellesøy rewrote Story Runner with real grammar and created the tool Cucumber. The idea was for BDD to have a cycle similar to ATDD, with Cucumber being the outer circle and RSpec forming the inner circle, but Cucumber was ported into many other languages and can also be used as a stand-alone tool.

As mentioned above, BDD user stories have title, a narrative and scenarios. Cucumber focuses only on the scenarios, as they make up the executable specification. Instead of stories, in Cucumber the scenarios that relate to one feature are grouped into a feature file. Feature files contain a description of the behavior written in Gherkin, which is very similar to the common language. They have a well-defined structure, based on which Cucumber builds automated tests. At the beginning of a feature file, there may be optional language declaration in the form "# [language]", where [language] is replaced by the language in which the feature file will be written. If this declaration is omitted, Cucumber defaults to English. After that follows the feature name in form "Feature: [name]", usually followed by a brief plain text description that may or may not contain the narrative described in the BDD subchapter. The title is then followed by one or more scenarios associated with the feature.

Each scenario begins with a name in the form of "Scenario: [name]", which describes the scenario. Then comes the description divided into steps consisting of short phrases, starting with keywords. These keywords are *Given*, *When*, *Then*, *And* and *But*, or their equivalents in the language in which the script is written. A step with the *Given* keyword describes the context of the scenario and the initial state of the application. *When* represents the action that is taking place, and *Then* the expected result. The keywords *And* and *But* are used to expand one of these three types of steps. This keyword breakdown is for the benefit of the reader, as Cucumber does not distinguish between them and even offers "*" as a universal keyword to mark a scenario step. Comment lines are distinguished by # at the beginning.

Listing 3.2 shows part of a sample Cucumber feature file for the Pocket Code with just one scenario for the Pocket Code Formula Editor and in listing 3.3 the same scenario is written using the universal keyword "*".

```
Feature: Formula Editor

   In order to use formulas as part of the catrobat
   brics, the formula editor offers a way for the users
   to create and modify them

  Scenario: Changing a formula with Formula Editor
    Given I have a Program with a Wait brick
    And I am in the script section
    When I open Formula Editor via brick
    And I change the formula to 5
    And I press ok in Formula Editor
    Then I should see 5 in the brick
```

Listing 3.2: A sample Cucumber feature file with one scenario

```
Feature: Formula Editor

   In order to use formulas as part of the Catrobat
   brics, the formula editor offers a way for the users
   to create and modify them

  Scenario: Changing a formula with Formula Editor
    * I have a Program with a Wait brick
    * I am in the script section
    * I open Formula Editor via brick
    * I change the formula to 5
    * I press ok in Formula Editor
    * I should see 5 in the brick
```

Listing 3.3: A sample Cucumber feature file using universal keyword

To automatically test app behavior, Cucumber, in addition to the feature file, also needs a set of step definitions describing what to do when it goes through the steps described in scenarios. This step definitions file, written in one of Cucumber's supported programming languages, maps the steps of scripts written in plain text to the corresponding code program methods. Each of these methods begins with annotation of one of the keywords *@Given*, *@When*, *@Then*, *@And* or *@But* followed by the parameter. This parameter is a regular expression containing one of the steps in the scenarios. It is surrounded with the "^" and "$" symbols indicating the beginning of the string and the end of the line. A sample step definition file for a feature described in listing 3.2 is shown in listing 3.4. The same step definition file works also with the scenario description in listing 3.3. While there is no code implemented yet, Cucumber uses a *PendingException* to indicate so when running the tests.

When performing the step, Cucumber searches with a regular expression for a method with an annotation corresponding to the step. Since there may often be scenarios with similar steps that differ only in some values, a regular expression in brackets can be added to the annotation instead of this value. It serves as a capture group that will be given to the annotated method as a parameter. For multiple capture groups, the parameters are added in the same order as the capture groups are in the annotation.

```
@Given("^I have a Program with a Wait brick$")
public void i_have_a_Program_with_a_Wait_brick() throws
    Exception {
  // Write code here that turns the phrase above into concrete
  // actions
  throw new PendingException();
}
@Given("^I am in the script section$")
public void i_am_in_the_scripts_section() throws Exception {
  // Write code here that turns the phrase above into concrete
  // actions
  throw new PendingException();
}
@When("^I open Formula Editor via brick$")
public void i_open_Formula_Editor_via_brick() throws Exception {
  // Write code here that turns the phrase above into concrete
  // actions
  throw new PendingException();
}
@When("^I change the formula to 5$")
public void i_change_the_formula_to_5() throws Exception {
  // Write code here that turns the phrase above into concrete
  // actions
  throw new PendingException();
}
@When("^I press ok in Formula Editor$")
public void i_press_ok_in_Formula_Editor() throws Exception {
  // Write code here that turns the phrase above into concrete
  // actions
  throw new PendingException();
}
@When("^I should see 5 in the brick$")
public void i_should_see_5_in_the_brick() throws Exception {
  // Write code here that turns the phrase above into concrete
  // actions
  throw new PendingException();
}
```

Listing 3.4: Step definitions file for the feature in listing 3.2

Other methods for removing duplicates in Cucumber are applied in feature files by using *Background* and *Scenario Outline*.A background is used when all scenarios of one feature file begin with the same steps. It groups these steps before the scenario definitions under the keyword *Background:* followed by those steps in the same form as they were in the scenarios from which they were removed. When running a scenario of a feature file with background, the background-steps are always executed before the scenario itself begins.

The *Scenario Outline* key phrase is used in scenarios where the group of scenarios differ only in the input values. It replaces the *Scenario* keyword and substitutes the variable input in each step by the variable name in chevrons as a placeholder. Then a table is put below the list of steps containing sets of variables that run each test. It starts with the *Examples:* keyword under which the table itself is located. In the first row of the table are all variable names, and each row below is a single test set. Each row of a table is marked by surrounding it with the "|" symbol, which also serves as the divisor of the individual variables in the row. Taking the listing 3.2 instead of writing several scenarios that would use different values, a scenario outline can be used to compress them into just one as shown in listing 3.5.

```
Feature: Formula Editor

  In order to use formulas in the Catrobat bricks, the formula
  editor offers a way for the users to create and modify them

  Background:
    Given I have a Program with a Wait brick

  Scenario Outline: Changing a formula with Formula Editor
    Given I am in the script section
    When I open Formula Editor via Wait brick
    And I change the formula to <input_value>
    And I press ok in Formula Editor
    Then I should see the value <expected_value> in the Wait
       brick

  Examples:
    | input_value | expected_value |
    | 5           | 5              |
    | 2+2         | 2+2            |
    | 35.35       | 35.35          |
```

Listing 3.5: A sample scenario outline

4. Problem Statement

This thesis addresses two main problems. The first one is allowing to perform tests on mobile devices for features, where different physical parameters play a role, such as the position in which the device is. The second problem is introducing BDD to Pocket Code to create platform independent tests to unify test suites for each platform on which Catrobat features are being developed.

4.1. Sensor Tests

Since the Catrobat project is being developed in a strict TDD way, it is a prerequisite that for each existing piece of code there will be one or more tests serving to confirm its functionality and as a form of documentation for that code. However, in Pocket Code, there are parts of the system where these tests do not exist. One group of such features is the capability to read the sensor data of the device on which Pocket Code runs. The only way to test such features at the current time is manual testing.

However, in his book *Succeeding with Agile: Software Development Using Scrum*, Cohn (2009) says manual tests should only be used for exploratory testing. Even some later versions of the testing pyramid introduced by Cohn that have evolved from it, put manual tests at the top of the pyramid as the smallest group of tests used to test the entire system. The ability to use the device's sensor data requires updating these data during runtime. Automated versions of such tests must be implemented as UI tests, since the device's sensor data are accessed via stage activity of Pocket Code. Along with the need to move the device to change certain sensor readings the tests will be relatively slow and thus better be part of an extended test suite, which is not executed at every test run.

To change sensor readings for sensors reacting to the mobile device's location/direction or movement in space, the device needs to be moved. For this automated movement, a robot arm will be built. This robot arm must be robust enough to handle a mobile device with a weight of up to 300 grams stably and safely. In addition, its range of motion must allow the movements necessary to perform all tests for sensors reacting to location/direction or movement in space, therefore the following minimum movements must be possible:

- Rotation of the device into horizontal and vertical position: A minimum of 90 degrees rotation on the y-z axis of the robot arm.
- Rotation and movement of the device allowing the transition from a lying position to standing position: A minimum of 90 degrees movement on the x-y axis of the robot arm.

4.2. Platform Independent Testing

In the Catrobat project Catrobat IDEs are developed for Android, iOS, and HTML-5 , with a distinct team working on each of these platforms. Since all teams work on the same project, just for different platforms, each feature should work the same. Catrobat is developed according to the TDD methodology, therefore the functionality of each of the features should be explained by tests written before these features are implemented and added to the project.

Thus, if it were possible to abstract the tests to a platform-independent form, it would be possible to create a unified test suite for all platforms and ensure that there is no different implementation of the same functionality by individual teams. This is where BDD comes in. BDD creates another layer of code in the form of tests written in Gherkin language that are part of user stories. Using the executable specification from BDD as an abstraction layer for new tests will create a platform-independent description of each feature.

To do this, a suitable BDD tool must be integrated into the Catrobat IDE Pocket Code and the new tests must be created using this tool.

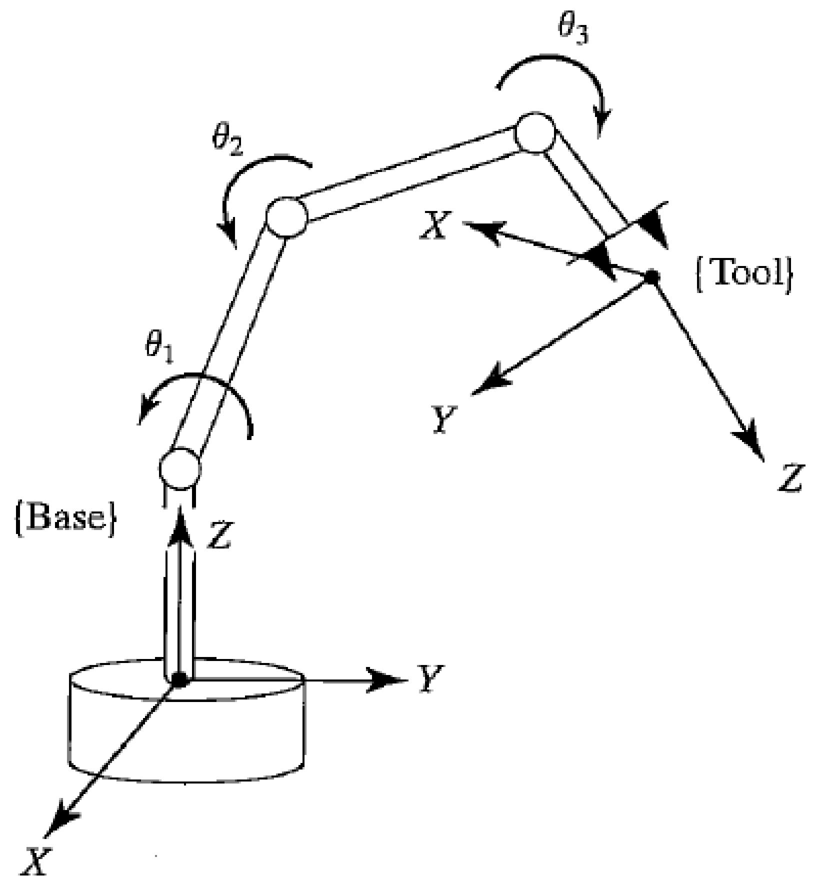
5. Robot Arm

To control a robot arm in a meaningful way, it is very important to know the position and the orientation of the robot arm in space. The information of the position and orientation may be used to calculate the position and the orientation of the end effector (gripper or holder in this case). An end effector is the part of a robot arm that is designed to interact with the environment such as a screwdriver, painting gun, welding gun or a gripper. A robot arm consists of joints. The angles occurring on these joints serve with the position and the orientation information for further calculations. In different manipulators (robots) different joints might be present realizing angular or sliding movements. The part of the robot where it is stabilized to a base, is called base frame, whereas the part of the robot containing the end effector is called tool frame.

The robot arm which is used in the Catrobat project for this thesis consists of 3 arm joints, allowing 3 degrees of freedom. The number of degrees of freedom that the robot arm has, can be considered as the number of independent position variables that would have to be specified in order to locate the position and the orientation of the gripper. Robot arm joints may have different terms of movement depending on how they have been constructed. For instance, joints may be rotary or revolute. The displacements of these joints are called joint angles. Some robots may contain sliding (or prismatic) joints, on which the relative displacement between links is a translation, sometimes called the joint offset (Craig, 2004). For the chosen design of the robot arm, rotary joints have been used, making use of servomotors.

Servomotors are rotary actuators that unlike conventional motors, can set the exact position of the axis rotation. Connection of all the joints together in a robot arm, often end up with an end effector according to Craig (2004). In the case of the robot used in the Catrobat project for this thesis, the end effector was chosen to be a gripper (robotic claw, using a servomotor for gripping movement). Later on, it was seen that the gripper is not stable enough for tshe purpose of this project, where it is aimed to make automated tests on a mobile phone, which can weigh up to 300g. Therefore, the gripper later on, is changed with another end-effector, basically a mobile phone holder.

The collection of the robot arm joints are considered as independent position variables. Their function depends on the type of the joint itself (angular movements, linear movements, etc.). In order to make calculations to allow proper movement of the robot arm, kinematic equations are used. Most often, two coordinate systems are defined in order to use with the kinematic equations. In many cases, one of the coordinate systems are positioned on the base joint of the robot arm with an orientation (angulation of the x,y,z coordinates in the three-dimensional space). This is called base-frame according to Craig (2004). The other coordinate system is positioned on the gripper (or end effector as called by Craig (2004)) with a desired default orientation. This coordinate frame is called tool frame in Craig's book. Kinematic equations can mainly be classified in two: forward kinematics, and inverse kinematics. Forward kinematics is a very basic problem in the study of mechanical manipulation according to Craig (2004). It is a static geometrical problem of computing the position and the orientation of the gripper of the robot arm. Given a set of joint angles, the forward kinematic problem for our case is to compute the position and orientation of the tool frame relative to the base frame. On the other hand, given the position and orientation of the gripper of the robot arm, inverse kinematics equations calculate all possible sets of joint angles that could be used to achieve this position and orientation of the gripper. In many cases, inverse kinematics equations are more suitable for robot arms as mechanical manipulators. See figure 5.1.



[t]

Figure 5.1.: Kinematic equations describe the tool frame relative to the base frame as a function of the joint variables. Adapted from Craig (2004).

Once the desired degrees of freedom (DOF) are decided for a robot arm, a kinematic configuration shall be used to design and connect the robot joints in order to realize the desired DOF. Different kinematic configurations may help achieving certain geometrical reach for the gripper of the robot arm. For instance, if a Cartesian manipulator is used with 2 DOF, the following cubic-workspace may be achieved in a 3 dimensional space as shown in the figure 5.2.

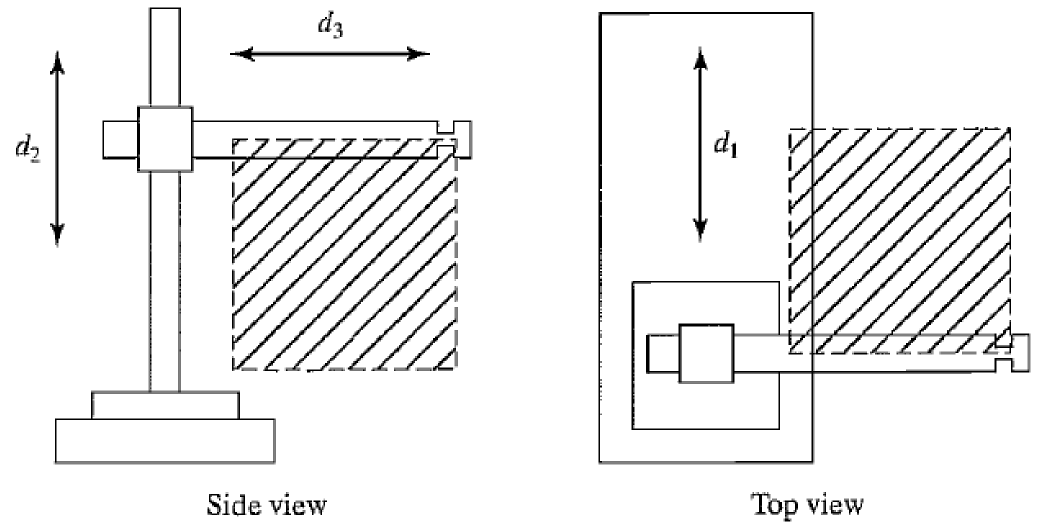


Figure 5.2.: Cartesian manipulator with 2 degrees of freedom may achieve a cubic workspace in the 3 dimensional space. Adapted from Craig (2004).

For the purpose of this project, a kinematic configuration similar to SCARA (selectively compliant assembly robot arm) configuration given in Craig (2004) has been used with some modifications, where all the joints are limited to 90 degrees of movement for safety purposes. In addition to the SCARA configuration given in the following figure 5.3, an additional base joint is used to allow 90 degrees of movement on the x-y plane (consider having the same orientation of the base frame axes as figure 5.1). Furthermore, the joint for the end effector has been set orthogonal to the other 2 joints following the base joint, allowing 90 degrees of rotation at the mobile phone holder (end-effector). Workspace created using this configuration would be very similar to the workspace given in the figure 5.3. Of course there are other kinematic configurations given or may be designed such as spherical, cylindrical, cubic, etc. depending of the target application of the robot arm (or mechanical manipulator). It is possible to create a universal manipulator (robot) with 5-6 degrees of freedom.

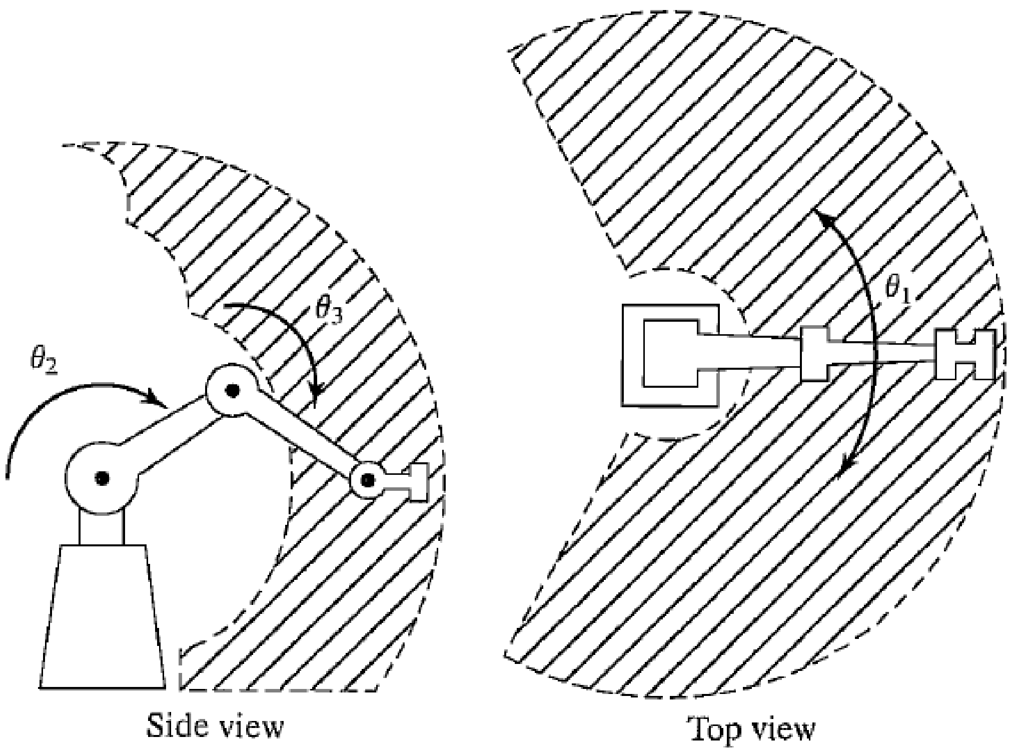


Figure 5.3.: An articulated manipulator with SCARA configuration. Adapted from Craig (2004).

5.1. Movement With Steady-Velocity

Without continuously modulated movement control in a feedback loop, which guarantees nearly-steady velocity, the observed velocity of servo-motors would change over time due to external disturbances (for example environmental disturbances or production flaws for the motors). When using two servomotors with same specs, using the same input for both servomotors does not guarantee same velocity of the motors. Control loop feedback mechanisms like proportional-integral-derivative (PID) control, calculate the error that emerged due to these disturbances and continuously adapt the input control signal to keep the output velocity steady. For the robot arm used for this thesis the nearly-steady velocity is handled by the *Maestro servo controller* from Pololu by its design. It allows controlling up to 6 servomotors, with speed and acceleration control along with the position control.

Normally, in order to achieve a steady velocity movement of the end effector, dynamics equations are used. With the help of dynamics equations, each joint must then be applied with a different torque. Since the movements of the robot arm are executed sequentially, dynamics equations for each servomotor movement are calculated separately. This is automatically handled by the servo controller, allowing to achieve nearly-steady velocity movement of the end effector of the robot arm.

5.2. Smooth Movements

It is important to have smooth movements of the end effector in applications which require high sensitivity operations such as painting. Even though for Pocket Code tests such a high sensitivity operations are not currently needed, a basic level of stability is still required in order to get stable sensor measurements and prevent shaking of end effector due to the weight of the mobile device when resting or moving. Smooth movements are achieved by movement of the joints with a function of time, where the time needed to start and finish the movement for each joint to achieve a certain position and orientation in 3d space is the same. The servo controller has the capability of controlling different servomotors with constant duration of time in order to achieve different angular positions and configuring the speed and acceleration for each motor separately. This made it possible to achieve smooth movements without so many calculations.

5.3. Design

A robot arm was initially designed as a 3d object using AutoCAD, but later on it was decided to make a change in the design in order to achieve joint movement using singular servomotors instead of double servomotors for each joint, since servomotors with enough torque was available. The initial design can be seen in the following figures5.4 and 5.5. For the mechanics of the robot arm, the following books have been used as a reference and source: Craig (2004), Iovine (2004). The AutoCAD tool is used to make technical drawings for the robot arm, specifically the robot arm joints to be cut from a 3mm aluminum plate. The aluminum is later laser cut as can be seen in the following figure 5.8 and bent to form joint as shown on figure 5.9.

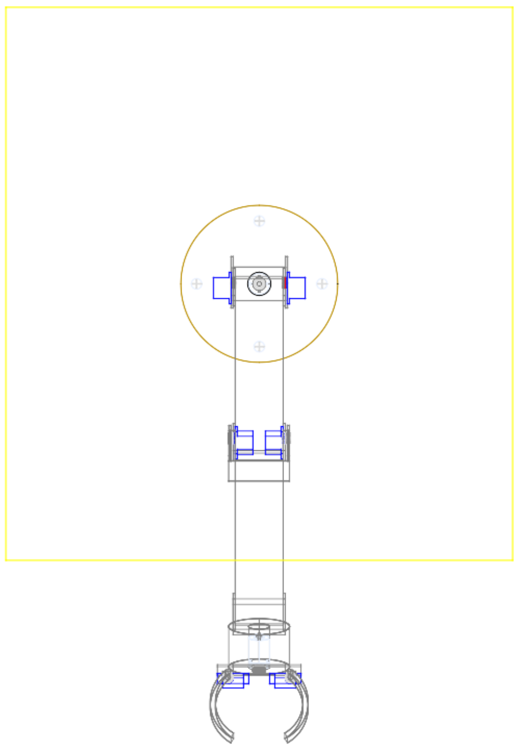


Figure 5.4.: Initial 2D design for the robot arm using AutoCAD.

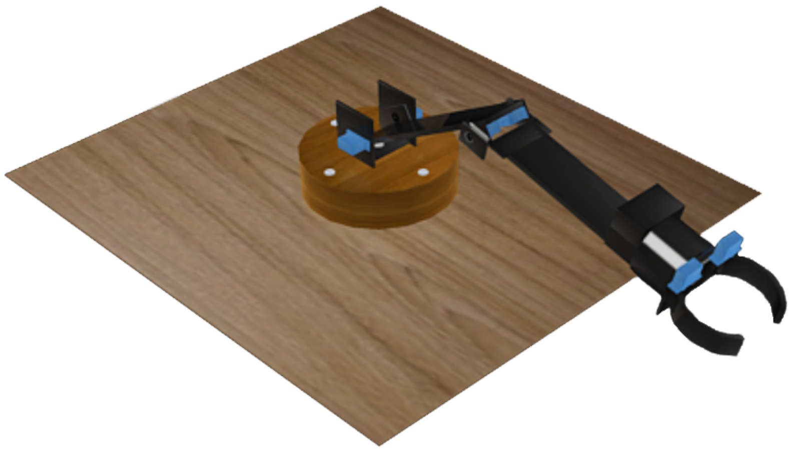


Figure 5.5.: Initial 3D design for the robot arm using AutoCAD.

The robot arm uses 4 servomotors, three of them being Power HD Ultra-High-Torque, High-Voltage Digital Giant Servo HD-1235MG models from Pololu with stall torque of 40 kg.cm at highest voltage strong enough to move the arm with a mobile device attached and one smaller servomotor holding the mobile phone holder with stall torque of 4 kg.cm. All four servomotors are connected to the Micro Maestro 6-Channel USB Servo Controller. The servo controller was initially powered by a Turnigy LiPo Battery, that was later substituted with an adapter to enable using the robot arm uninterrupted. A Raspberry Pi 2 was used as the control unit for the robot arm. The Raspberry Pi 2 was chosen because it was available, but another single board computer or personal computer could serve the same purpose.

The complete test setup consists of the Raspberry Pi, the servo controller, the robot arm, a mobile device with Pocket Code, where the tests are run and a computer, that installs the tests, initiates their execution and gathers the test results as shown on figure 5.7. The computer is connected to the mobile device via USB. The mobile device is held in the test bed of the robot arm and communicates with Raspberry Pi via WLAN. Raspberry Pi is connected to the servo controller via USB. The servomotors and their power supply are connected to the servo controller as shown on the figure 5.6. Both Raspberry Pi and the servo controller require their own power supply. The final version of the robot arm is shown on figure 5.10.

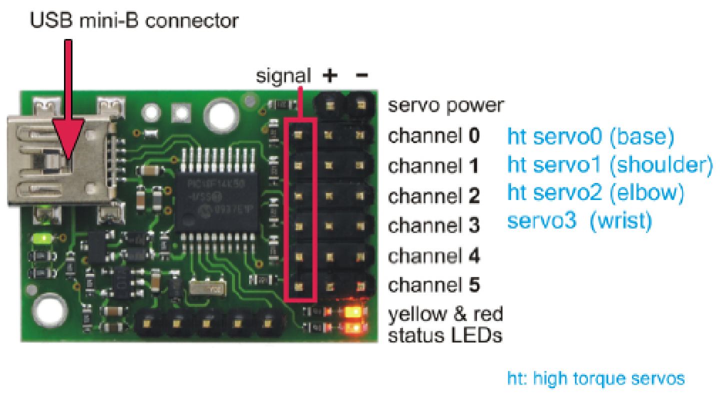


Figure 5.6.: Wiring of the servomotors to the servo controller. The servomotors are connected to the first four PINS of the servo controller.

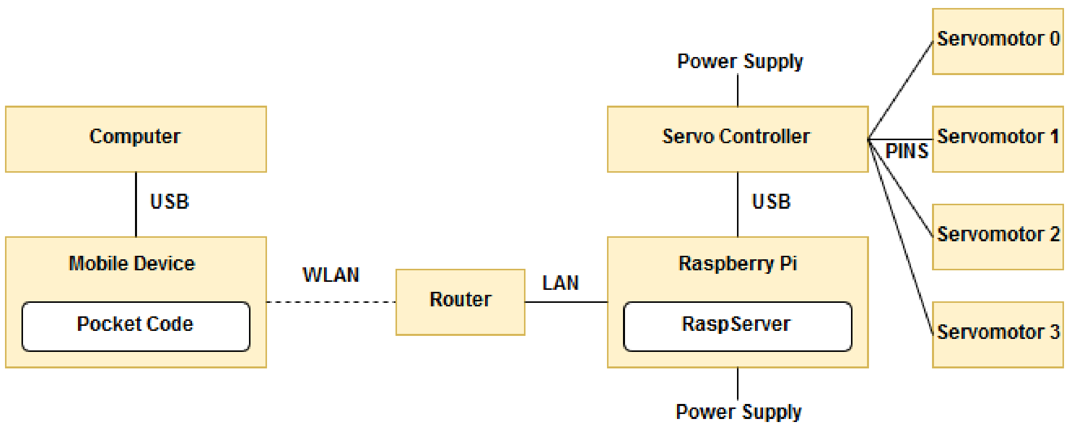


Figure 5.7.: The diagram of the full test setup required when using the robot arm.

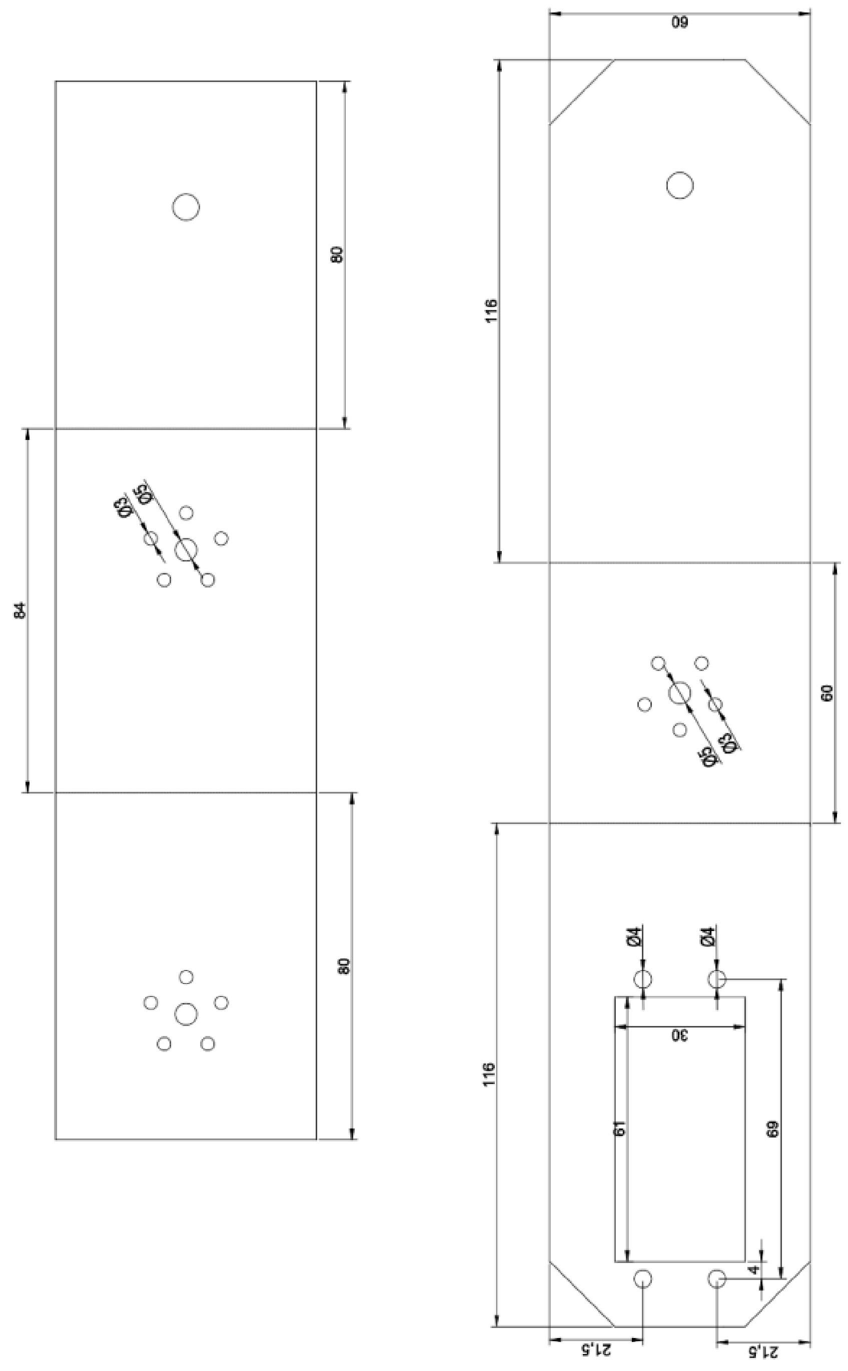


Figure 5.8.: Laser cut drawings for the robot arm on 2d plane, to be later on laser-cut on a 3mm aluminum plate and to be bent along the specified lines using a press and bending machine.

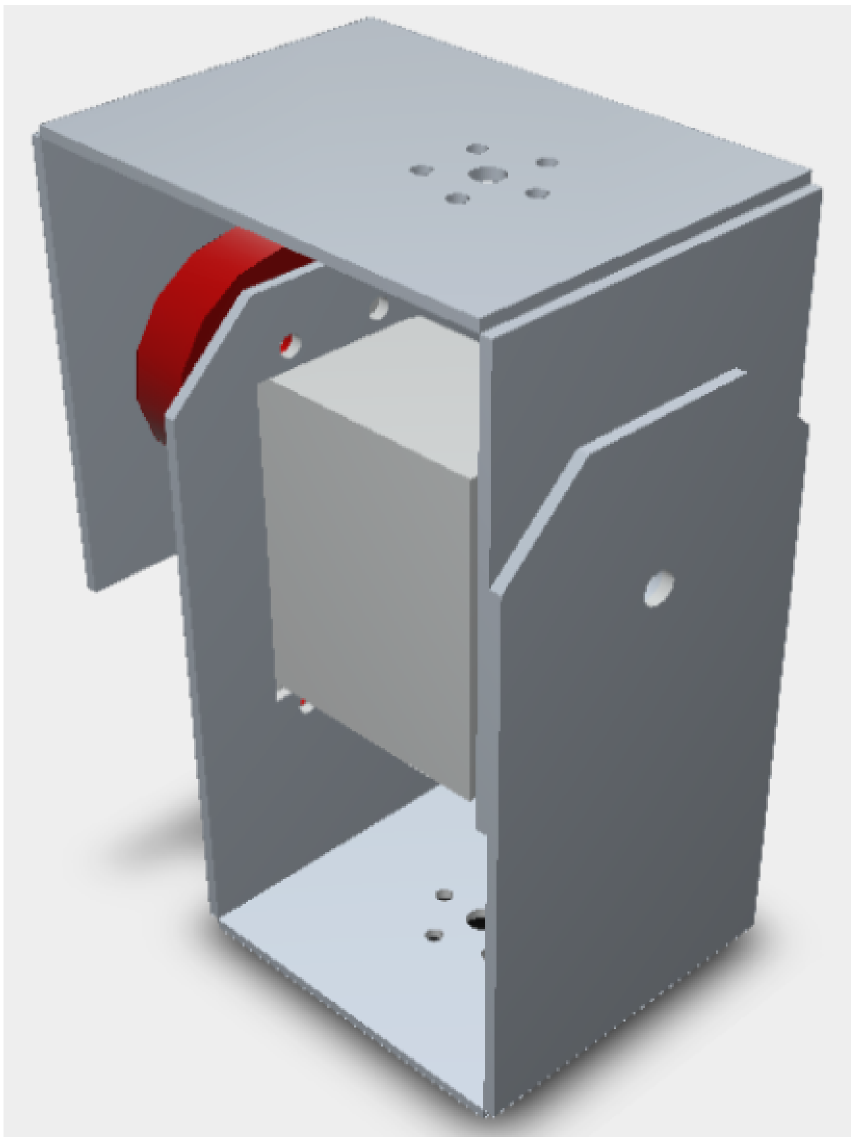


Figure 5.9.: Final 3D design of the robot arm joints using AutoCAD.

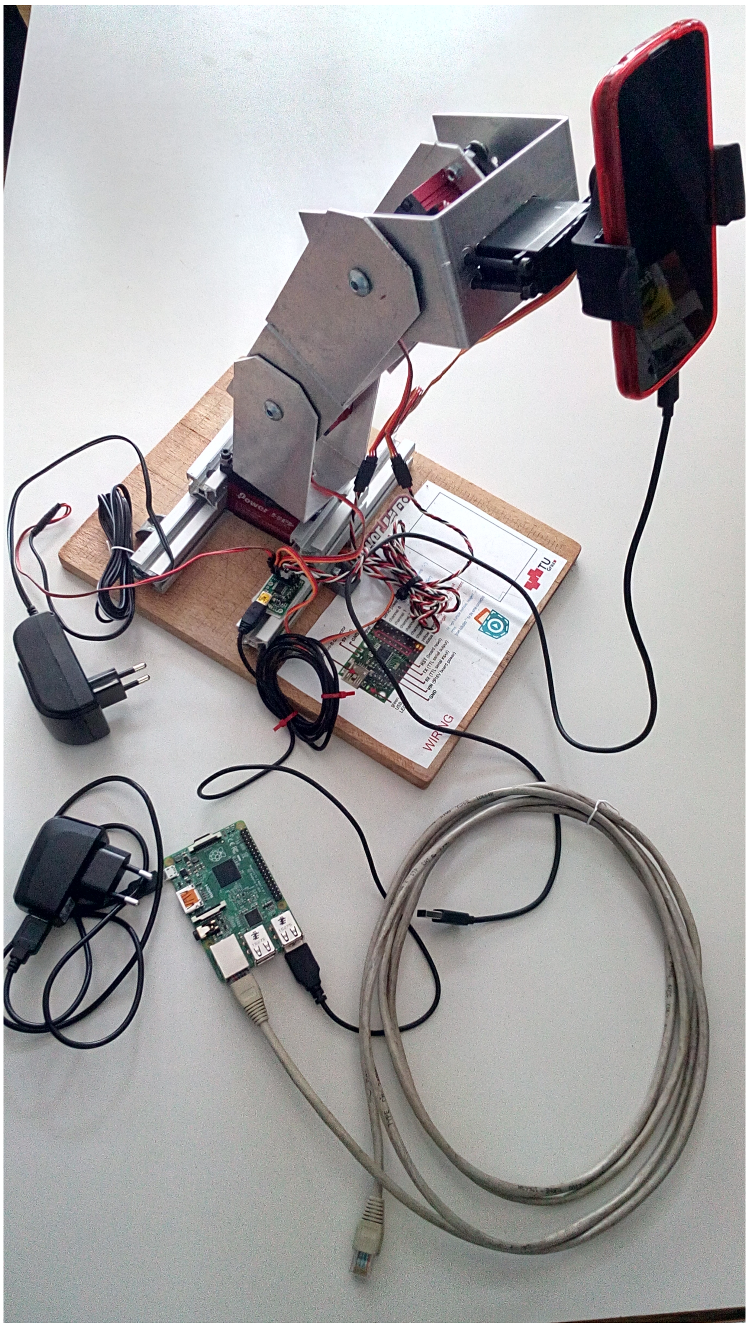


Figure 5.10.: The robot arm, with a Raspberry Pi acting as a server communicating with the actual servo controller controlling the robot arm's servomotors.

5.4. Installing Maestro Servo Controller Software

To use the Micro Maestro 6-Channel USB Servo Controller additional software from Pololu is required to be installed on the Raspberry Pi unit. The mentioned software is openly available on the webpages of Pololu. However, additional packages must be installed beforehand on the Raspberry Pi:

- libusb-1.0-0-dev
- mono-runtime
- libmono-winforms2.0-cil

After that the 99-pololu.rules file contained in the installation software must be copied to /etc/udev/rules.d/ in order to grant permission to use the Pololu USB devices. Finally the remaining files from the Maestro Servo Controller Linux Software archive are copied into a folder that will later contain all software for controlling the robot arm. With this setup the robot arm can be manually controlled from the Raspberry Pi using the UscCmd program (one of the files copied). The UscCmd program is used to give commands to the servo controller and through it to control the servomotors connected to the servo controller.

6. Implementation

The software in Catrobat is being developed in a Test-Driven way. Test driven development as well as behavior driven development place great emphasis on code coverage by tests or executable specifications in case of BDD. Therefore, it is important to be able to run and test all implemented features in the test environment. However, this does not currently apply to the features that work with the sensor data of the device on which Catrobat software is running.

Tests that require changing the values in sensors measuring, for example, the inclination or acceleration of the device (later just physical tests) could be only done manually before using the robot arm. In practice, this means that these tests are only performed if one of the features that directly handles the device sensors is changed. Thus, physical tests are never performed as part of regression testing. As a result, the correctness and persistent code quality of these features cannot be guaranteed and the most likely way of discovering regression of this code is based on negative customer feedback.

By introducing automated physical tests, they could be included in regression testing and by regularly testing used to discover potential problems as soon as they are introduced into the system, reducing the cost of finding the problem and repairing the bug as mentioned in the previous chapter.

6.1. Introduction of BDD into Pocket Code

Pocket Code for Android is being developed in Java. The user interface tests are performed using the Espresso framework developed by Google[1]. Therefore, a tool compatible with Espresso is needed to implement the BDD into an Android project that meets the following two requirements: i.) easy implementation and incorporation into an existing Android project; ii.) the

[1]https://developer.android.com/reference/android/support/test/espresso/Espresso

use of the Gherkin language for compatibility in case of a later tool change.

BDD tools for Java include:

- JBehave
- JDave
- Easyb
- Concordion
- FitNesse
- Cucumber-JVM

All of these tools use an open license. The drawback of Easyb is that it does not offer reporting. Another disadvantage is that Easyb and also Concordion do not support IDE integration. FitNesse, Concordion, and JDave also favor their own way of defining requirements instead of using Gherkin.

Considering these requirements the most suitable tools are JBehave and Cucumber-JVM. Cucumber-JVM has an easy setup and thus integration into the project and offers an easier learning curve. Compared to JBehave, Cucumber offers more versatile reporting capabilities, helping to create a living self-sustaining product documentation. Based on these factors, Cucumber was chosen as thetool to implement BDD into Pocket Code.

During the inclusion of Cucumber into Pocket Code two problems were encountered. They were caused by external factors. The original set of Cucumber libraries added to Pocket Code, contained the following library versions:

- Cucumber-Android (ver 1.2.0),
- Cucumber-picocontainer (ver 1.2.0),
- Cucumber-java (ver 1.2.0),
- Cucumber-jvm-deps (ver 1.0.3) and
- Gherkin (ver 2.12.2).

The first problem was that the chosen set of Cucumber libraries seemed to be incompatible and led to system crashes at the beginning of the execution of any Cucumber tests. Other configurations, like libraries Cucumber-java, Cucumber-junit, and Cucumber-picocontainer or Cucumber-android, Cucumber-picocontainer, and Cucumber-junit libraries, have also shown similar behavior. Also the use of newer or older versions of these libraries did not solve the crashes. The update of Pocket Code to version 0.9.41 revealed that these issues were caused by the ongoing changes in Pocket Code that preceded the update.

The version of the Android Studio IDE used to develop Pocket Code in the time of including of Cucumber to Pocket Code was the cause of the second problem. It manifested as Cucumber crashes after changing the test instrumentation class in Gradle, without making any other changes in the system since the last test run. Android Studio did not recognize this change as relevant enough to update the testing environment on the mobile device used for testing. This led to incompatibility resulting in crash of Cucumber at the beginning of every test. These two problems led to a decision to only include the minimum functionality required to run Cucumber to minimize potential causes for crashes as the source of the Android Studio problem was found only in late stages of this project.

As a minimum to include Cucumber into an Android project, it is necessary to add dependencies to two of Cucumber's libraries: Cucumber-Android and the Cucumber-Picocontainer, which handles the dependency injection allowing the sharing of state information between steps in scenarios. After adding these libraries, a custom instrumentation runner for the instrumented tests is needed.

```
@CucumberOptions(features = "features",
        glue = {"org.catrobat.catroid.test.cucumber
             .stepdefinitions"}
)
public class Instrumentation extends MonitoringInstrumentation {
    private final CucumberInstrumentationCore
       instrumentationCore =
         new CucumberInstrumentationCore(this);
    @Override
    public void onCreate(Bundle arguments) {
        super.onCreate(arguments);
        instrumentationCore.create(arguments);
        start();
    }
    @Override
    public void onStart() {
        super.onStart();
        waitForIdleSync();
        instrumentationCore.start();
    }
}
```

Listing 6.1: Step Instrumentation class for Cucumber

This new instrumentation runner combines feature files located in the features folder in the androidTest assets with step definitions, allowing the Cucumber to compile scenario and glue code tests, making features executable specifications. The resulting file structure is displayed on figure 6.1 with the instrumentation class being one step higher than all cucumber java files.

With Cucumber included in the system scenarios can be implemented to test if Cucumber is working correctly. A simple scenario and a scenario outline with basic tasks in the Pocket Code formula editor is implemented in the CucumberTest.feature in the features package. The implementation is shown in listing 6.2.

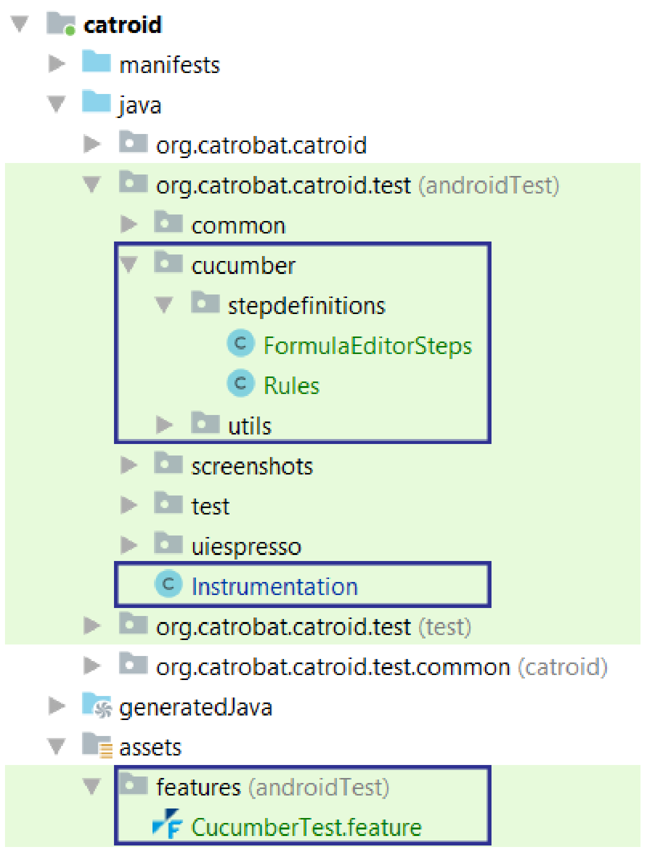


Figure 6.1.: Structure of Cucumber files in Pocket Code.

```
Feature: Temporary tests
Scenario: Creating a variable in formula editor
    Given I have an sprite activity in formula editor
    And I am in the data fragment
    When I create a variable
     Then I see that variable in the list
Scenario Outline: Using Calculator buttons in formula editor
    Given I have an sprite activity in formula editor
    When I press button <button>
     Then value <value> should show up in the edit text
    Examples:
    | button | value |
    | 0      | "0"   |
    | 1      | "1"   |
    | 2      | "2"   |
    | 3      | "3"   |
```

Listing 6.2: The CucumberTest.feature feature file

The scenario describes creating a variable in the formula editor while the scenario outline describes one of the basic functions of the calculator that is part of the formula editor as shown in listing 6.3. This is then implemented in the glue code in FormulaEditorSteps step definition file using Espresso encapsulated in the Cucumber framework.

```
@Given("^I have an sprite activity in formula editor$")
public void I_have_an_sprite_activity() {
    Rules.rule = spriteActivityTestRule;
    Script script = BrickTestUtils
 .createProjectAndGetStartScript("FormulaEditorAddVariableTest")
    ;
    script.addBrick(new ChangeSizeByNBrick(0));
    Rules.rule.launchActivity(null);
    onView(withId(R.id.brick_change_size_by_edit_text))
            .perform(click());
}
@Given("^I am in the data fragment$")
public void I_am_in_the_data_fragment() {
    onFormulaEditor()
            .performOpenDataFragment();
}
@When("^I create a variable$")
public void I_create_an_variable () {
    onDataList()
            .performAdd(varName);
}
@When("^I press button (\\d+)$")
public void I_press_button (int arg1) {
    onFormulaEditor().performEnterNumber(arg1);
}
@Then("^value \"([^\"]*)\" should show up in the edit text$")
public void value_should_show_up_in_the_edit_text (String arg1)
   {
    onFormulaEditor().checkShows(arg1 + " ");
}
@Then("^I see that variable in the list$")
public void I_see_that_variable () {
    onDataList().onVariableAtPosition(0)
            .checkHasName(varName);
}
```

Listing 6.3: The step definitions in FormulaEditorSteps

The last file shown in the figure 6.1 is the *Rules.java* file, that currently serves only to solve the following problem with the interworking between Espresso and Cucumber. When using Cucumber at the end of a scenario the currently running activity must be ended manually to properly start another scenario. Therefore the *Rules* class holds the Espresso rule of the current activity to call the *finish()* method.

```
@After
public void tearDown() throws Exception {
    if(isRobotArmtest){
        RobotArmControl.resetRobotArm();
        isRobotArmtest = false;
    }
    rule.getActivity().finish();
}
```

Listing 6.4: Manual finishing of activities in Rules class

6.2. Pocket Code Robot Arm Interface

The last step required to perform robot arm tests is the establishment of connection and communication between the system performing the Pocket Code tests and the robot arm. The connection is established via WLAN using a two-tier client-server model. Client-server model is an architecture in which processing is divided into 2 or more processes. One or more clients require services and/or resources and the server supplies them. For this project, the Raspberry Pi acts as a server and allows the client (mobile device with Pocket Code where the tests are run) to send commands to the servo controller of the robot arm.

6.2.1. Client Side

Since the Raspberry Pi and the mobile device used for physical tests are connected to a private WLAN, there is no need to implement any specific web services. A primitive socket connection over TCP/IP is enough to establish communication between the mobile device and Raspberry Pi. Four methods were added to simplify the use of the robot arm when writing

BDD tests on client side. A method opens and a method closes a connection to the Python program acting as a server on the Raspberry Pi. Two more methods are available that allow to use this connection and to send requests:

- public boolean resetRobotArm()
- public boolean sendRobotArmCommand(RobotArmCommands command, int value)

Both methods return a boolean value based on whether they have received confirmation from the server that the command has been processed and sent to the servo controller. Calling the resetRobotArm method sends a reset command to run a subroutine on the server that returns the robot arm to the default state with a series of commands. The sendRobotArmCommand method has two parameters. The first parameter is one of the five robot arm commands defined in the *enum* RobotArmCommands:

- SET_ACCELERATION
- SET_SPEED
- MOVE_ARM
- MOVE_BASE
- MOVE_HEAD

The second parameter is an integer value for the robot arm command, for instance the angle at which the specified part of the robot arm is to be rotated. The SET_ACCELERATION and SET_SPEED commands change the acceleration and speed parameter on each servomotor. The default value for these parameters is 10.

The MOVE_ARM, MOVE_BASE and MOVE_HEAD commands change the rotation of specific servomotors. For these commands the second parameter represents desired position in degrees with a rotation-resolution of 1 degree. This parameter is then transformed on the server side into a value in the movement range of the servomotors. MOVE_BASE allows rotation of the robot arm on the x-y-axis in 90 degree range. The MOVE_ARM command controls two servomotors that determine the angle between the joints of the robot arm, each of which can rotate one joint in a 90 degree range, creating a combined 180 degree range on the x-z-axis. The last command, MOVE_HEAD, changes the rotation of the servomotor holding the mobile phone holder in a 90 degree range on the y-z-axis.

As the default/resting state of the robot arm, a state was chosen in which all joints are in a vertical position so that the forces acting on the axes and the servomotor bearings are best distributed as shown in figure 6.3. In this way, the load on the servomotors by the weight of the robot arm parts they carry is minimized, which increases their lifetime as shown in figure 6.2. This position of the servomotors is referred to as position 0, and thus any MOVE command with parameter 0 returns the respective part of the robot arm to the default position. All other positions are counted as a deviation from the default position given in degrees. Therefore, the range parameter of the MOVE_BASE command is between -45 and 45, and for the MOVE_ARM command it is -90 to 90. The MOVE_Head command works differently as it allows to move the mobile device between horizontal and vertical positions. As such the default position 0 is the vertical position of the mobile device and the range is shifted to 0 to 90, with 90 representing horizontal position of the mobile device.

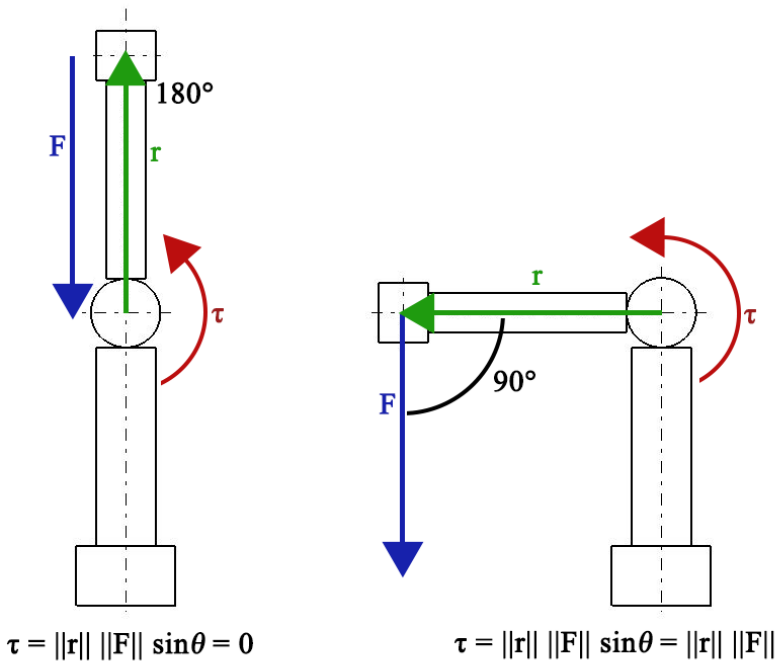


Figure 6.2.: The torque affecting the robot arm when in default position is 0, minimizing the load on the servomotors.

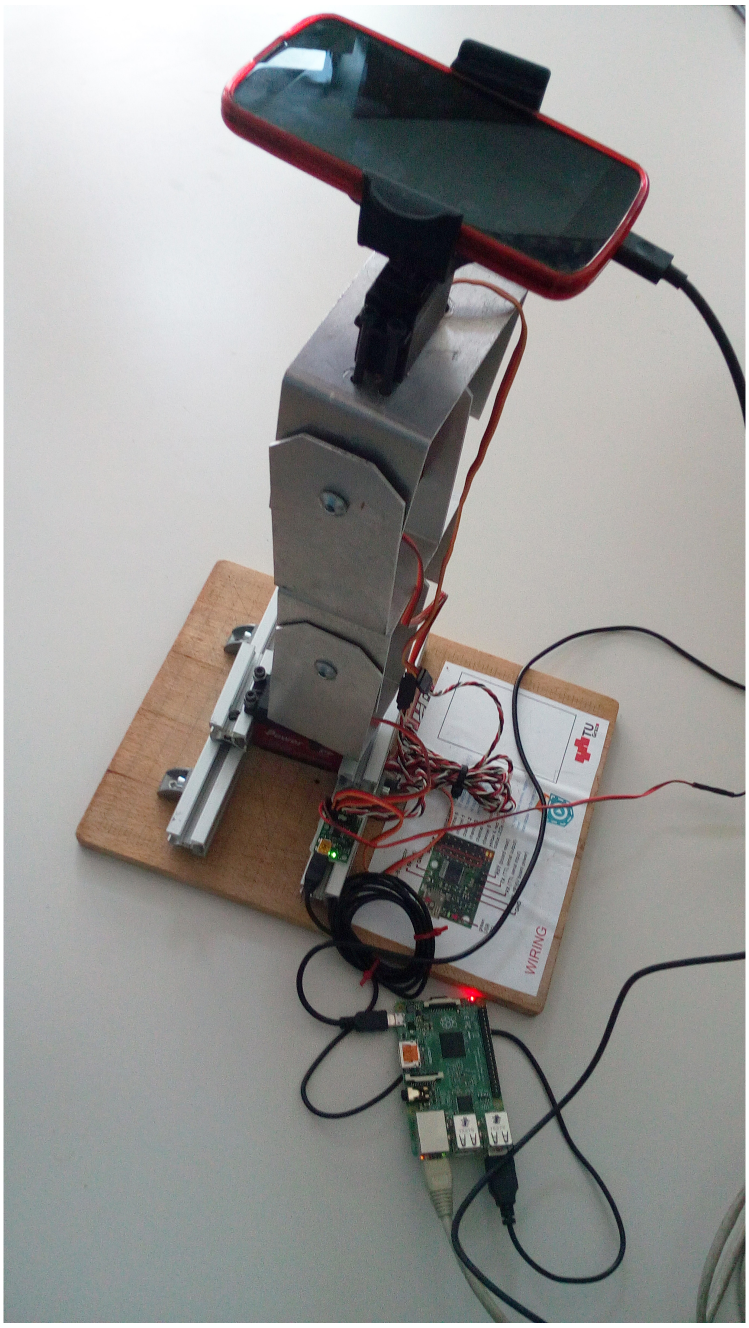


Figure 6.3.: The Robot Arm in default position pointing upwards to minimize the strain on the servomotors that hold it in position.

6.2.2. Parameter Limitations

To ensure the safety of the robot arm, when entering an invalid parameter in the sendRobotArmCommand method, this parameter is overwritten to the nearest valid parameter. For MOVE commands, this means that if the maximum or minimum angle has been exceeded, the servomotors will be commanded to move to the maximum or minimum valid value respectively (-45 or 45 for MOVE_BASE and MOVE_HEAD, and -90 or 90 for MOVE_ARM). Setting the SET_ACCELERATION and SET_SPEED to 0 is understood by the servo controller as setting it to the highest achievable value, which, depending on the power supply, can lead to dangerous velocity, therefore when the acceleration or speed is set to less than 0, the value parameter is overwritten with 1 instead, setting the servomotor to lowest acceleration or speed. The maximum values for acceleration and speed are not explicitly set, as the power supply limits the performance of the servomotors already.

6.2.3. Server Side

The server side is controlled by the RaspServer, a Python program. Full code of the program can be found in the Appendix A in listing A.1. It is a simple single-thread program created to receive plaintext commands from a mobile device via WLAN and to translate them into a form understood by the servo controller. The program is waiting for the client's socket connection on port 12000. When it receives a message from the client with up to two parameters (one parameter in case of reset command, two parameters otherwise), it uses the first parameter to recognise the requested action and translates the second parameter accordingly. RaspServer then calls the UscCmd program (part of the Servo Controller software) using the second parameter as TARGET parameter for UscCmd. Three servo controller commands are relevant for controlling the robot arm:

- –servo NUM,TARGET
- –speed NUM,TARGET
- –accel NUM,TARGET

These commands set angle, speed, and acceleration respectively, for a servomotor on one of the six channels of the servo controller according to the NUM parameter (0 to 5) to the value specified by the TARGET parameter. The servomotor position variable is an integer value in the movement range of the servomotor, which may vary depending on the servomotor brand and is obtained at the initial calibration of that servomotor. The movement range of the servomotors in the robot arm used in this project is between 3968 and 7932 representing 90 degree of movement. The resolution of 0.023 degrees resulting from the 3964 different positions is not required for the hardware tests. A resolution of 1 degree is used instead. The servomotor holding the test bed is an exception as it is a different model and its movement range representing 90 degree is 4700 to 7932. Its maximum resolution 0.027 degrees is also substituted with a resolution of 1 degree.

The RaspServer program accepts one of the six commands with the correct amount of parameters that it can receive from the test system, otherwise it returns a negative response indicating that the execution of the sent command has been interrupted. Before the received command is processed, RaspServer first checks it and modifies it as mentioned concerning the safety parameter limitations. Subsequently, in the case of SET_ACCELERATION and SET_SPEED, RaspServer calls the UscCmd with the required value for each of the servomotors.

In the case of a MOVE command, the parameter with the angle must be translated into a value in the above mentioned servomotor range. The adjusted value is then used as the TARGET in the UscCmd call (–servo NUM,TARGET) for all servomotors that must be moved to achieve the particular position. For MOVE_BASE and MOVE_HEAD, this is only one servomotor, but in MOVE_ARM, two servomotors that control the robot arm's joints move at the same time. The servomotor closer to the base moves only between 3 positions: default position and both edge positions it can reach. The servomotor closer to the head (test bed where the mobile is mounted) then ensures that the robot arm reaches the desired angle. In this way, the servomotor carrying the greater load spends more time in the default position where this load is best distributed. When their combined movement range is represented in an interval of -90 degree to 90 degree with 0 being the default position, all possible combinations of their positions can be divided into three groups as shown in figure 6.4:

1. The angle of robot arm to its default position is between 90 and 45 degree. The joint servomotor closer to the base is in its edge position

of 45 degrees while the joint servomotor closer to the head is in a position between 0 to 45 degree to reach the desired angle.
2. The angle of the robot arm to its default position is between 45 and -45 degree. The joint servomotor closer to the base rests in the default position while the servomotor closer to the head moves the upper joint to required angle.
3. The angle of robot arm to its default position is between -45 and -90 degree. The joint servomotor closer to the base is in its edge position of -45 degrees while the joint servomotor closer to the head is in a position between 0 to -45 degree to reach the desired angle.

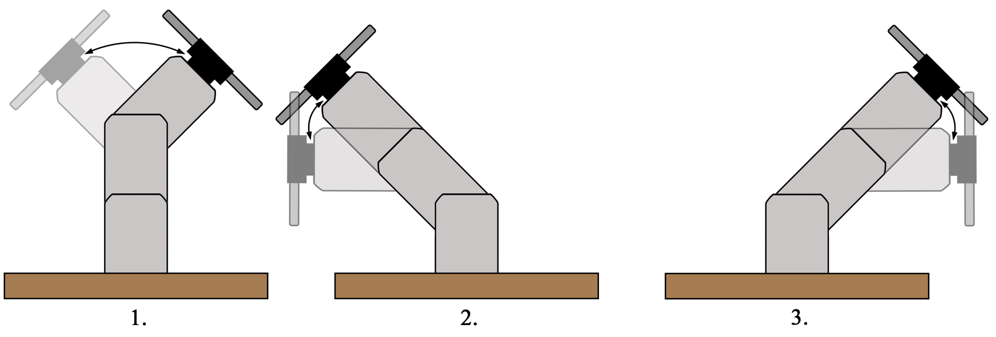


Figure 6.4.: Movement of the joints of the robot arm. The lower joint can move only in one of three positions while the position of the upper joint achieves the desired angle.

When the *resetRobotArm* method is called, a couple of UscCmd commands are run in the following sequence:

1. The acceleration and speed values of all servomotors are set to the value of 10.
2. A command is sent to all servomotors to return to the default position.
    a) The default position of the servomotor at the base of the robot arm and the servomotors at the robot arm joints is in the middle of their movement range at the value 5950.
    b) The default position of the servomotor holding the test bed is at the lowest value in its movement range (4700) representing vertical position of the mobile device held by the test bed.

The UscCmd program also contains the –status command for reporting the state of all connected servomotors, but this command only reports the position values estimated by the Servo Controller based on previous commands, and thus does not offer credible information about the true position of the servomotors. Therefore, the status command cannot be used to check that the servomotors have actually moved to the desired position and thus the positive response from the RaspServer that the client receives means that the commands have been successfully received and sent to the servo controller, not that the servomotors have successfully performed the required task.

7. Results and Evaluation

With a functional robot arm and the ability to control the robot arm directly from the test environment, tests that rely on the robot arm's capabilities can be performed using the Cucumber framework. In this chapter, an example feature is shown, describing the use of the mobile device's inclination sensor readings. Therefore Pocket Code is also implicitly tested if it correctly obtains and handles these sensor readings. The following sections are an evaluation of the use of Cucumber and BDD for such tests in the Catrobat project.

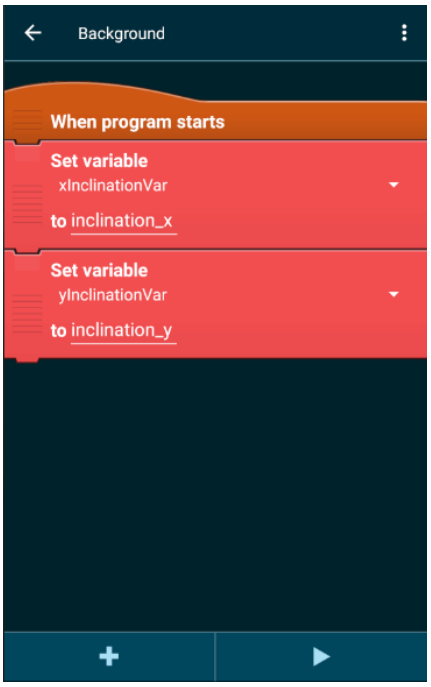
7.1. Inclination Sensor Feature

Using the robot arm control methods described in the Implementation chapter, it is possible to test the inclination sensor measurements if it behaves as expected. Listing 7.1 shows a simple Cucumber file containing one scenario outline describing an example of using the inclination sensors in Pocket Code.

```
Feature: Inclination Sensors values are readable from Formula
    Editor.

  The inclination_x and inclination_y values accessible from
  device specific variables section in formula editor should
  contain measured values of the inclination sensors. These
  values should update continuously while the stage
  activity is active.

  Scenario Outline: Pocket code correctly reads inclination values
      of the device.
    Given I have a stage activity for inclination tests
    When I rotate the phone <x_angle> degree on the x-axis
    And I rotate the phone <y_angle> degree on the y-axis
    Then I should see inclination values approximately <x_angle>
        and <y_angle>

  Examples:
    | x_angle | y_angle |
    |       0 |       0 |
    |      90 |       0 |
    |     -45 |       0 |
    |       0 |      45 |
    |       0 |     -20 |
    |      60 |      30 |
    |      30 |     -45 |
```

Listing 7.1: InclinationSensors feature file

Since the measured inclination values may not be completely accurate, either due to inaccuracy of the inclination sensors or to environmental influences (workspace unevenness, slight shaking of the device or other reasons), the tests performed on the robot arm do not compare exact values, but check whether the values correspond to approximate intervals around the expected value and whether the value changes are proportional to the changes in the position and inclination of the device held in the mobile phone holder of the robot arm. The test is run within the following Pocket Code program on a mobile phone shown in figure 7.1. It uses two variables, xInclinationVar and yInclinationVar, which are used to store the measured values of the inclination sensors.



Set variable xInclinationVar to inclination_x

Set variable yInclinationVar to inclination_y

Figure 7.1.: Program ran on Pocket Code during the inclination test.

The robot arm is initially in the default position from which it moves to the position specified by x_angle and y_angle variables given in the InclinationSensors feature file. The arm movement is performed using the MOVE_ARM and MOVE_HEAD commands respectively, described in more detail in section 6.2.1. The program then waits one second for the inclination sensors to stabilize and then compares the measurements with the expected values, with a tolerance of 15 degree.

```
@Given("^I have a stage activity for inclination tests$")
public void I_have_a_stage_activity () {
    Rules.rule = stageActivityTestRule;
    Rules.isRobotArmtest = true;
    createProject("\"BroadcastForClonesRegressionTest\"");
    Rules.rule.launchActivity(null);
}

@When("^I rotate the phone (\\d+) degree on the x-axis$")
public void I_rotate_the_phone_on_the_x_axis(int x_angle) throws
     IOException, InterruptedException {
    if(!sendRobotArmCommand(RobotArmControl.RobotArmCommands.
        MOVE_ARM, x_angle)) {
         fail();
    }
}

@When("^I rotate the phone (\\d+) degree on the y-axis$")
public void I_rotate_the_phone_on_the_y_axis(int y_angle) throws
     IOException, InterruptedException {
    if(!sendRobotArmCommand(RobotArmControl.RobotArmCommands.
        MOVE_HEAD, y_angle)) {
         fail();
    }
}

@Then("^the inclination values should be approximately (\\d+)
   and (\\d+)$")
public void the_inclination_values_should_be(int x_angle, int
   y_angle) throws InterruptedException {
    TimeUnit.SECONDS.sleep(1);
    double x_value = (double) xInclinationVar.getValue();
    assertTrue(x_value < (x_angle + 15) && x_value > (x_angle
        -15));
    double y_value = (double) yInclinationVar.getValue();
    assertTrue(y_value < (y_angle + 15) && y_value > (y_angle +
        15));
}
```

Listing 7.2: Step definitions for the InclinationSensors feature file

7.2. Evaluation of Using Cucumber

Working with Cucumber has shown several compatibility issues with the original system. Since Cucumber requires overwriting the instrumentation runner used with a Cucumber instrumentation runner, it is not possible to include current "pure Espresso" instrumented tests and executable features from BDD that are implemented via Cucumber and Espresso together in one test suite. In addition, if both systems were to be used at the same time, a Gradle sync would have to be run between running the Espresso tests and the Cucumber tests for the system to use the second instrumentation runner. But here the first severe problem shows up: Android Studio does not reinstall Pocket Code on the test device (mobile OR emulator) when changing the instrumentation runner, which means that the wrong instrumentation runner is used for the tests and therefore the whole testrun crashes. To fix this problem, the Pocket Code application must be manually removed from the device (or the emulator instance) before running the tests so that Android Studio will install the correct version of the testrunner. Therefore, if Cucumber were used, old tests would become unusable until rewritten as executable specification or a workaround would have to be found.

The second problem with using Cucumber compared to the current test system is the way Cucumber handles syntactically wrongly written tests. Cucumber feature files and their connection to the glue code are not controlled during the build and the occurrence of a bug often leads to a complete crash of the entire test suite, not just the test where the error occurred. In this case, the test run is interrupted and the system only provides a not very informative message *Test run failed: Instrumentation run failed due to „Process crashed.“*

Another remark to Cucumber is that Espresso and Cucumber were not developed to be compatible. As a result, some Espresso features that are automated when using "pure" Espresso must be manually re-added to the tests when also using Cucumber. An example of such a feature is the automatic teardown of an activity at the end of an Espresso test. When using Cucumber, the teardown must be manually initiated at the end of the test or the next test in the test suite will start to run on the activity that was not manually ended.

A large part of the Catrobat team is made up of students, often joining or leaving the team and the use of another layer of methods and tools makes it harder to enter the system. But what BDD brings for these new members of the development team is a clearer documentation in form of Cucumber feature files, allowing a faster understanding of underlying code.

In summary, BDD meets the original goal of platform independent testing, but the use of Cucumber in this form provides a potentially lower stability of the test system and requires solving of new problems that occur during writing tests and otherwise would not.

7.3. Discussion

The biggest weak point of the presented approach for implementing platform independent tests is the fragility that stems from combining Cucumber with Espresso. During the integration of Cucumber into the current system, many variants were rejected as nonfunctional for what later turned out to be caused by ongoing changes of Pocket Code and Android Studio. Looking back, several of those variants should be reevaluated, as they might solve the fragility, particularly the open source library Green Coffee, specifically created to combine Cucumber with Espresso. Because of the problems caused by Android Studio, many optional Cucumber features were omitted, that could be added to the system if the platform independent testing is to be done in presented way.

To continue the use of Cucumber would mean that either the whole test suite of Catrobat would have to be rewritten using the tool with feature files in Gherkin language and tests split into glue code, or that two different testing environments would be required, one for Cucumber tests and another for pure Espresso tests. Therefore to implement platform independent tests it would be better to find another tool that would be used insted of Cucumber.

Regarding the robot arm, the servomotors are currently handled by the servo controller from Pololu. Although it can show the servomotor status information including the position values of each servomotor connected, these values are not always correct. The status information stored in the servo controller is based on the previous commands issued to the servo controller regardless whether they were actually executed. Therefore probably the most interesting improvement of the robot arm would be the development of a different way to control the servomotors, which could reliably return the true current position of the servomotors.

With the use of the robot arm further additional tests can be implemented. In addition to testing the functionality of the inclination or acceleration sensors it is easily possible to add tests for the mobile device's camera, and advanced Pocket Code functionality such as the face recognition. For testing the face recognition a picture or a photo with a face can be placed in front of the robot arm, and the mobile device is then moved so that the camera captures the image and then it is checked whether the face recognition system works properly. Similar simple would be the testing of the compass sensor. The robot arm would have to be adjusted to face in a certain cardinal direction and then be moved to another cardinal direction while checking the sensor readings for these directions. These kind of tests and the application of using the robot arm for so called hardware tests are completely independent of the inclusion of the BDD framework.

8. Summary

This thesis addresses two related problems. The first challenge was to create a system that allows testing the Catrobat Android app Pocket Code by manipulating the mobile device in a way that the hardware sensor readings change. Although the changing of the readings could be mocked a manipulation of the mobile device in space makes sense to manipulate the hardware sensor readings for real to ensure correct functionality. Therefore a controllable robot arm is needed which manipulates the mobile device where the tests are running on.

The second challenge was to ensure platform independence of such hardware sensor tests. Pocket Code is also developed for the iOS platform and the idea was to create such hardware tests only once and reuse them to a.) ensure that the same test code does not have to be developed for the other platform from scratch a second time, and b.) that the same tests are executed on both platforms ensuring that both systems behave the same in regard to the change of the sensor readings due to the mobile device movement in space.

Regarding the first challenge, to generally implement sensor tests, a robot arm capable of partially rotating on all three axes was needed and constructed. As an end effector, a mobile phone holder is used to securely mount a mobile device. In this way the changing of the sensor value readings can be achieved during automated testing by controlling the robot arm.

The robot arm is controlled using four servomotors. Two servomotors connect three joints of the robot arm, one servomotor is used to move the base of the robot arm, and one servomotor is mounted at the robot arm's end with a mobile phone holder attached to it. The servomotors of the robot arm are directly wired to and controlled by a servo controller that itself is connected to a Raspberry Pi computer. On the Raspberry Pi a simple python server program is running - the *RaspServer*. It handles the abstract movement commands which it receives via socket communication in plaintext from the tests which run on the mobile device. The RaspServer translates these abstract commands into "servo controller understandable" commands. The translated commands are then forwarded to the servo controller which in turn controls the servos on the robot arm. The test running on the mobile device can now check the changed sensor readings for plausibility and determine if the physical movement of the mobile device initiated by sending a command to the robot arm leads to the expected change in the hardware sensor readings and furthermore, if these sensor readings are correctly interpreted by the Pocket Code system. Due to its simple design and permanent power supply it is possible to use this robot arm for any tests which intend to react to changing hardware sensor readings. It is also easily possible to use this robot arm in combination with a continuous integration system like Jenkins for running such a hardware test suite on a regular basis.

Regarding the second challenge the goal was to achieve platform independence of such hardware tests especially for the Catrobat Pocket Code application for Android and iOS. The idea was to write hardware tests in an abstract reusable meta-language and use these tests on both platforms to have a common testbase. Since once in the history of the development of Pocket Code BDD using Cucumber already was successfully implemented it was the idea to try to do it again and integrate the BDD tool Cucumber. The reason why the early BDD tests vanished again from the codebase was a lack of maintenance and unsuccessful porting to other platforms. During the work of this thesis it was again pursued to integrate BDD again. Due to compatibility reasons and to keep the learning curve low tools were evaluated using the Gherkin language and finally Cucumber was chosen. Cucumber feature files serve as platform independent tests as well as documentation for each Pocket Code feature which is to be tested, but the use of Cucumber has led to many problems beside the very unstable behavior in combination with the Android Studio as well as the Espresso testing framework. Nevertheless a proof of concept was worked out which showed that a combination is possible but would make it necessary to rewrite the whole Espresso testing suite to be able to execute Espresso and Cucumber tests in one go. These experienced problems with the integration of Cucumber and the current test base are very valuable so now it can be focussed on finding other tools which are better suitable to be integrated in a more transparent way.

Appendix

Appendix A.

RaspServer Program

```
import os
import socket
from subprocess import Popen

cwd = os.getcwd()
uscLocation = cwd + "/UscCmd"


def init_cmd():
    Popen([uscLocation, '--accel', '0,10'])
    Popen([uscLocation, '--speed', '0,10'])
    Popen([uscLocation, '--accel', '1,10'])
    Popen([uscLocation, '--speed', '1,10'])
    Popen([uscLocation, '--accel', '2,10'])
    Popen([uscLocation, '--speed', '2,10'])
    Popen([uscLocation, '--accel', '3,10'])
    Popen([uscLocation, '--speed', '3,10'])

    Popen([uscLocation, '--servo', '0,5950'])
    Popen([uscLocation, '--servo', '1,5950'])
    Popen([uscLocation, '--servo', '2,5950'])
    Popen([uscLocation, '--servo', '3,4700'])


def accel_cmd(arg1):
    if arg1 < 1:
        arg1 = 1

    Popen([uscLocation, '--accel', '0,' + arg1])
    Popen([uscLocation, '--accel', '1,' + arg1])
    Popen([uscLocation, '--accel', '2,' + arg1])
    Popen([uscLocation, '--accel', '3,' + arg1])


def speed_cmd(arg1):
    if arg1 < 1:
        arg1 = 1

    Popen([uscLocation, '--speed', '0,' + arg1])
    Popen([uscLocation, '--speed', '1,' + arg1])
    Popen([uscLocation, '--speed', '2,' + arg1])
    Popen([uscLocation, '--speed', '3,' + arg1])


def rotate_cmd(arg1):
    arg1 = int(arg1)
    if arg1 < -45:
        arg1 = -45

    if arg1 > 45:
        arg1 = 45

    arg1 = change_scale(arg1)
    Popen([uscLocation, '--servo', '0,' + str(int(arg1))])


def rotate_head_cmd(arg1):
    arg1 = int(arg1)
    if arg1 < 0:
        arg1 = 0

    if arg1 > 90:
        arg1 = 90

    arg1 = change_scale_head(arg1)
    Popen([uscLocation, '--servo', '3,' + str(int(arg1))])


def tilt_cmd(arg1):
    arg1 = int(arg1)
    if arg1 < -90:
        arg1 = -90

    if arg1 > 90:
        arg1 = 90
    if arg1 < -45:
        Popen([uscLocation, '--servo', '1,3968'])
        arg1 += 45

    elif arg1 > 45:
        Popen([uscLocation, '--servo', '1,7932'])
        arg1 -= 45

    else:
        Popen([uscLocation, '--servo', '1,5950'])

    arg1 = change_scale(arg1)
    Popen([uscLocation, '--servo', '2,' + str(int(arg1))])


# change scale from (-45)-(45) to 3968-7932
def change_scale(arg1):
    arg1 += 45
    arg1 *= 3964
    arg1 /= 90
    arg1 += 3968
    return arg1


# change scale from 0-90 to 4700-7932
def change_scale_head(arg1):
    arg1 *= 3232
    arg1 /= 90
    arg1 += 4700
    return arg1

host = ''
port = 12000
s = socket.socket()

try:
    s.bind((host, port))
except socket.error as e:
    print(str(e))
    print('socket bind failed')

s.listen(1)
init_cmd()
```

```
print('Arm ready')

while True:
    conn, addr = s.accept()
    print('connected to: ' + addr[0] + ' : ' + str(addr[1]))
    try:
        data = conn.recv(1024).decode('UTF-8')
        cmd_args = data.split()
        print(cmd_args)

        if len(cmd_args) == 1:
            if cmd_args[0] == 'reset':
                init_cmd()
                conn.send("done\n".encode('UTF-8'))
            else:
                conn.send("error\n".encode('UTF-8'))

        elif len(cmd_args) == 2:
            try:
                val1 = int(cmd_args[1])
            except ValueError:
                conn.send("error\n".encode('UTF-8'))
                continue

            if cmd_args[0] == 'speed':
                speed_cmd(cmd_args[1])
                conn.send("done\n".encode('UTF-8'))

            elif cmd_args[0] == 'accel':
                accel_cmd(cmd_args[1])
                conn.send("done\n".encode('UTF-8'))

            elif cmd_args[0] == 'move_base':
                rotate_cmd(cmd_args[1])
                conn.send("done\n".encode('UTF-8'))

            elif cmd_args[0] == 'move_head':
                rotate_head_cmd(cmd_args[1])
                conn.send("done\n".encode('UTF-8'))

            elif cmd_args[0] == 'move_arm':
                tilt_cmd(cmd_args[1])
                conn.send("done\n".encode('UTF-8'))
            else:
                conn.send("error\n".encode('UTF-8'))

        else:
            conn.send("error\n".encode('UTF-8'))

    finally:
        conn.close()
        print('closed')
```

Listing A.1: The RaspServer program

Bibliography

Adzic, Gojko (2011). *Specification by Example: How Successful Teams Deliver the Right Software*. 1st. Greenwich, CT, USA: Manning Publications Co. ISBN: 1617290084, 9781617290084 (cit. on p. 23).

Alshamran, Adel and Abdullah Bahattab (2015). "A Comparison Between Three SDLC Models Waterfall Model, Spiral Model, and Incremental/Iterative Model." In: *International Journal of Computer Science Issues* 12.1, pp. 106–111 (cit. on p. 16).

Astels, Dave (2006). *A new look at test-driven development*. accessed 3rd April, 2019. URL: https://web.archive.org/web/20061206004208/http://blog.daveastels.com/files/BDD_Intro.pdf (cit. on p. 20).

Astels, David (2003). *Test Driven Development: A Practical Guide*. Prentice Hall Professional Technical Reference. ISBN: 0131016490 (cit. on pp. 17, 18).

Bau, David et al. (May 2017). "Learnable Programming: Blocks and Beyond." In: *Communications of the ACM* 60.6, pp. 72–80. ISSN: 0001-0782. DOI: 10.1145/3015455 (cit. on p. 5).

Beck, Kent (2002). *Test Driven Development: By Example*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc. ISBN: 0321146530 (cit. on p. 17).

Burnett, Margaret M. et al. (Mar. 1995). "Scaling up visual programming languages." In: *Computer* 28.3, pp. 45–54. ISSN: 0018-9162. DOI: 10.1109/2.366157 (cit. on p. 3).

Chelimsky, David et al. (2010). *The RSpec Book: Behaviour Driven Development with Rspec, Cucumber, and Friends*. 1st. Pragmatic Bookshelf. ISBN: 1934356379, 9781934356371 (cit. on p. 25).

Cohn, Mike (2009). *Succeeding with Agile: Software Development Using Scrum*. 1st. Addison-Wesley Professional. ISBN: 0321579364, 9780321579362 (cit. on p. 31).

Craig, John J. (2004). *Introduction to Robotics: Mechanics and Control*. 3rd. Pearson Education International. ISBN: 0201543613, 9780201543612 (cit. on pp. 33–37, 39).

Dees, Ian, Aslak Hellesøy, and Matt Wynne (2013). *Cucumber Recipes: Automate Anything with BDD Tools and Techniques*. The pragmatic programmers. Pragmatic Bookshelf. ISBN: 9781937785017 (cit. on p. 24).

Dijkstra, Edsger Wybe et al. (1970). *Notes on structured programming* (cit. on p. 13).

Evans, Eric (2003). *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley. ISBN: 0321125215 (cit. on p. 24).

Freeman, Steve and Nat Pryce (2009). *Growing Object-Oriented Software, Guided by Tests*. 1st. Addison-Wesley Professional. ISBN: 0321503627, 9780321503626 (cit. on pp. 17, 19).

Harzl, Annemarie et al. (2013). "Comparing Purely Visual with Hybrid Visual/Textual Manipulation of Complex Formula on Smartphones." English. In: *DMS 2013*. ., pp. 0–0 (cit. on p. 10).

Iovine, John (2004). *PIC Robotics: A Beginner's Guide to Robotics Projects Using the PIC Micro*. McGraw-Hill Education. ISBN: 0071373241, 9780071373241 (cit. on p. 39).

Ko, Andrew J., Brad A. Myers, and Htet H. Aung (Sept. 2004). "Six Learning Barriers in End-User Programming Systems." In: *2004 IEEE Symposium on Visual Languages - Human Centric Computing*, pp. 199–206. DOI: 10.1109/VLHCC.2004.47 (cit. on p. 5).

Koskela, Lasse (2007). *Test Driven: Practical Tdd and Acceptance Tdd for Java Developers*. Greenwich, CT, USA: Manning Publications Co. ISBN: 9781932394856 (cit. on pp. 17–19).

Kurihara, Azusa et al. (2015). "A Programming Environment for Visual Block-Based Domain-Specific Languages." In: *Procedia Computer Science* 62, pp. 287–296. DOI: 10.1016/j.procs.2015.08.452 (cit. on p. 4).

Larman, Craig and Victor Robert Basili (June 2003). "Iterative and incremental developments. a brief history." In: *Computer* 36.6, pp. 47–56. ISSN: 0018-9162. DOI: 10.1109/MC.2003.1204375 (cit. on p. 15).

Larman, Craig and Bas Vodde (2010). *Practices for scaling lean and agile development: large, multisite, and offshore product development with large-scale Scrum*. Addison-Wesley. ISBN: 0321636406 (cit. on p. 19).

Lazăr, Ioan, Simona Motogna, and Bazil Pârv (Aug. 2010). "Behaviour-Driven Development of Foundational UML Components." In: *Electronic Notes in Theoretical Computer Science* 264.1, pp. 91–105. DOI: 10.1016/j.entcs.2010.07.007 (cit. on p. 21).

Luhana, Kirshan Kumar, Christian Schindler, and Wolfgang Slany (June 2018). "Streamlining mobile app deployment with Jenkins and Fastlane in the case of Catrobat's Pocket Code." English. In: *2018 IEEE International Conference on Innovative Research and Development (ICIRD)*. IEEE Xplore, pp. 1–6. DOI: 10.1109/ICIRD.2018.8376296 (cit. on p. 8).

Maloney, John et al. (Nov. 2010). "The Scratch Programming Language and Environment." In: *Trans. Comput. Educ.* 10.4, 16:1–16:15. ISSN: 1946-6226. DOI: 10.1145/1868358.1868363. URL: http://doi.acm.org/10.1145/1868358.1868363 (cit. on p. 6).

Martin, Robert Cecil (2003). *Agile Software Development: Principles, Patterns, and Practices*. Upper Saddle River, NJ, USA: Prentice Hall PTR. ISBN: 0135974445 (cit. on p. 14).

North, Dan (2014). *BDDWiki: BehaviourDrivenDevelopment*. accessed 4th April, 2019. URL: https://behaviourdriven.org/ (cit. on p. 21).

North, Dan (2017). *Introducing BDD*. accessed 3rd April, 2019. URL: https://dannorth.net/introducing-bdd/ (cit. on pp. 21, 24).

North, Dan (Apr. 2019). *What's in a Story?* accessed 18th May, 2019. URL: https://dannorth.net/whats-in-a-story/ (cit. on pp. xi, 23, 24).

Parnas, David Lorge (May 1972). "A Technique for Software Module Specification with Examples." In: *Communications of the ACM* 15.5, pp. 330–336. ISSN: 0001-0782. DOI: 10.1145/355602.361309 (cit. on p. 22).

Patton, Ron (2000). *Software testing*. Indianapolis, IN, USA: Sams. ISBN: 0672319837 (cit. on p. 13).

Repenning, Alexander (July 2017). "Moving Beyond Syntax: Lessons from 20 Years of Blocks Programing in AgentSheets." In: *Journal of Visual Languages and Sentient Systems* 3.1, pp. 68–91. DOI: 10.18293/vlss2017-010 (cit. on p. 4).

Resnick, Mitchel et al. (Nov. 2009). *Scratch: Programming for All*. DOI: 10.1145/1592761.1592779 (cit. on p. 6).

Scratch Wiki (2019). accessed 17th May, 2019. URL: https://en.scratch-wiki.info/ (cit. on p. 7).

Slany, Wolfgang (2012a). "A mobile visual programming system for Android smartphones and tablets." English. In: *IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2012.* ., pp. 265–266 (cit. on p. 8).

Slany, Wolfgang (2012b). “Catroid: a mobile visual programming system for children.” English. In: *11th International Conference on Interaction Design and Children, IDC '12*. ., pp. 300–303 (cit. on p. 3).

Slany, Wolfgang et al. (Aug. 2018). “Rock Bottom, the World, the Sky: Catrobat, an Extremely Large-scale and Long-term Visual Coding Project Relying Purely on Smartphones.” English. In: *Constructionism 2018, Vilnius*. Ed. by Valentina Dagienė and Eglė Jasutė, pp. 104–119 (cit. on pp. 8, 10).

Smart, John Ferguson (2014). *BDD in Action: Behavior-Driven Development for the Whole Software Lifecycle*. Manning Publications. ISBN: 9781617291654 (cit. on p. 21).

Solis, Carlos and Xiaofeng Wang (Aug. 2011). “A Study of the Characteristics of Behaviour Driven Development.” In: *2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications*, pp. 383–387. DOI: 10.1109/SEAA.2011.76 (cit. on p. 22).

Stoica, Marian, Marinela Mircea, and Bogdan Ghilic-Micu (Dec. 2013). “Software Development: Agile vs. Traditional.” In: *Informatica Economica* 17, pp. 64–76. DOI: 10.12948/issn14531305/17.4.2013.06 (cit. on p. 15).

Wah, Benjamin Wan-Sang (2007). *Wiley Encyclopedia of Computer Science and Engineering*. New York, NY, USA: Wiley-Interscience. ISBN: 0470107928 (cit. on p. 3).

Weintrop, David and Uri Wilensky (2015). “To Block or Not to Block, That is the Question: Students' Perceptions of Blocks-based Programming.” In: *Proceedings of the 14th International Conference on Interaction Design and Children*. IDC '15. Boston, Massachusetts: ACM, pp. 199–208. ISBN: 978-1-4503-3590-4. DOI: 10.1145/2771839.2771860 (cit. on p. 4).

Wynne, Matt and Aslak Hellesøy (2012). *The cucumber book: behaviour-driven development for testers and developers*. Pragmatic Bookshelf. ISBN: 9781934356807 (cit. on p. 13).