Manuel Wallner, BSc

Specification by Example of the Broadcast Mechanism of Catrobat

Master's Thesis

Graz University of Technology

Institute for Softwaretechnology

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

Graz, August 2014

Deutsche Fassung:
Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008
Genehmigung des Senates am 1.12.2008

EIDESSTATTLICHE ERKLÄRUNG

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

Graz, am …… ……..
(Unterschrift)

Englische Fassung:

STATUTORY DECLARATION

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

…… ……..
date (signature)

Acknowledgments

Ich möchte mich bei allen bedanken, die mich während meines Studiums unterstützt haben.

In erster Linie bedanke ich mich bei meinen Eltern Franz und Roswitha für ihre endlose Geduld und ihre Unterstützung.

Mein Bruder Michael war mit seinem Fleiß und seiner Zielstrebigkeit immer ein Vorbild.

Meiner Freundin Johanna danke ich besonders für ihren Zuspruch in schwierigen Zeiten.

Ich bedanke mich bei Christine für alles, was sie für mich getan hat.

Zu guter Letzt bedanke ich mich bei Professor Wolfgang Slany und dem gesamten Catrobat Team.

Abstract

Catrobat is a visual programming language intended to create and run programs by solely using mobile devices. Visual elements are arranged on-screen in order to form scripts. As scripts cannot communicate directly, a broadcast mechanism is used for communication between scripts. This broadcast mechanism is an important part of Catrobat and needs therefore to be properly specified. The concept of "Specification by Example", manifested in behavior-driven development, aims for creating a living, machine-executable documentation. This work introduces the concept of Specification by Example and presents its application to the broadcast mechanism of Catrobat. Initially, an overview of the visual programming language Catrobat is given. A text-based language is introduced for textually specifying Catrobat programs. Furthermore, the approach of behavior-driven development is introduced as manifestation of Specification by Example. Finally, the behavior-driven testing tool Cucumber is used to specify the broadcast mechanism of Catrobat by the means of examples.

Kurzfassung

Catrobat ist eine visuelle Programmiersprache, dafür ausgelegt, Programme, einzig durch Verwendung von mobilen Geräten, zu erstellen und auszuführen. Visuelle Elemente werden auf dem Bildschirm angeordnet, um Skripte zu bilden. Weil Skripte nicht direkt kommunizieren können, wird ein Broadcast-Mechanismus für die Kommunikation zwischen den Skripten verwendet. Dieser Broadcast-Mechanismus ist ein wichtiger Teil von Catrobat und muss daher korrekt spezifiziert werden. Das Konzept von „Spezifikation durch Beispiele“, das in der verhaltensgetriebenen Softwareentwicklung manifestiert ist, zielt darauf ab, eine lebendige, ausführbare Dokumentation zu erschaffen. Diese Arbeit stellt das Konzept von Spezifikation durch Beispiele vor, und präsentiert die Anwendung auf den Broadcast-Mechanismus von Catrobat. Zunächst wird ein Überblick über die visuelle Programmiersprache Catrobat gegeben. Eine textbasierte Sprache wird eingeführt, um Catrobat Programme textuell zu spezifizieren. Zudem wird der Ansatz der verhaltensgetriebenen Softwareentwicklung als Manifestation von Spezifikation durch Beispiele vorgestellt. Schließlich wird das verhaltensgetriebene Test-Werkzeug Cucumber verwendet, um den Broadcast-Mechanismus von Catrobat durch Beispiele zu spezifizieren.

Contents

List of Figures

List of Listings

1. Introduction

Delivering the right software product is of utmost importance in modern software development. Plenty of unsuccessful software projects have proven that there is still a demand for enhancement in the field of software development. Over the past few decades several agile methods have been evolved to support developers in their effort of delivering the software product the customer has asked for. Behavior-driven development is an attempt to bring together developers and customers, as well as all other stakeholders, to specify the requirements of the desired software product. The specifications are communicated in an ubiquitous language shared by all stakeholders. Supported by the usage of examples developers gain a deep understanding of the problem. Furthermore, examples written in an ubiquitous language act as living documentation. Behavior-driven testing can also support and complement an existing testing framework.

Catrobat is a visual programming language inspired by Scratch and intended to create and execute programs by solely using mobile devices like smartphones or tablet computers. Instead of writing statements like in a text-based programming language the programmer arranges visual elements to create a program. Special elements of the programming language provide the possibility of sending and receiving broadcast messages which allow communication between objects.

The following chapters provide an overview of the concept of behavior-driven development. Exemplary Catrobat programs are used to demonstrate the principle of executable specification. A text-based language syntax is introduced for specifying the exemplary Catrobat programs. Finally, the behavior-driven test framework Cucumber is used to specify the behavior of the broadcast mechanism of Catrobat.

2. Terminology

The terms defined in this chapter are used consistently throughout the remainder of this work.

Visual programming language

In a text-based programming language the programmer writes textual statements which, taken together, constitute the program. In a "visual programming language", however, the programmer, instead of writing textual statements, composes visual elements on the screen to create a program. Scratch, Snap!, and Catrobat are examples of visual programming languages.

Block, brick

The smallest elements of the aforementioned visual programming languages are called "blocks" or "bricks". Each of those elements has a special meaning and function. They are combined in certain ways to create a program.

Ubiquitous language

In a software project many people need to share their knowledge and communicate with each other. While domain experts communicate in a jargon of their field of expertise, developers usually understand and discuss the system in descriptive, functional terms, devoid of the meaning carried by the experts' language [13]. By agreeing upon a "ubiquitous language" all stakeholders of a project can communicate over a language understood by everyone involved.

Specification by example

A specification provides all relevant information to implement and use a system correctly. "Specification by example" is a set of process patterns

aiding in successfully creating and changing a software product [3]. The outcome is a machine-executable specification which represents a living documentation. Behavior-driven development is a manifestation of specification by example.

Cucumber

A famous testing framework for behavior-driven testing is "Cucumber". The test cases in Cucumber are called scenarios and are machine-executable. They serve as living documentation.

Story

"Stories" are representations of requirements. In behavior-driven development stories express some desired behavior. In Cucumber stories are called features. Features consist of one or more scenarios.

3. Visual Programming Languages for Children

In a visual programming language, the programmer arranges visual elements on the screen in order to form a program, instead of writing text-based statements. Catrobat is a visual programming language developed for mobile devices. Due to the dynamic nature of the language, which will be discussed further in section 3.6, children easily learn to program without having to worry about drawbacks like syntax errors or complicated workflows [31].

Inspired by Scratch, Catrobat also defines blocks which can be snapped together in order to create a program. Unlike Scratch, Catrobat programs can be created and executed by solely using mobile devices. A text-based Catrobat language is introduced in section 3.4, which is then utilized in chapter 5.

An interesting extension to Scratch is Snap! which allows to create user defined blocks. With user defined blocks, several programming language concepts, which are not realizable with Scratch, can be implemented.

This chapter also provides an introduction to statically and dynamically typed languages and highlights advantages as well as disadvantages in the usage of either. The last section of this chapter presents an analysis of whether Scratch, Snap!, and Catrobat could be defined as dynamic programming languages.

3.1. Scratch

Scratch is a visual programming environment created by the "Lifelong Kindergarten Group" at the "MIT Media Lab"[1]. Programming is done by snapping together blocks. Those blocks control 2-D graphical objects called "sprites" which are moving on a background called "stage" (see figure 3.2). Sprites encapsulate state by using variables and behavior via scripts [30].

There are four kinds of blocks in Scratch which are visually distinguishable by their color and their shape (see figure 3.1). "Command blocks" can be connected to form a sequence of commands. They correspond to statements in a text-based programming language. "Function blocks" are not joined in sequences like command blocks are. They act as functions and return values. A script is run upon the occurrence of a certain event defined by its "trigger block". "Control structure blocks" hold nested command sequences [23].

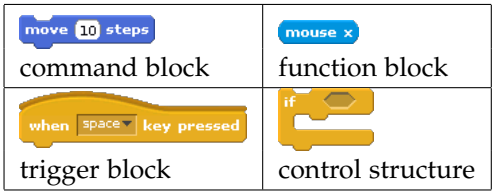


Figure 3.1: Scratch block types (adapted from Maloney et al. [23]).

The blocks can only be snapped together in certain ways according to their shapes. There are no syntax errors in Scratch as blocks fit together only in ways that make sense. Blocks which take parameters have parameter slots. The parameter slots are shaped according to the type of the parameter they require (see section 3.6 for an overview of data types in Scratch).

Command blocks, together with control structures, are attached to trigger blocks in order to form a script. The scripts describe the behavior of the sprite, and the state is encapsulated by variables. All scripts of all sprites run concurrently. The correct handling of thread switches, which may occur between any two instructions, is challenging in concurrent environments. The Scratch threading model, however, eliminates side effects from the interleaved execution of parallel running scripts by allowing thread switches only at the end of a loop or on a command that explicitly waits [23].

[1] `http://scratch.mit.edu` (visited on 2014-07-14)

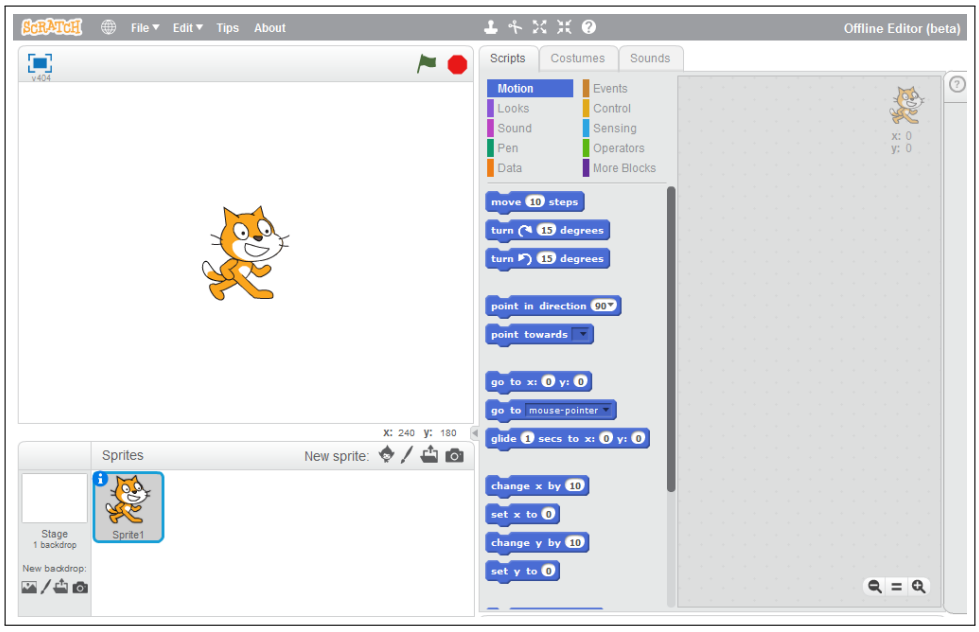


Figure 3.2: Scratch version 2.0 (offline editor).

Scripts of one sprite cannot call scripts from another sprite directly. Instead a broadcast mechanism is used. Any sprite can broadcast a message, which triggers all matching scripts containing the corresponding trigger block. Since all scripts run in parallel there is no obvious way to directly control threads. However, sometimes it is important to pause the execution of one script until another script has been executed. The `broadcast and wait` block provides this ability by pausing the execution of one script until all scripts with the corresponding trigger blocks have finished their execution.

3.2. Snap!

"Snap!"[2] is an extended reimplementation of Scratch. The former name of Snap! – BYOB (Build Your Own Blocks) – indicates that it allows to build

[2] `http://snap.berkeley.edu` (visited on 2014-07-14)

user defined blocks. Hence it is possible with Snap! to use programming language concepts by creating the appropriate blocks using the Block Editor. An example of an user defined block can be seen in figure 3.3. It specifies a block which draws a circle, when triggered.

Another concept, which Scratch is lacking, is recursion. In Snap! a new custom block can be dragged into its own definition forming a recursion.

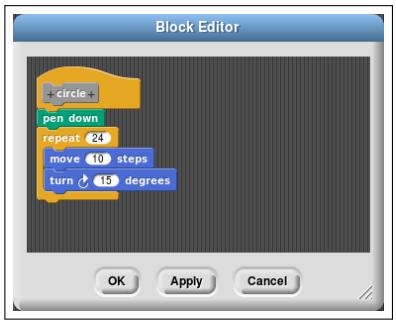


Figure 3.3: A user defined block.

In Scratch, only Boolean, numbers, and strings are first class types. A data type is first class, if data of that type can be the value of a variable, an input to a procedure, the value returned by a procedure, a member of a data aggregate, and anonymous (not named). In Snap! all data is first class [16]. This means, that in Snap! lists are also first class, which allows to have lists of lists.

Snap!, like Scratch, runs several scripts concurrently. It is usually not necessary to care about the threading system as thread switches happen at the end of loops and therefore cannot cause race conditions. In Snap!, however, it is possible to create a custom threading model by defining "thread" and "yield" blocks [16].

3.3. Catrobat

Catrobat is a visual programming language inspired by Scratch. The main difference between Scratch and Catrobat is, that Scratch requires a personal computer to run programs, whereas Catrobat programs are created and run by solely using smartphones or tablets. There is also a version in development which can be run in any HTML5-compatible browser, be it on a desktop computer, or a smartphone, or a tablet.

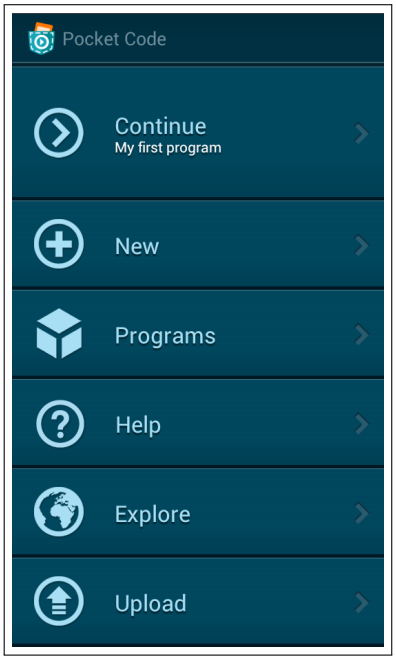


Figure 3.4: Pocket Code main menu.

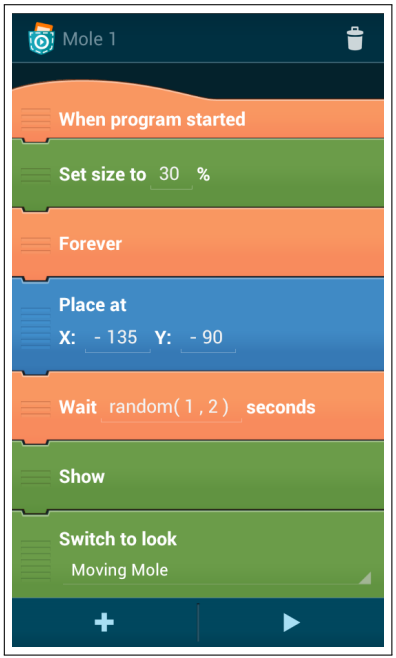


Figure 3.5: Catroid program.

The version of Catrobat which is developed for Android smartphones is named Catroid and is available on Google's Play Store as "Pocket Code"[3]. Figure 3.4 displays the main menu of Pocket Code.

Similar to Scratch, programs in Catroid are created by snapping together command blocks which are called "bricks" (see figure 3.5). The bricks are arranged in "scripts" which can run in parallel allowing concurrent execution. Just like in Scratch, Broadcast messages in Catrobat are used to ensure sequential execution of scripts [32].

[3] `https://pocketcode.org` (visited on 2014-07-14)

3.4. Text-based Catrobat Language Syntax

In this section the text-based Catrobat language syntax is analyzed. This language is mainly used to textually specify Catrobat programs which are used in Cucumber feature files (see section 5.3 for more information about Cucumber feature files in Catrobat). The language elements are organized into the categories "Control", "Motion", "Sound", "Looks", and "Variables".

- **Control**: Elements of this category are responsible for the program flow. By using bricks of this category broadcast messages can be sent and received, loops can be defined, and events can be caught.
- **Motion**: Elements of this category can modify the position and the orientation of an on-screen object.
- **Sound**: These elements play or stop a predefined sound file (sounds may also be recorded via the internal sound recorder) and alter the volume. Google's speak engine is used to read out some text.
- **Looks**: The visual appearance of objects can be defined with these elements. The size of an object can be set as well as transparency and brightness.
- **Variables**: Setting and changing the value of a variable is possible with elements of this category.

The language elements are referred to as bricks. Bricks are arranged in virtual containers called scripts. A brick is always part of a script and cannot exist stand-alone, except for the script bricks "`when program started`", "`when tapped`", and "`when I receive`", which can form an empty script by themselves. A Script starts with one of the bricks "`when program started`", "`when tapped`", or "`when I receive`", followed by any other brick, except for another script brick, or nothing. Two or more scripts may follow consecutively. The list of scripts may also be empty. The elements "`forever`" and "`repeat 10 times`" introduce loops and end with an "`end of loop`" brick. They contain zero or more other elements except for script bricks. They also may contain more "`forever`" and "`repeat 10 times`" bricks. The element "`if 1 is true then`" introduces a conditional path of execution. The alternative path is introduced by the element "`else`". The end of the conditional path is signaled by the element "`end if`". There are zero or more other

elements in between "`if 1 is true then`" and "`else`" as well as between "`else`" and "`end if`" except for script bricks. Those other elements may also be other "`if 1 is true then`", "`else`", and "`end if`" brick sequences.

The following list displays all brick elements of the Catrobat programming language (bricks which require a parameter are provided with arbitrary values). Notice, that the elements "`when program started`", "`when tapped`", and "`when I receive 'message 1'`" denote the beginning of scripts. The element "`end of loop`" cannot exist stand-alone, but is required by the elements "`forever`" and "`repeat 10 times`". Similar, the elements "`else`" and "`end if`" cannot exist stand-alone, but are required by the element "`if 1 is true then`".

- Control
  - `when program started`
  - `when tapped`
  - `when I receive 'message 1'`
  - `wait 1 second`
  - `broadcast 'message 1'`
  - `broadcast 'message 1' and wait`
  - `note 'add comment here...'`
  - `forever`
  - `repeat 10 times`
  - `end of loop`
  - `if 1 is true then`
  - `else`
  - `end if`
- Motion
  - `place at X: 100, Y: 200`
  - `set X to 100`
  - `set Y to 200`
  - `change X by 10`
  - `change Y by 10`
  - `move 10 steps`
  - `turn left 15 degrees`
  - `turn right 15 degrees`

  - `point in direction 90 degrees`
  - `point towards 'test object'`
  - `glide 1 second to X: 100, Y: 200`
- Sound
  - `start sound 'record'`
  - `stop all sounds`
  - `set volume to 60%`
  - `change volume by -10`
  - `speak 'Hello!'`
- Looks
  - `switch to look 'look 1'`
  - `next look`
  - `set size to 60%`
  - `change size by 10`
  - `hide`
  - `show`
  - `set transparency to 50%`
  - `change transparency by 25`
  - `set brightness to 50%`
  - `change brightness by 25`
  - `clear graphic effects`
- Variables
  - `set variable 'variable 1' to 0`
  - `change variable 'variable 1' by 0`

3.5. Static and Dynamic Programming Languages

Programming languages are often categorized into dynamically and statically typed languages. Statically typed languages define and enforce types at compile-time while dynamically typed languages check types at run-time. Bracha [8] states, that a dynamically typed languages is one that has no effect on the run-time semantics of the programming language, and does not mandate type annotations in the syntax. Hence, the main difference

between statically and dynamically typed languages is, *when* types are enforced [34]. Harper [15] depicts, that some candidate programs written in a statically typed language might be ill-typed and ruled out at compile time, whereas every program written in a dynamically typed language would be well-formed. An example for a dynamically typed languages is Lisp, which is also the earliest of its kind.

Some advantages of statically typed languages are described by Bracha [8]. He states, that types provide a form of machine-checkable documentation. Moreover, types provide a conceptual framework for the programmer, that is extremely useful for program design, maintenance, and understanding. Types also support early error detection. Another profit gained from languages with mandatory types might be a significant performance advantage due to optimization based on type information.

Tratt [34] states also some disadvantages of statically typed languages. He mentions, that type systems make languages excessively complex. There is a risk that type systems are overly restrictive or overly permissive. Type systems are a complex part of a programming language's specification. Errors in type systems might lead to impossible run-time behavior [9]. Static types make changing a system difficult, which may lead to premature ossification, making successive changes harder. Statically typed languages are mostly incapable of meaningful reflection, because compilers might discard information about a programs structure and its types in the process of optimization.

Dynamically typed languages, on the other hand, might be simpler to learn and to use because there are less corner cases to be aware of [34]. They often trade run-time efficiency for programmer productivity. Dynamically typed languages often provide high level features like build-in data types – lists, strings, or dictionaries, for example. Many dynamically typed languages also provide automatic memory management. The concept of garbage collection was first introduced by the Lisp programming language. Nowadays also many statically typed languages provide some kind of automatic memory management.

Some disadvantages might also be experienced in dynamically typed languages. Performance might be an issue, as the speed of execution is usually slower than with statically typed languages. However, highly optimized

libraries, dealing with performance critical work, are often provided with dynamically typed languages. The performance issue of dynamically typed languages might also be less problematic when programming can be done faster by programmers, which can focus on improving algorithms on a high level, instead of dealing with low-level coding [34].

Type systems are a form of documentation though, therefore the lack of explicit types might be a disadvantage in the expressiveness of dynamically typed languages. However, most expected types may be recognized informally [34].

3.6. Scratch, Snap!, and Catrobat as Dynamic Programming Languages

Scratch

Scratch has three first class data types, which are Boolean, number, and string. These data types can be used in expressions, stored in variables, or returned by functions (see section 3.2 for a definition of first class data types). However, Scratch variables can hold values of any data type, which avoids the requirement of explicitly specifying the type. The data type of a variable is converted between numbers and strings depending on context [23]. Additionally to variables, Scratch also defines lists. Lists also can contain the data types Boolean, number, and string. It is also possible to define lists of lists.

According to the definition of dynamic programming languages in section 3.5, Scratch can be seen as dynamic programming languages. The data types of variables are not required to be explicitly defined. Scratch even converts between data types at run-time, if possible. However, some kind of type checking is done at programming-time, concerning the data types of parameters. Blocks of functions, which take parameters, have parameter slots, which are shaped accordingly to the data type they require (see section 3.1). Boolean functions are the most strict, accepting only Boolean function blocks, whereas number and string parameter slots are less strict, accepting

a function block of any type, converting the parameter to the target type, if necessary [23].

Snap!

Similar to Scratch, Snap! would also be classified as dynamic programming language. Data types are not enforced and if possible are converted to something useful at run-time. Parameter slots are shaped according to the type they require. Hence it is not possible to provide a wrong type since the language simply would not allow it. When defining a new block using the Block Editor, the shape of the parameter slot can be defined according to the data type which is required.

Catrobat

In Catroid, the implementation of the Catrobat language for Android devices, a special module, the Formula Editor (see figure 3.6), is used to input values into the parameter slots. The Formula Editor displays formulas textually and resembles a pocket calculator [17]. It provides only valid characters, functions, variables, object values, and sensor values. It is not required to explicitly specify a type, although the type of a formula is always numeric. Before saving the formula for a certain parameter, the expression is checked for syntactical correctness. When syntactically correct, the value of the formula can be calculated. Values of variables in Catroid are also set via the Formula Editor and do not require a type to be explicitly specified.

The bricks `NoteBrick` and `SpeakBrick` provide a free form text input field. The text is internally stored as string data type, however it is not required to explicitly declare the input as string. The `NoteBrick` stores a string for informative purpose, as its name implies. The `SpeakBrick` also takes the input as a string which may of course also contain numeral characters.

Some bricks store internally a list of strings. These bricks, like the `Broadcast` brick for example, take a string as input and append it to the internal list. Again, it is not required to care about the data type as it is defined by the language.

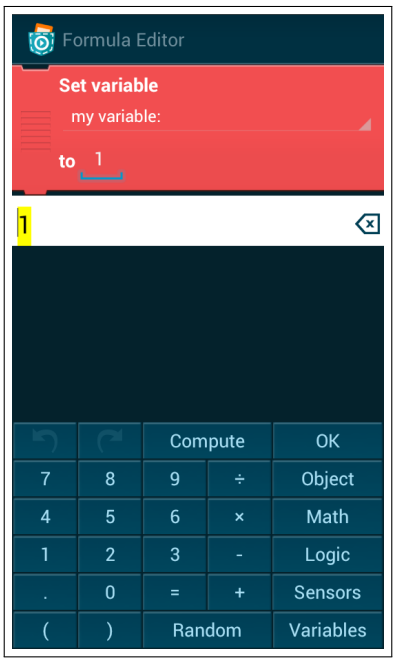


Figure 3.6: Catroid Formula Editor.

4. Behavior-Driven Development

Testing is an important part of software development. Good quality of a software product can only be assured if testing accompanies the entire development process. By involving every stakeholder of a software product chances are good that the right product will be created. Behavior-Driven Development is an approach to bring together developers, testers, and customers to create the specification of the product together. This can be achieved by communicating in a ubiquitous language which is understood by every stakeholder.

4.1. Test-Driven Development

Testing is vital important for successfully delivering software products. Beck [5] goes even a step further and states, that a software feature without an automated test simply does not exist. In an ideal agile development environment, tests would be written before even writing any code. Hence, the desired behavior of a feature has to be well-defined beforehand.

Koskela [20] states, that test-driven development (TDD) turns around the traditional design–code–test sequence:

$$Design \longrightarrow Code \longrightarrow Test$$

$$Test \longrightarrow Code \longrightarrow Design$$

The intent of putting design after code might not be obvious at first glance, but refactoring might be considered a powerful design technique [6]. Thus the TDD development sequence becomes test–code–refactor. This sequence

is also often referenced to as red–green–refactor. As developers are encouraged to write tests before writing code, new added tests will fail. Failing tests are often displayed as red bars. By adding code and making the tests pass the bar will turn green. Once the bar turns green it is safe to refactor.

Astels [4] describes TDD as a style of development where:

- an exhaustive suite of Programmer Tests is maintained,
- no code goes into production unless it has associated tests,
- the tests are written first,
- the tests determine what code is needed to be written.

4.2. Acceptance Test-Driven Development

While TDD is a developer technique, acceptance test-driven development (ATDD) is a whole-team technique, hence they must not be confused. Both write tests first, but their goals are unalike [21]. Similar, Koskela [20] elaborates, that TDD is a technique for improving the software's internal quality, whereas ATDD keeps a product's external quality upright.

McConnell [24] provides a definition for internal and external software quality:

Internal characteristics of software quality:

- Maintainability
- Flexibility
- Portability
- Reusability
- Readability
- Testability
- Understandability

External characteristics of software quality:

- Correctness
- Usability
- Efficiency

- Reliability
- Integrity
- Adaptability
- Accuracy
- Robustness

Some of the characteristics of internal and external software quality may influence each other. A high internal quality makes software easy to understand and therefore easy to extend and to test. It states how well the needs of the developers and administrators are met [14]. TDD ensures that the internal quality stays upright. External characteristics of quality, however, measures how well the requirements of the stakeholders are held. ATDD ensures just that the external characteristics of quality are held upright. An external user of the system does not care about how well the source code or the tests can be read or how reusable some modules are. From an external point of view, a system needs to be correct and easy to use, as well as reliable.

Adzic [1] describes the process of acceptance testing as a pattern where examples become acceptance tests. First, real-world examples are used to build a shared understanding of the domain. A set of these examples is then selected to be a specification and acceptance tests. The verification of the acceptance tests has to be automated. Software development can then focus on the acceptance tests. Finally, the set of acceptance tests might be used as base for discussion about future change requests, and a new cycle of development can start.

Pugh [29] emphasizes the importance of having testable requirements. Acceptance tests are a communication medium between several roles in a software development team. A test that passes is a specification of how the system works.

ATDD is also called behavior-driven development (BDD) or specification by example (SbE) [3]. The goal of ATDD is to specify executable requirements. Acceptance tests written as Cucumber features serve as executable specifications [18]. Cucumber features will be described in section 4.5.1.

While unit tests are aimed at developers, acceptance tests are aimed at the whole team including developers, testers, or business stakeholders. Instead

of having business stakeholders passing requirements to the developers without opportunity for feedback, business stakeholders and developers collaborate to write automated tests [18].

Acceptance tests act as living documentation. They can be read and written by all stakeholders of a team, but they also can be automatically verified by a computer at any time. Instead of being written once and going out of date, they become a living thing and reflect the state of the project [18].

4.3. Introduction to Behavior-Driven Development

The idea of BDD was originally introduced by North [26] as an evolution of TDD. As practices of TDD left many questions unanswered, he came up with several techniques and practices which provided answers to those questions and resulted in software that had a value for a stakeholder. With the concept of business value in mind it is always clear what the next most important thing would be, a system does not do yet. So a developer would always know what to implement next. A further concept, which was introduced, was the naming scheme of test methods. Test method names should be sentences so that it is absolutely clear, what behavior a test method ensures. Furthermore, the method name should be in the language of the business domain to assure that every person involved can communicate over it. Evans [13] uses the term "ubiquitous language" to describe this shared vocabulary.

BDD provides answers to questions like "where to start", "what to test", "what not to test", "how much to test", "what to call the tests", and "how to understand why a test fails". If a test fails, either a bug was introduced, or the specified behavior is no longer valid, or the behavior is still valid but had moved somewhere else.

It was pointed out by Keogh [19], that BDD is strongly aligned with Lean principles. BDD focuses on learning because it encourages questions, conversations, creative exploration, and feedback. A common (ubiquitous)

language empowers the team by creating a shared understanding of a domain.

The importance of interaction between business and software design was also emphasized by Lazăr et al. [22]. They combine a model-driven development context with the principles of BDD and present a domain model of the main BDD concepts.

Solís and Wang [33] investigated related work on BDD and extracted the main characteristics of BDD. They affirm the importance of a ubiquitous language to be used to describe the behavior of a system. Through an iterative decomposition process several user stories can be derived from a defined business value. User stories and scenarios are described in plain text with the use of predefined templates (see also section 4.3.2). An acceptance test in BDD validates the behavior of a system via executable specification. The code is part of the system's documentation. Finally, BDD happens at different phases of software development, from the initial planing phase to analysis phase to implementation phase.

4.3.1. JBehave and Executable Specification

North [26] created a tool called JBehave. Derived from JUnit it emphasizes the principles of BDD. The specification is in a sense executable, as the object model allows a direct mapping of the scenario fragments to Java classes.

4.3.2. User Stories in Behavior-Driven Development

A story is a description of a requirement and its business benefit. Moreover it represents a set of criteria by which all involved project members agree that a task is done. North [27] suggests that a story should contain at least a title, a narrative, and some acceptance criteria. The title would be one line describing the story. The narrative describes a role, a feature, and a benefit gained from this feature. North suggests to use the "Connextra" format (after the company where it was used first): "As a [role] I want [feature] so

that [benefit]". Thus, it is always clear that a requested feature is in some way beneficial for a person in a certain role. Finally, the acceptance criteria is presented as one or more scenarios. A scenario starts with a title describing the desired behavior. The scenario then is described in terms of Givens, Events, and Outcomes. A story template containing the minimal suggested content can be seen in listing 4.1.

```
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 4.1: What's in a Story? (adapted from North [27])

4.3.3. Specification by Example

The concept of specifying software by the means of examples was introduced by Parnas [28], who proposes, that a specification must provide all information necessary to implement and use a system correctly, and nothing more. Moreover, it must be sufficient formal that it can be machine tested. It also should make use of terms normally used by user and implementer alike.

Adzic [3] states that all stakeholders and delivery team members must understand what needs to be delivered in order to deliver the right software

product. Examples act as a living documentation, which facilitates changes in the software as well as in the team.

By communicating via a ubiquitous language, customers are able to converse in their own domain language [29]. Examples are also important as a communication instrument between developers among each other. Practice has shown that SbE can help to straighten out inconsistencies in specifications and successfully support communication between all involved project members [2]. Examples might be also be helpful to illustrate corner cases.

Adzic [3] derives a set of process patterns that go along with SbE. He locates the initial starting point of a software project within the business goals of a customer. The scope of the project is then derived from these business goals. Key examples help defining the scope and can be easily obtained by specifying collaboratively. Once the key examples are agreed upon the specification may be refined. In order to make the specification executable the examples have to be automated without changing the specification. To have an executable specification of a software system means to be able to validate it frequently. Another benefit of an executable specification is, that it further acts as a living documentation system.

4.4. The Role of Documentation in Behavior-Driven Development

The function and design of a system will always be changing. Hence, a written documentation will always be out of date and not in sync with the code. At the same time, designers need to remember and express their design ideas [11]. Thus, there might be need for documentation but following the principles of agile software development [7] working software is more important.

The outcome of SbE is machine-executable documentation. While having working software guaranteed by the test first principle, the machine-executable examples also serve as a documentation, which is easy to understand and consistent. This form of documentation always reflects the actual state of the project and can be automatically validated at any time.

4.5. Cucumber

RSpec was created as a successor of JBehave. Hellesøy rewrote the "Story Runner", a component of RSpec, as Cucumber [18]. Originally written in the Ruby programming language Cucumber nowadays supports many programming languages [12].

4.5.1. Cucumber and Gherkin

Like mentioned in section 4.3.2, stories usually have a title, a narrative, and a number of scenarios. The stories in Cucumber are called features. Cucumber features are written in the Gherkin language. The Gherkin grammar defines only a few keywords which are mandatory. The rest of the feature is free-form text. The summarized Gherkin keywords are:

- Feature
- Background
- Scenario
- Scenario outline
- Scenarios (or Examples)
- Given
- When
- Then
- And (or But)
- *

Reserved characters include "|", which is used for defining tables, """", used for defining strings spanning multiple lines , and "#", marking comment lines [10]. The keywords "Given", "When", "Then", "And", "But", and "*" mark the beginning of a step. The Gherkin keywords taken by themselves have no special meaning to Cucumber and are freely interchangeable. Hence it does not matter if several "Given" steps each start with the keyword "Given" or the keyword "And" – or simply any other of the keywords.

Upon execution, Cucumber parses the steps in a given feature file and tries to match them to a step definition. The step definitions can be written in any

programming language supported by Cucumber. When using Java[1], the step definitions start with `@Given()`, `@When()`, and `@Then()` as Java annotations. The annotations take a regular expression string, which is used to match the steps of the feature to the corresponding step definition.

A sample feature which can be parsed by Cucumber is shown in listing 4.2. The feature starts with the keyword "Feature" followed by a short description. The keyword "Background" tells Cucumber to execute the following steps before every scenario. The only scenario in this example contains two steps. The second step uses a data table which can combine data from several steps into one table. Instead of writing "`Then I should see the 'Continue' button`", "`And I should see the 'New' button`", et cetera, all the steps can be expressed compact in one step.

```
Feature: Main menu

  In order to give the user a starting point
  The main menu offers a number of distinctive options

  Background:
    Given I have a Program

  Scenario: The main menu has a list of labeled buttons
    Given I am in the main menu
    Then I should see the following buttons
      | Continue |
      | New      |
      | Programs |
      | Help     |
      | Explore  |
      | Upload   |
```

Listing 4.2: A sample Cucumber feature describing some UI components of the main menu of the Catroid application.

The feature describes some UI components of the main menu of the Catroid application (compare figure 3.4 in section 3.3).

[1]Java is used in this examples because Catroid is an Android application. Android applications are mainly written in Java. See section 3.3 for more information about Catroid.

When the feature gets parsed by Cucumber, the text after the Gherkin keywords is extracted and matched to the regular expression strings of the step definitions. If Cucumber finds no suitable step definition, it then suggests a regular expression and an empty method body which can be used as a starting point for implementing this step (see listing 4.3). The step is therefore marked as "pending". This is indicated in Java by throwing a `PendingException`.

```
@Given("^I have a Program$")
public void i_have_a_Program() throws Throwable {
  // Write code here that turns the phrase above into concrete actions
  throw new PendingException();
}

@Given("^I am in the main menu$")
public void i_am_in_the_main_menu() throws Throwable {
  // Write code here that turns the phrase above into concrete actions
  throw new PendingException();
}

@Then("^I should see the following buttons$")
public void i_should_see_the_following_buttons(DataTable arg1)
    throws Throwable {
  // Write code here that turns the phrase above into concrete actions
  // For automatic conversion, change DataTable to List<YourType>
  throw new PendingException();
}
```

Listing 4.3: Step definitions matching the steps in listing 4.2.

Listing 4.4 shows an exemplary step definition which implements the "`I am in the main menu`" step. As can be seen in the example listing, Cucumber can work together with other testing libraries like Robotium[2], a test automation framework for the Android platform. In the step definitions the Robotium class "`Solo`" is called. The Activity returned by the call to `Solo` is then assured to be of the desired type. If the comparison fails, an exception is thrown notifying Cucumber to mark the step as failed. If the comparison

[2] `https://code.google.com/p/robotium` (visited on 2014-07-14)

holds true, the method completes without exception signaling that the step succeeded.

```
@Given("^I am in the main menu$")
public void I_am_in_the_main_menu() {
  Solo solo = (Solo) Cucumber.get(Cucumber.KEY_SOLO);
  assertEquals("I am not in the main menu.", MainMenuActivity.class,
    solo.getCurrentActivity().getClass());
}
```

Listing 4.4: Implementation of a step definition introduced in listing 4.3.

4.5.2. Scenario Outline

Repetition of steps which follow the same pattern and only differ in input values, makes scenarios hard to read and to maintain. When specifying several scenarios following the same scheme, Cucumber provides a mechanism for compacting these scenarios into one scenario by using a scenario outline. Listing 4.5 shows a Cucumber feature with two similar scenarios. They only differ in the value of the coordinate.

The same feature can be rewritten by using a scenario outline. Listing 4.6 illustrates the usage of this property. The terms in angle brackets indicate the placeholder and the data table labeled “Examples” provides the exemplary values for the scenario. Internally, each row of the “Examples” table gets converted into a Cucumber scenario, but the readability is greatly increased by using a scenario outline. An important advantage of using a scenario outline is, that gaps in examples can easily be spotted [18]. By having the examples in a table it can be clearly determined, for example, if there are enough corner cases present.

```
Feature: A brick setting the x coordinate of an Object

  Background:
    Given I have a Program
    And this program has an Object 'test object'

  Scenario: Set x coordinate of an Object
    Given 'test object' has a Start script
    And this script has a Set x to 100 brick
    When I start the program
    And I wait until the program has stopped
    Then 'test object' should be at x position 100

  Scenario: Set x coordinate of an Object
    Given 'test object' has a Start script
    And this script has a Set x to 500 brick
    When I start the program
    And I wait until the program has stopped
    Then 'test object' should be at x position 500
```

Listing 4.5: A Cucumber feature with two similar scenarios.

```
Feature: A brick setting the x coordinate of an Object

  Background:
    Given I have a Program
    And this program has an Object 'test object'

  Scenario Outline: Set x coordinate of an Object
    Given 'test object' has a Start script
    And this script has a Set x to <xPosition> brick
    When I start the program
    And I wait until the program has stopped
    Then 'test object' should be at x position <xPosition>

    Examples:
      | xPosition |
      | 100       |
      | 500       |
```

Listing 4.6: A Cucumber feature with a scenario outline.

5. Catroid and Behavior-Driven Testing

Catroid is an Android application for creating and running programs in the Catrobat programming language. Section 3.3 provides an introduction to Catrobat. The concept of BDD is introduced in chapter 4. This chapter is about introducing the concept of BDD into Catroid. The behavior-driven testing framework Cucumber is used to create a test suite which specifies the broadcast mechanism of Catrobat.

5.1. Cucumber Android

As mentioned in section 4.5.1, Android applications are mainly written in Java. However, Android does not directly use a Java Virtual Machine but a derivation named "Dalvik Virtual Machine", optimized for running on mobile devices.

The module "Cucumber-JVM"[1] of the Cucumber framework contains a sub-module "Cucumber-Android" allowing the execution of step definitions directly on the Dalvik Virtual Machine.

5.2. Test-driven Development in Catroid

Following the agile principles of Extreme Programming (XP) concerning testing and continuous integration (CI), the Catroid tests are run at least

[1] `https://github.com/cucumber/cucumber-jvm` (visited on 2014-07-14)

once a day on a CI server. The tests cover functionality as well as UI design. Every Catroid developer is encouraged to follow the TDD principles defined in section 4.1 and write tests before writing or changing any code.

What is currently missing in the Catroid testing framework is a possibility to easily test behavior. Although it is possible to test behavior with unit tests, they are often hard to maintain and hard to understand for people lacking technical knowledge. For this reason Cucumber is established in Catroid as a behavior testing framework.

5.3. Catroid and Cucumber Feature Files

Cucumber feature files are introduced in section 4.5.1. Listing 5.1 displays another Cucumber feature file. It specifies a scenario, which assures that a `BroadcastAndWait` brick is unblocked when the broadcast message is sent again (see section 5.4.1 for an overview of the broadcast messaging system in Catroid and a description of the `BroadcastAndWait` brick). Note, however, that the `Print` brick is not part of the Catrobat language, but was introduced for debugging purpose. The `Print` brick is placed at key positions after other bricks. Once the message of a particular `Print` brick gets printed it is certain, that the preceding brick was finally handled. This is especially useful to visually express the termination of a `BroadcastAndWait` brick's waiting phase.

The scenario documents how the system should behave for this particular event. It is specified in terms of bricks which are added to scripts and scripts which are added to a Catroid program. However, the Gherkin keywords together with the repetition of text like "`this script has a`" make the scenario quite noisy. It may take the reader a considerable amount of time to understand the meaning of the scenario in question. Replacing the Gherkin keywords with "`*`", as shown in listing 5.2, removes some of the clutter, however, some of the meaning is lost too. It is not instantly clear, which steps specify the presumed context, the events, or the expected outcome. The repetition of phrases like "`this script has a`" is also still present.

```
Feature: BroadcastAndWait blocking behavior (like in Scratch)

  Background:
    Given I have a Program
    And this program has an Object 'test object'

  Scenario: A waiting BroadcastAndWait brick is unblocked when the
            broadcast message is sent again.

    Given 'test object' has a Start script
    And this script has a BroadcastAndWait 'Print a after 0.1 seconds,
          and then b after another 0.3 seconds' brick
    And this script has a Print brick with 'c'
    Given 'test object' has a Start script
    And this script has a Wait 200 milliseconds brick
    And this script has a Broadcast 'Print a after 0.1 seconds, and
          then b after another 0.3 seconds' brick
    Given 'test object' has a WhenBroadcastReceived 'Print a after 0.1
          seconds, and then b after another 0.3 seconds' script
    And this script has a Wait 100 milliseconds brick
    And this script has a Print brick with 'a'
    And this script has a Wait 300 milliseconds brick
    And this script has a Print brick with 'b'
    When I start the program
    And I wait until the program has stopped
    Then I should see the printed output 'acab'
```

Listing 5.1: A Cucumber feature specifying the behavior of a `BroadcastAndWait` brick when the broadcast message is sent again.

The approach taken for integrating Cucumber features with Catroid is to omit the Gherkin keywords within a scenario except for "When" and "Then". The remaining part of the steps is rephrased in order to make them more expressive (see listing 5.3).

By omitting the Gherkin keywords and the repetitive part of the steps the scenario now rather can be read as a Catroid program (compare figure 5.1 for the corresponding Catroid program). Thus it takes the reader less time to fully understand the meaning of the scenario.

However, the feature as it is now is not able to be run by Cucumber di-

```
Feature: BroadcastAndWait blocking behavior (like in Scratch)

  Background:
    * I have a Program
    * this program has an Object 'test object'

  Scenario: A waiting BroadcastAndWait brick is unblocked when the
            broadcast message is sent again.

    * 'test object' has a Start script
    * this script has a BroadcastAndWait 'Print a after 0.1 seconds,
        and then b after another 0.3 seconds' brick
    * this script has a Print brick with 'c'
    * 'test object' has a Start script
    * this script has a Wait 200 milliseconds brick
    * this script has a Broadcast 'Print a after 0.1 seconds, and
        then b after another 0.3 seconds' brick
    * 'test object' has a WhenBroadcastReceived 'Print a after 0.1
        seconds, and then b after another 0.3 seconds' script
    * this script has a Wait 100 milliseconds brick
    * this script has a Print brick with 'a'
    * this script has a Wait 300 milliseconds brick
    * this script has a Print brick with 'b'
    * I start the program
    * I wait until the program has stopped
    * I should see the printed output 'acab'
```

Listing 5.2: Replace Gherkin keywords with "*".

rectly because Cucumber would not properly recognize the steps when the Gherkin keywords are missing. Hence, a script is used, which adds Gherkin keywords and is run before Cucumber parses the feature file.

The script in question (see listing 5.4) is just a simple call to the Unix-Tool Stream Editor (sed) scanning for line beginnings followed by certain trigger words and adding the Gherkin keyword "And" if needed (like already mentioned in section 4.5.1, Gherkin keywords are interchangeable and it makes no difference to Cucumber which keywords are in use). A full description of the syntax of this new Catrobat Language can be found in section 3.4. The complete feature specifying the behavior of the "`BroadcastAndWait`"

brick is discussed in section 5.5.

```
Feature: BroadcastAndWait blocking behavior (like in Scratch)

  Background:
    Given I have a Program
    And this program has an Object 'test object'

  Scenario: A waiting BroadcastAndWait brick is unblocked when the
            broadcast message is sent again.

    Given Object 'test object' has the following scripts:
      when program started
        broadcast 'Print a after 0.1 seconds, and then b after another
          0.3 seconds' and wait
        print 'c'
      when program started
        wait 0.2 seconds
        broadcast 'Print a after 0.1 seconds, and then b after another
          0.3 seconds'
      when I receive 'Print a after 0.1 seconds, and then b after another
          0.3 seconds'
        wait 0.1 seconds
        print 'a'
        wait 0.3 seconds
        print 'b'

    When I start the program
    And I wait until the program has stopped
    Then I should see the printed output 'acab'
```

Listing 5.3: Shorter and more expressive steps.

By taking this approach and keeping step lines as short as possible the readability of the features is increased. Omitting Gherkin keywords when writing scenarios may increase the readability further because scenarios describe Catroid programs and can thus be read as such. As Cucumber cannot parse steps without Gherkin keywords, they are added by a script which can run automatically before Cucumber starts parsing the file. Certainly, the regular expressions of the step definitions have to be adapted to match the new steps.

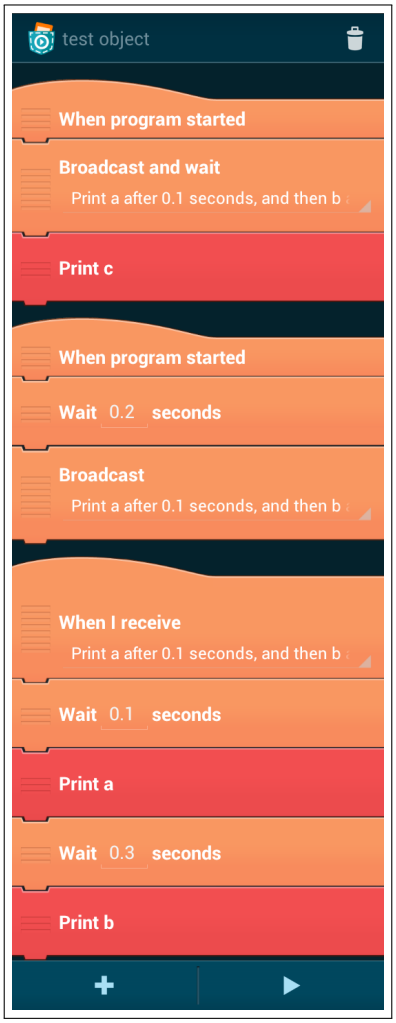


Figure 5.1: Catroid program which is created by the Cucumber scenario in listing 5.3.

```
#!/bin/sh
for f in $(find assets -name '*.source'); do
  sed -r 's/^([[:space:]]*)(when |broadcast |print |wait
    |repeat |end of loop)/\1And \2/g' $f > $f.feature;
done

@echo off
for /R "assets" %%f in (*.source) do (
  sed.exe -r "s/^([[:space:]]*)(when |broadcast |print |wait
    |repeat |end of loop)/\1And \2/g" %%f > %%f.feature
)
```

Listing 5.4: Script adding Gherkin keywords to step definitions for all files ending in `.source` in the directory `assets` as Unix shell script (top) and Windows batch script (bottom).

5.3.1. Waiting Periods in Cucumber Features

Fast feedback is especially important for testing in an agile environment. Developers are encouraged to run tests often, ideally before and after every change in code. When tests tend to take too long to run, developers would merely skip them altogether.

Listing 5.3 in section 5.3 displays a Cucumber feature specifying the behavior of a `BroadcastAndWait` brick. Notice, that the values for the waiting periods are chosen to be as small as possible yet still assuring the desired order of execution. Within one and the same scenario different values for different waiting periods are taken. Those are 0.1 seconds apart. It is thereby obvious that the waiting periods have nothing in common and serve different purposes.

Choosing the waiting periods as small as possible but adding small amounts of time to make them unique is a trade-off between a short overall testing time and an unmistakable meaning.

5.3.2. Using Descriptive Strings as Broadcast Messages

Like mentioned in section 3.3 Catrobat makes use of a broadcast mechanism to provide means of communication between scripts. Arbitrary strings are used as broadcast messages, distinctively identifying broadcasts. Broadcast messages in the Cucumber features are chosen to be descriptive because the features also serve as documentation. An example of a broadcast message, taken from listing 5.3, would be:

```
'Print a after 0.1 seconds, and then b after another 0.3 seconds'
```

The message describes that "a" should be printed after 0.1 seconds, and then "b" after another 0.3 seconds. Thus it is absolutely clear what scripts will be triggered and which actions will be performed after a particular broadcast message has been sent.

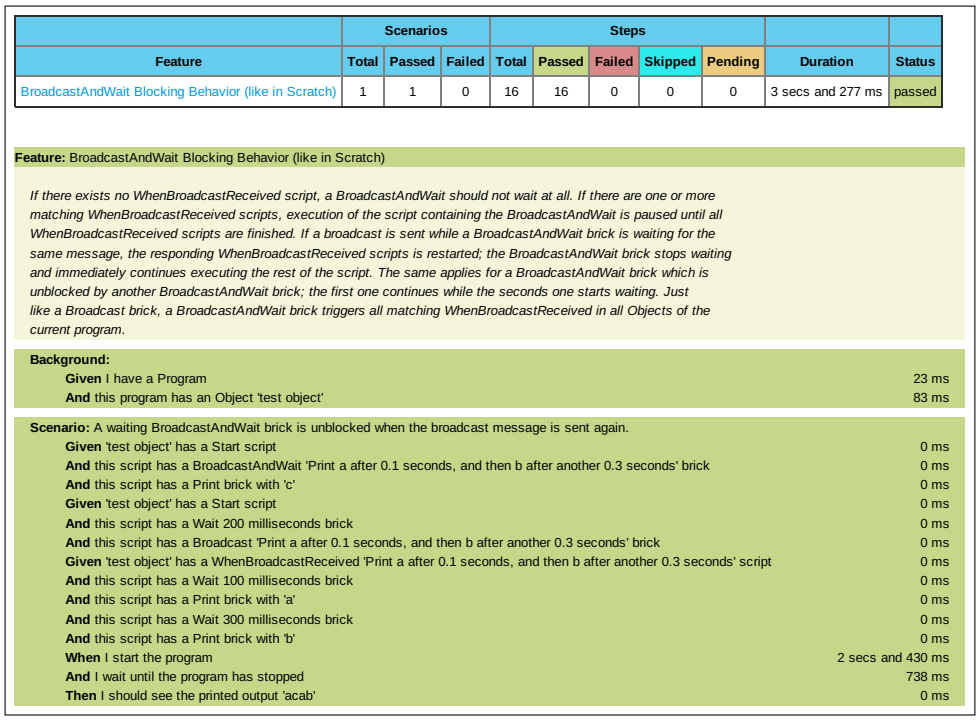
5.3.3. Cucumber and Continuous Integration

Jenkins[2] is a tool, providing CI for software development. Like already mentioned in section 5.2 the whole Catroid test suite runs at least once a day on a dedicated CI server using Jenkins. Every developer is encouraged to follow the test-first principle and to write tests before every code change. Upon requesting a merge of a code change into the code base, the developer runs all tests on the CI server.

After enabling the Cucumber plugin, the CI server is also able to process Cucumber feature files. See figure 5.2 for a successful test run of a Cucumber feature on the Jenkins CI server. The green background of the steps indicates that the steps passed.

The Cucumber plugin also reports the time taken by each step. Therefore it would be easy to spot bottlenecks taking a significant amount of time to execute. The most steps in this example take zero milliseconds to execute because they only set up the Catroid program which is then run via step "`When I start the program`".

[2] `http://jenkins-ci.org` (visited on 2014-07-14)



	Scenarios			Steps						
Feature	Total	Passed	Failed	Total	Passed	Failed	Skipped	Pending	Duration	Status
BroadcastAndWait Blocking Behavior (like in Scratch)	1	1	0	16	16	0	0	0	3 secs and 277 ms	passed

Feature: BroadcastAndWait Blocking Behavior (like in Scratch)

If there exists no WhenBroadcastReceived script, a BroadcastAndWait should not wait at all. If there are one or more matching WhenBroadcastReceived scripts, execution of the script containing the BroadcastAndWait is paused until all WhenBroadcastReceived scripts are finished. If a broadcast is sent while a BroadcastAndWait brick is waiting for the same message, the responding WhenBroadcastReceived scripts is restarted; the BroadcastAndWait brick stops waiting and immediately continues executing the rest of the script. The same applies for a BroadcastAndWait brick which is unblocked by another BroadcastAndWait brick; the first one continues while the seconds one starts waiting. Just like a Broadcast brick, a BroadcastAndWait brick triggers all matching WhenBroadcastReceived in all Objects of the current program.

Background:

Given I have a Program	23 ms
And this program has an Object 'test object'	83 ms

Scenario: A waiting BroadcastAndWait brick is unblocked when the broadcast message is sent again.

Given 'test object' has a Start script	0 ms
And this script has a BroadcastAndWait 'Print a after 0.1 seconds, and then b after another 0.3 seconds' brick	0 ms
And this script has a Print brick with 'c'	0 ms
Given 'test object' has a Start script	0 ms
And this script has a Wait 200 milliseconds brick	0 ms
And this script has a Broadcast 'Print a after 0.1 seconds, and then b after another 0.3 seconds' brick	0 ms
Given 'test object' has a WhenBroadcastReceived 'Print a after 0.1 seconds, and then b after another 0.3 seconds' script	0 ms
And this script has a Wait 100 milliseconds brick	0 ms
And this script has a Print brick with 'a'	0 ms
And this script has a Wait 300 milliseconds brick	0 ms
And this script has a Print brick with 'b'	0 ms
When I start the program	2 secs and 430 ms
And I wait until the program has stopped	738 ms
Then I should see the printed output 'acab'	0 ms

Figure 5.2: Test report from the Jenkins Cucumber plugin.

What also can be seen in figure 5.2 is the description of the feature following the "Feature" keyword. Notice that this description not only refers to the single scenario visible in this figure, but also to the whole feature specifying the behavior of the `BroadcastAndWait` brick. The whole feature is discussed in section 5.5.

5.3.4. Regression Testing with Cucumber Features

Regression testing is performed after making a functional improvement or repair to the program [25]. Erroneous behavior of Catrobat can be easily reproduced by using the text-based language defined in section 3.4. When a bug is discovered the workflow from section 6.1 may be applied and a text-based Catrobat program simulating the erroneous behavior may be

created in a Cucumber feature. This would also serve as regression test to assure that no other behavior is influenced by the correction of this error.

5.3.5. Cucumber and Concurrency

Like mentioned in section 3.3 scripts in Catrobat run in parallel. When two scripts run concurrently it is not determinable which script finishes first. Listing 5.5 shows a Cucumber feature specifying two `WhenIReceive` scripts. The scripts are supposed to run concurrently and it is not determinable if the `Print` brick with "a" or the `Print` brick with "b" gets executed first. This is expressed by the expected outcome: "`Then I should see the printed output 'ab' or 'ba'`".

```
Feature: Concurrency in Catroid

  Scenario: A Broadcast brick sends a message in a program with two
            WhenBroadcastReceived scripts. The order of the execution
            of the WhenBroadcastReceived scripts is arbitrary.

    Given I have a Program
    And this program has an Object 'test object'
    Given 'test object' has the following scripts:
      when program started
        broadcast 'message'
      when I receive 'message'
        print 'a'
      when I receive 'message'
        print 'b'

    When I start the program
    And I wait until the program has stopped
    Then I should see the printed output 'ab' or 'ba'
```

Listing 5.5: Cucumber scenario describing the ambiguous outcome of two scrips running concurrently.

5.4. Correcting Undesired Behavior in the Catroid Broadcast Messaging System

Like mentioned in section 3.3 scripts cannot call other scripts directly. Instead a broadcast mechanism is used to intercommunicate with other scripts. The broadcast messages can also be used to ensure sequential execution of scripts. Listing 5.6 shows an example of how the sequential execution of three scripts can be ensured by the usage of broadcast messages.

```
Feature: Scripts communicating via broadcast messages

  Scenario: Sequential execution of three scripts

    Given I have a Program
    And this program has an Object 'test object'
    Given 'test object' has the following scripts:
      when program started
        print 'a'
        broadcast 'message one'
      when I receive 'message one'
        print 'b'
        broadcast 'message two'
      when I receive message two'
        print 'c'

    When I start the program
    And I wait until the program has stopped
    Then I should see the printed output 'abc'
```

Listing 5.6: Sequential execution of three scripts. The broadcast messages ensure the correct order of execution.

5.4.1. Overview of the Broadcast Messaging System in Catroid

The bricks `Broadcast`, `BroadcastAndWait`, and `WhenIReceive` are responsible for handling broadcasts in Catroid. While the bricks `Broadcast` and

`BroadcastAndWait` are sending broadcast messages, `WhenIReceive` denotes a script responding to the broadcast messages. The actions appended to this script are triggered when the broadcast message is received.

The difference between `Broadcast` and `BroadcastAndWait` bricks is that the former is non-blocking which means that the broadcast message gets sent and the script containing the `Broadcast` brick directly continues with its execution. The `BroadcastAndWait` brick, however, is blocking, meaning that the containing script is paused until all scripts listening to the message sent by the `BroadcastAndWait` brick have returned. Listings 5.7 and 5.8 show Cucumber scenarios demonstrating the usage of either broadcast brick. Notice the difference in the expected outcome. Figure 5.3 shows the corresponding Catroid program.

```
Feature: Demonstration of a Broadcast brick

  Scenario: A Broadcast brick sends its message and the containing
            script directly continues with its execution.

    Given I have a Program
    And this program has an Object 'test object'
    Given 'test object' has the following scripts:
      when program started
        broadcast 'message'
        print 'a'
      when I receive 'message'
        wait 0.1 seconds
        print 'b'

    When I start the program
    And I wait until the program has stopped
    Then I should see the printed output 'ab'
```

Listing 5.7: A Cucumber Scenario demonstrating the usage of a `Broadcast` brick.

```
Feature: Demonstration of a BroadcastAnd Wait brick

  Scenario: A BroadcastAndWait brick pauses the containing script until
            all scripts listening to the broadcast message have returned.

    Given I have a Program
    And this program has an Object 'test object'
    Given 'test object' has the following scripts:
      when program started
        broadcast 'message 1' and wait
        print 'a'
      when I receive 'message 1'
        wait 0.1 seconds
        print 'b'

    When I start the program
    And I wait until the program has stopped
    Then I should see the printed output 'ba'
```

Listing 5.8: A Cucumber Scenario demonstrating the usage of a `BroadcastAndWait` brick.

The broadcast system in Catroid is implemented using the "libgdx"[3] library. Catroid scripts can be thought of as "libgdx" `Actions`. Whenever a Catroid script is run the corresponding `BroadcastAction` fires a `BroadcastEvent`. A `BroadcastListener` listens for `BroadcastEvents` and notifies the class `BroadcastHandler` about the event. The `BroadcastHandler` reacts to the `BroadcastEvent` and takes care of it.

If the `BroadcastEvent` originates from a `BroadcastAndWait` brick special treatment is required. A special `Action` is appended to each script reacting to this very broadcast message – the `BroadcastNotifyAction` notifies the `BroadcastHandler` when the corresponding script has finished. After the `BroadcastNotifyAction` has been appended to the scripts they get notified that they now can start executing. Every script finishing execution then notifies the `BroadcastAndWait` brick. When all scripts are finished, the `BroadcastAndWait` has received a `BroadcastNotifyAction` from every

[3] `http://libgdx.badlogicgames.com` (visited on 2014-07-14)

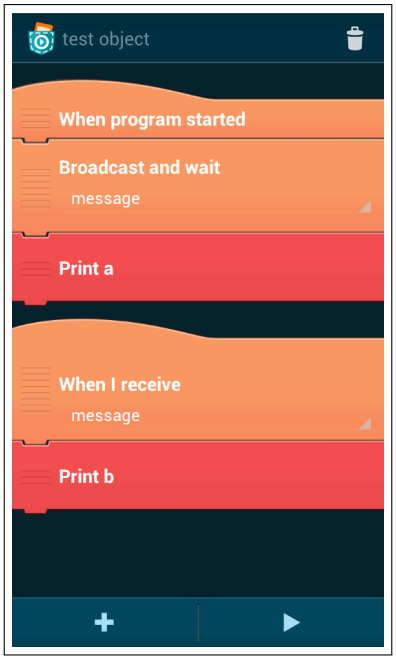


Figure 5.3: Catrobat program demonstrating the use of `BroadcastAndWait`. The output will be "ba" – the `BroadcastAndWait` brick doesn't continue execution until the script `WhenIReceive` finishes.

script. This signals the script containing the `BroadcastAndWait` brick to resume with its own execution.

5.4.2. Uncovering Misbehavior with Cucumber

Catroid endeavors to implement the behavior of the most language components just like Scratch. Scratch can therefore be taken as a reference for when it is not clear how some parts of the language should behave.

As an example the behavior of the `BroadcastAndWait` brick is analyzed. By definition the `BroadcastAndWait` brick sends out a broadcast message and pauses its own execution until all `WhenIReceive` scripts listening to this very broadcast message have completed. But how should the `BroadcastAndWait` brick react if the same broadcast message is sent again by a different `Broadcast` brick, while the `BroadcastAndWait` brick still waits for some scripts to finish? The answer can be found by creating a Scratch program, which reflects the desired behavior. The Scratch program displayed in figure 5.4 defines a "broadcast and wait" block which is activated right at the start of the program.

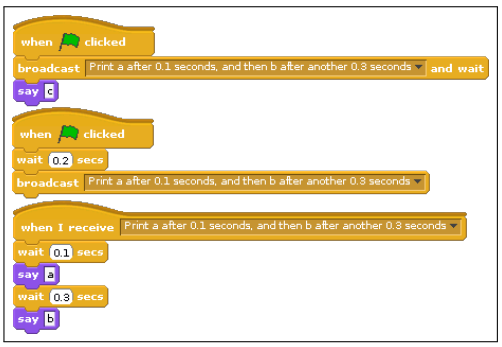


Figure 5.4: Scratch program defining a "broadcast and wait" block which gets interrupted by another broadcast.

A receiver script reacts to the broadcast message and reports its receipt. The broadcast message is then sent after 0.2 seconds from within another script, just after the receiver script has reported "a". Next "c" is output meaning that after the broadcast message is sent again the waiting block, waiting for the same message, is unblocked. The receiving script is also restarted. So the complete output is "acab" and the behavior of the `BroadcastAndWait` brick is clear. It stops waiting and resumes execution if it is in its waiting state and the same message is sent again.

See listing 5.3 as well as figure 5.2 for the specification of the desired behavior regarding the recurrence of the same broadcast message when there is already a `BroadcastAndWait` brick in a waiting state. The "Then" step is derived from the behavior of the corresponding Scratch program displayed in figure 5.4.

Upon execution of feature 5.3 Cucumber reports a failure (see listing 5.9). This indicates that the `BroadcastAndWait` brick behaves not like desired. When sending the broadcast message for the second time the receiver script is restarted, but the `BroadcastAndWait` brick does not continue execution. After correcting the behavior in Catroid, the test run finishes successfully, like already shown in figure 5.2. See section 5.4.3 for how the found misbehavior is corrected in Catroid. Section 5.5 discusses the specification of the broadcast mechanism of Catroid.

```
junit.framework.ComparisonFailure: The printed output is wrong.
  expected:<a[cab]> but was:<a[abc]>
at junit.framework.Assert.assertEquals(Assert.java:85)
at org.catrobat.catroid.test.cucumber.ProgramSteps.
  I_should_see_printed_output(ProgramSteps.java:361)
at org.catrobat.catroid.test.cucumber.ProgramSteps.
  I_should_see_printed_output_s(ProgramSteps.java:354)
at *.Then I should see the printed output 'acab'
  (features/bricks/BroadcastWaitBlockingBehavior.feature:56)
```

Listing 5.9: Failure upon execution of the Cucumber feature in listing 5.3.

5.4.3. Correcting Misbehavior of Catroid

Like described in section 5.4.2 some undesired behavior was uncovered in Catroid. The `BroadcastAndWait` brick did not correctly response to a broadcast message from another `Broadcast` brick when it already was in its waiting state. Upon uncovering this circumstance, an appropriate Cucumber scenario was created with assistance of Scratch. A Scratch program containing broadcast blocks and simulating the Catroid program in question was created and observed. From the behavior of the Scratch program the

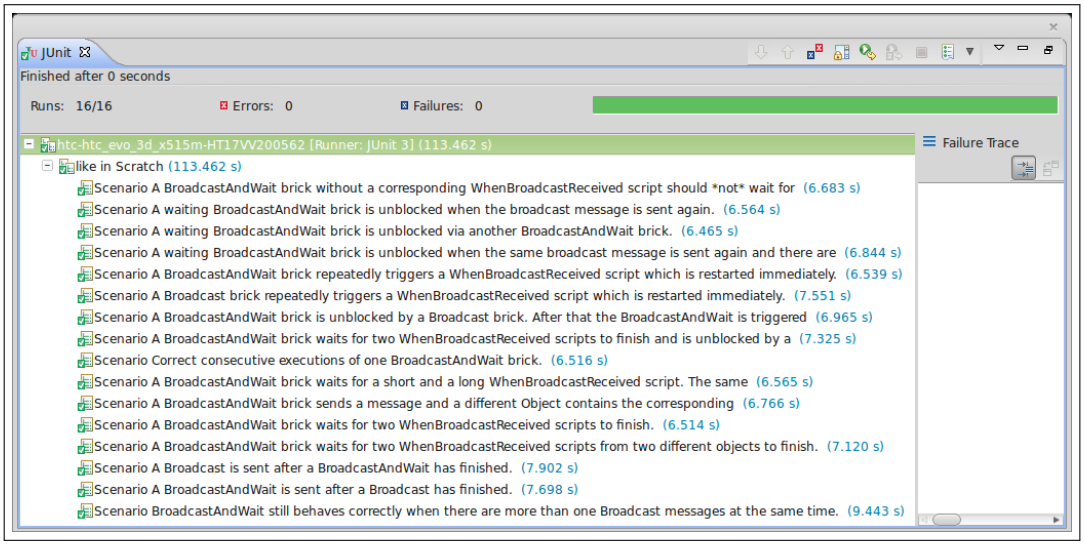


Figure 5.5: Full test run of the Cucumber feature in the IDE "Eclipse".

"Then" step of the Catroid program could be derived. Thus the incorrect behavior of Catroid could be corrected. Figure 5.5 shows a full test run of the Cucumber feature from within the integrated development environment (IDE) "Eclipse".

5.5. Specifying the Broadcast Mechanism of Catrobat by Example

This chapter describes the specification of the broadcast mechanism of Catrobat by the means of examples. Each scenario describes some particular situation and the expected behavior of Catrobat. The examples represent Catrobat programs.

The header of the Cucumber feature file with the "`Feature`" keyword and the feature description is shown in figure 5.10.

```
Feature: WhenBroadcastReceived blocking behavior (like in Scratch)

  If there exists no WhenBroadcastReceived script, a BroadcastAndWait
  should not wait at all. If there are one or more matching
  WhenBroadcastReceived scripts, execution of the script containing
  the BroadcastAndWait is paused until all WhenBroadcastReceived
  scripts are finished. If a broadcast is sent while a
  BroadcastAndWait brick is waiting for the same message, the
  responding WhenBroadcastReceived scripts is restarted; the
  BroadcastAndWait brick stops waiting and immediately continues
  executing the rest of the script. The same applies for a
  BroadcastAndWait brick which is unblocked by another
  BroadcastAndWait brick; the first one continues while the seconds
  one starts waiting. Just like a Broadcast brick, a BroadcastAndWait
  brick triggers all matching WhenBroadcastReceived in all Objects of
  the current program.
```

Listing 5.10: Description of the `BroadcastWaitBlockingBehavior` feature.

The background steps displayed in listing 5.11 are valid for all subsequent scenarios. Each scenario has a Catrobat program and at least one object called "`test object`".

```
Background:
  Given I have a Program
  And this program has an Object 'test object'
```

Listing 5.11: Background steps.

```
Scenario: A BroadcastAndWait brick without a corresponding
          WhenIReceive script should *not* wait for anything.

  Given Object 'test object' has the following scripts:
    when program started
      broadcast 'This message does not matter as there is no receiver
                 script' and wait
      print 'a'
    when program started
      wait 0.1 seconds
      print 'b'

  When I start the program
  And I wait until the program has stopped
  Then I should see the printed output 'ab'
```

Listing 5.12: A `BroadcastAndWait` brick without a corresponding `WhenIReceive` script.

The scenario in listing 5.12 simply demonstrates, that a `BroadcastAndWait` brick does not wait at all, if there is no corresponding `WhenIReceive` script present. The expected output is "`ab`" because the two `WhenProgramStarted` scripts run concurrently but the second one waits for 0.1 seconds before printing "`b`".

```
Scenario: A waiting BroadcastAndWait brick is unblocked when the
          broadcast message is sent again.

  Given Object 'test object' has the following scripts:
    when program started
      broadcast 'Print a after 0.1 seconds, and then b after another
                 0.3 seconds' and wait
      print 'c'
    when program started
      wait 0.2 seconds
      broadcast 'Print a after 0.1 seconds, and then b after another
                 0.3 seconds'
    when I receive 'Print a after 0.1 seconds, and then b after another
                    0.3 seconds'
      wait 0.1 seconds
      print 'a'
      wait 0.3 seconds
      print 'b'

  When I start the program
  And I wait until the program has stopped
  Then I should see the printed output 'acab'
```

Listing 5.13: When the same broadcast message is sent again, a waiting `BroadcastAndWait` brick gets unblocked.

The scenario in listing 5.13 shows, that a `BroadcastAndWait` brick is unblocked, when the broadcast message is sent again. First "`a`" gets printed. Then the second `WhenProgramStarted` script sends the same broadcast message again after 0.2 seconds. Because of that message the `BroadcastAndWait` brick is unblocked and "`c`" gets printed. Finally the `WhenIReceive` script is restarted and prints "`ab`".

```
Scenario: A waiting BroadcastAndWait brick is unblocked via another
          BroadcastAndWait brick.

  Given Object 'test object' has the following scripts:
    when program started
      broadcast 'Print b after 0.2 seconds' and wait
      print 'a'
    when program started
      wait 0.1 seconds
      broadcast 'Print b after 0.2 seconds' and wait
      print 'c'
    when I receive 'Print b after 0.2 seconds'
      wait 0.2 seconds
      print 'b'

  When I start the program
  And I wait until the program has stopped
  Then I should see the printed output 'abc'
```

Listing 5.14: A waiting `BroadcastAndWait` brick can also be unblocked by another `BroadcastAndWait` brick.

The scenario in listing 5.14 shows, that a `BroadcastAndWait` brick also is unblocked, when the broadcast message is sent again from another `BroadcastAndWait` brick. The first `BroadcastAndWait` brick sends its message and waits, but gets interrupted by the second `BroadcastAndWait` brick and therefore prints "`a`". The second `BroadcastAndWait` brick then waits for the `WhenIReceive` script, which prints "`b`", to finish. It then prints "`c`".

```
Scenario: A waiting BroadcastAndWait brick is unblocked when the
          same broadcast message is sent again and there are two
          WhenIReceive scripts responding to the same message.

  Given Object 'test object' has the following scripts:
    when program started
      broadcast 'Print b and c from different scripts' and wait
      print 'a'
    when I receive 'Print b and c from different scripts'
      wait 0.3 seconds
      print 'b'
    when I receive 'Print b and c from different scripts'
      wait 0.4 seconds
      print 'c'
    when program started
      wait 0.1 seconds
      broadcast 'Print b and c from different scripts'
      wait 0.2 seconds
      print 'd'

  When I start the program
  And I wait until the program has stopped
  Then I should see the printed output 'adbc'
```

Listing 5.15: A `BroadcastAndWait` brick is correctly unblocked when there are two `WhenIReceive` scripts.

The scenario in listing 5.15 shows, that a `BroadcastAndWait` brick is correctly unblocked, when there are two `WhenIReceive` scripts and the broadcast message is sent again. First, the `BroadcastAndWait` brick sends it broadcast message and waits for the scripts. It then gets interrupted by another broadcast message and continues with printing "`a`". Due to the waiting periods, "`d`" gets printed next. Finally, "`b`" and "`c`" are printed from the `WhenIReceive` scripts.

```
Scenario: A BroadcastAndWait brick repeatedly triggers a
          WhenIReceive script which is restarted immediately.

  Given Object 'test object' has the following scripts:
    when program started
      repeat 3 times
        broadcast 'Send the BroadcastAndWait message'
      end of loop
    when I receive 'Send the BroadcastAndWait message'
      broadcast 'Print a, then b after 0.1 seconds' and wait
      print 'c'
    when I receive 'Print a, then b after 0.1 seconds'
      print 'a'
      wait 0.1 seconds
      print 'b'

  When I start the program
  And I wait until the program has stopped
  Then I should see the printed output 'aaabc'
```

Listing 5.16: Repeatedly trigger a `WhenIReceive` script.

The scenario in listing 5.16 shows a `BroadcastAndWait` brick which repeatedly sends the broadcast message. The corresponding `WhenIReceive` script prints "`a`" and is then restarted due to the next broadcast message. When the `BroadcastAndWait` sends its broadcast message for the last time, it then waits for the `WhenIReceive` scripts to finish. The `WhenIReceive` scripts prints "`ab`" and, finally, "`c`" gets printed.

```
Scenario: A Broadcast brick repeatedly triggers a WhenIReceive
          script which is restarted immediately.

  Given Object 'test object' has the following scripts:
    when program started
      repeat 3 times
        broadcast 'Send the second broadcast message'
        wait 0.2 seconds
        print 'c'
      end of loop
    when I receive 'Send the second broadcast message'
      broadcast 'Print b after 0.1 seconds, then d after another 0.3
                 seconds'
      print 'a'
    when I receive 'Print b after 0.1 seconds, then d after another 0.3
                    seconds'
      wait 0.1 seconds
      print 'b'
      wait 0.3 seconds
      print 'd'

  When I start the program
  And I wait until the program has stopped
  Then I should see the printed output 'abcabcabcd'
```

Listing 5.17: A `Broadcast` brick repeatedly triggers a `WhenIReceive` script.

The scenario in listing 5.17 shows a `Broadcast` brick which repeatedly triggers a `WhenIReceive` script. The second `WhenIReceive` script is restarted whenever it receives the broadcast message and the sequence "`abc`" gets printed repeatedly. When the broadcast message is sent for the last time, the second `WhenIReceive` script finishes and finally prints "`d`".

```
Scenario: A BroadcastAndWait brick is unblocked by a Broadcast brick.
          After that the BroadcastAndWait is triggered again.

  Given Object 'test object' has the following scripts:
    when program started
      repeat 2 times
        broadcast 'Send the BroadcastAndWait message'
        wait 0.6 seconds
      end of loop
    when I receive 'Send the BroadcastAndWait message'
      broadcast 'Print a after 0.1 seconds, then b after another 0.2
                 seconds' and wait
      wait 0.2 seconds
      print 'd'
    when program started
      wait 0.3 seconds
      broadcast 'Print a after 0.1 seconds, then b after another 0.2
                 seconds'
      print 'c'
    when I receive 'Print a after 0.1 seconds, then b after another 0.2
                    seconds'
      wait 0.1 seconds
      print 'a'
      wait 0.4 seconds
      print 'b'

  When I start the program
  And I wait until the program has stopped
  Then I should see the printed output 'acadabd'
```

Listing 5.18: A `BroadcastAndWait` brick behaves correctly after it was interrupted.

The scenario in listing 5.18 demonstrates, that a `BroadcastAndWait` brick is correctly restarted after it was interrupted by a `Broadcast` brick. At first the `BroadcastAndWait` sends its broadcast message and waits for the `WhenIReceive` script to finish. The `WhenIReceive` script prints "`a`". Then the broadcast message is sent again, "`c`" gets printed, and the `BroadcastAndWait` brick gets unblocked. The next output is "`a`" from the restarted `WhenIReceive` script and "`d`" from the unblocked `BroadcastAndWait` brick. Then the second round of the `Repeat` brick starts and the `BroadcastAndWait` block sends

the broadcast message again. This time it does not get interrupted and the WhenIReceive script prints "ab". Finally, "d" gets printed.

```
Scenario: A BroadcastAndWait brick waits for two WhenIReceive
          scripts to finish and is unblocked by a Broadcast brick.
          After that the BroadcastAndWait is triggered again.

  Given Object 'test object' has the following scripts:
    when program started
      repeat 2 times
        broadcast 'Print a and b from two different scripts' and wait
        wait 0.3 seconds
        print 'c'
      end of loop
    when program started
      wait 0.1 seconds
      broadcast 'Print a and b from two different scripts'
    when I receive 'Print a and b from two different scripts'
      print 'a'
    when I receive 'Print a and b from two different scripts'
      wait 0.2 seconds
      print 'b'

  When I start the program
  And I wait until the program has stopped
  Then I should see the printed output 'aabcabc'
```

Listing 5.19: A BroadcastAndWait brick behaves correctly after it was interrupted and there are two WhenIReceive scripts present.

The scenario in listing 5.19 show, that a BroadcastAndWait brick correctly behaves after it was interrupted and there are two WhenIReceive scripts present. The BroadcastAndWait brick sends it message and waits for the WhenIReceive scripts to finish. The first WhenIReceive script prints "a". Then the broadcast message is sent again. The two WhenIReceive scripts are restarted and print "ab". The BroadcastAndWait brick then gets unblocked and "c" gets printed. In the second round of the Repeat brick, the BroadcastAndWait brick still behaves correctly. It waits for the WhenIReceive scripts to print "ab". Finally, "c" gets printed.

```
Scenario: Correct consecutive executions of one BroadcastAndWait brick.

  Given Object 'test object' has the following scripts:
    when program started
      repeat 2 times
        broadcast 'Print a immediately' and wait
        print 'b'
      end of loop
    when I receive 'Print a immediately'
      print 'a'

  When I start the program
  And I wait until the program has stopped
  Then I should see the printed output 'abab'
```

Listing 5.20: A `BroadcastAndWait` brick behaves correctly after it was triggered once.

The scenario in listing 5.20 shows, that a `BroadcastAndWait` brick correctly behaves after it was triggered once. When the `BroadcastAndWait` brick sends its broadcast message, the `WhenIReceive` script prints "`a`". The `WhenIReceive` script finishes, the `BroadcastAndWait` brick gets unblocked, and "`b`" gets printed. The `Repeat` brick starts the second round and the `BroadcastAndWait` brick still behaves correctly.

```
Scenario: A BroadcastAndWait brick waits for a short and a long
          WhenIReceive script. The same BroadcastAndWait is
          triggered again before the long one finishes.

  Given Object 'test object' has the following scripts:
    when program started
      repeat 2 times
        broadcast 'Send the BroadcastAndWait message'
        wait 0.2 seconds
      end of loop
    when I receive 'Send the BroadcastAndWait message'
      broadcast 'Print a immediately and b after 0.3 seconds' and wait
      print 'c'
    when I receive 'Print a immediately and b after 0.3 seconds'
      print 'a'
    when I receive 'Print a immediately and b after 0.3 seconds'
      wait 0.3 seconds
      print 'b'

  When I start the program
  And I wait until the program has stopped
  Then I should see the printed output 'aabc'
```

Listing 5.21: A `BroadcastAndWait` waits for a short and a long `WhenIReceive` script.

The scenario in listing 5.21 shows a `BroadcastAndWait` brick which waits for a short and a long `WhenIReceive` script to finish. When the broadcast message is sent, the first `WhenIReceive` script prints "a". The `BroadcastAndWait` brick sends its broadcast message again before the second `WhenIReceive` script finishes. The scripts are restarted properly and "ab" gets printed. Then the `BroadcastAndWait` brick gets unblocked and finally "c" gets printed.

```
Scenario: A BroadcastAndWait brick sends a message and a different
          Object contains the corresponding WhenIReceive script.

  Given Object 'test object' has the following script:
    when program started
      repeat 2 times
        print 'a'
        broadcast 'Print b immediately' and wait
        print 'c'
      end of loop

  And this program has an Object '2nd test object'
  Given Object '2nd test object' has the following script:
    when I receive 'Print b immediately'
      print 'b'

  When I start the program
  And I wait until the program has stopped
  Then I should see the printed output 'abcabc'
```

Listing 5.22: The `WhenIReceive` script is present in a different object.

The scenario in listing 5.22 demonstrates, how a `BroadcastAndWait` brick behaves, if the corresponding `WhenIReceive` script is present in a different object. At first "`a`" gets printed. Then the `BroadcastAndWait` brick sends its broadcast message and the `WhenIReceive` script from the second object reacts to it by printing "`b`". Then the `BroadcastAndWait` brick is unblocked and "`c`" gets printed. Finally, the sequence is run again to ensure still correct behavior.

```
Scenario: A BroadcastAndWait brick waits for two WhenIReceive
          scripts to finish.

  Given Object 'test object' has the following scripts:
    when program started
      print 'a'
      broadcast 'Print b and c from two different scripts' and wait
      print 'd'
    when I receive 'Print b and c from two different scripts'
      print 'b'
    when I receive 'Print b and c from two different scripts'
      wait 0.1 seconds
      print 'c'

  When I start the program
  And I wait until the program has stopped
  Then I should see the printed output 'abcd'
```

Listing 5.23: A `BroadcastAndWait` brick behaves correctly, when there are two `WhenIReceive` scripts.

The scenario in listing 5.23 simply shows, that a `BroadcastAndWait` brick behaves correctly, when there are more than one `WhenIReceive` scripts. At first "`a`" gets printed. Then the `BroadcastAndWait` brick sends its broadcast message and the `WhenIReceive` scripts print "`b`" and "`c`" respectively. Then the `BroadcastAndWait` brick is unblocked and "`d`" gets printed.

```
Scenario: A BroadcastAndWait brick waits for two WhenIReceive
          scripts from two different objects to finish.

  Given Object 'test object' has the following script:
    when program started
      print 'a'
      broadcast 'Print b and c from two different objects' and wait
      print 'd'

  And this program has an Object '2nd test object'
  Given Object '2nd test object' has the following script:
    when I receive 'Print b and c from two different objects'
      print 'b'

  And this program has an Object '3rd test object'
  Given Object '3rd test object' has the following script:
    when I receive 'Print b and c from two different objects'
      wait 0.1 seconds
      print 'c'

  When I start the program
  And I wait until the program has stopped
  Then I should see the printed output 'abcd'
```

Listing 5.24: Two `WhenIReceive` scripts are in two different objects.

The scenario in listing 5.24 shows, similar to the scenario in listing 5.23, that a `BroadcastAndWait` brick behaves correctly, when there are two `WhenIReceive` scripts. This time, the scripts are in two different objects. At first "a" gets printed. Then the `BroadcastAndWait` brick sends its broadcast message and the `WhenIReceive` scripts print "b" and "c" respectively. Then the `BroadcastAndWait` brick is unblocked and "d" gets printed.

```
Scenario: A Broadcast is sent after a BroadcastAndWait has finished.

  Given Object 'test object' has the following script:
    when program started
      print 'a'
      broadcast 'Print c immediately' and wait
      print 'b'

  And this program has an Object '2nd test object'
  Given Object '2nd test object' has the following script:
    when program started
      wait 0.3 seconds
      print 'd'
      broadcast 'Print c immediately'

  And this program has an Object '3rd test object'
  Given Object '3rd test object' has the following script:
    when I receive 'Print c immediately'
      print 'c'

  When I start the program
  And I wait until the program has stopped
  Then I should see the printed output 'acbdc'
```

Listing 5.25: A broadcast message is sent from a `Broadcast` brick after a `BroadcastAndWait` has finished.

The scenario in listing 5.25 shows that the broadcast system behaves correctly, when a `Broadcast` brick broadcasts a message, after a `BroadcastAndWait` brick already has sent the same broadcast message. At first "`a`" gets printed. Then the `BroadcastAndWait` brick sends its broadcast message and the `WhenIReceive` script prints "`c`". The `BroadcastAndWait` brick is unblocked and "`b`" gets printed. After 0.3 seconds "`d`" gets printed and a `Broadcast` brick sends the same broadcast message. Finally, "`c`" gets printed again.

```
Scenario: A BroadcastAndWait is sent after a Broadcast has finished.

  Given Object 'test object' has the following script:
    when program started
      wait 0.2 seconds
      print 'a'
      broadcast 'Print c immediately' and wait
      print 'b'

  And this program has an Object '2nd test object'
  Given Object '2nd test object' has the following script:
    when program started
      print 'd'
      broadcast 'Print c immediately'

  And this program has an Object '3rd test object'
  Given Object '3rd test object' has the following script:
    when I receive 'Print c immediately'
      print 'c'

  When I start the program
  And I wait until the program has stopped
  Then I should see the printed output 'dcacb'
```

Listing 5.26: A broadcast message is sent from a `BroadcastAndWait` brick after a `Broadcast` has finished.

The scenario in listing 5.25 is similar to the scenario in listing 5.26. This time the broadcast message is first sent from the `Broadcast` brick. At first "`d`" gets printed. Then the `Broadcast` brick sends the broadcast message and "`c`" gets printed. Then the sequence "`acb`" gets printed when the `BroadcastAndWait` brick and the `WhenIReceive` script interact.

```
Scenario: BroadcastAndWait still behaves correctly when there are
          more than one Broadcast messages at the same time.

  Given Object 'test object' has the following scripts:
    when program started
      broadcast 'Run second script' and wait
      print 'a'
    when I receive 'Run second script'
      broadcast 'Run third script' and wait
      print 'b'
    when I receive 'Run third script'
      broadcast 'Run fourth script' and wait
      print 'c'
    when I receive 'Run fourth script'
      print 'd'

  When I start the program
  And I wait until the program has stopped
  Then I should see the printed output 'dcba'
```

Listing 5.27: `BroadcastAndWait` chain.

The scenario in listing 5.27 shows, that the broadcast mechanism works correctly, when there are more `BroadcastAndWait` bricks waiting at the same time. The first `BroadcastAndWait` brick sends its broadcast message and waits for the corresponding `WhenIReceive` script to finish. The corresponding `WhenIReceive` script contains another `BroadcastAndWait` brick which sends another broadcast message. This sequence continues until the last `WhenIReceive` script finally prints "`d`" and finishes. The waiting `BroadcastAndWait` brick unblocks and "`c`" gets printed. This sequence goes on until "`b`" and "`a`" are printed and all scripts are finished.

```
Scenario: A BroadcastAndWait brick repeatedly triggers two
          WhenIReceive scripts

  Given Object 'test object' has the following script:
    when program started
      repeat 3 times
        broadcast 'Print a from different scripts' and wait

  And this program has an Object '2nd test object'
  Given Object '2nd test object' has the following script:
    when I receive 'Print a from different scripts'
      print 'a'

  And this program has an Object '3rd test object'
  Given Object '3rd test object' has the following script:
    when I receive 'Print a from different scripts'
      print 'a'

  When I start the program
  And I wait until the program has stopped
  Then I should see the printed output 'aaaaaa'
```

Listing 5.28: A `BroadcastAndWait` brick interacts with two `WhenIReceive` scripts

The scenario in listing 5.28 shows the interaction of a `BroadcastAndWait` brick with two `WhenIReceive` scripts, which are located in two different objects. The `BroadcastAndWait` brick sends its broadcast message and the two scripts print "a" each. This cycle is repeated three times to ensure the correct restarting of the `BroadcastAndWait` brick.

6. Conclusion and Outlook

The previous chapters have provided an overview over the concept of behavior-driven development. The visual programming language Catrobat and the behavior-driven testing framework Cucumber were used to demonstrate some of the key ideas of specification by example. The broadcast system of Catrobat was thoroughly specified by examples. Some misbehavior in the existent system could be uncovered and corrected.

6.1. Workflow of Behavior-Driven Testing with Cucumber in Catroid

To utilize Cucumber in Catroid the following workflow is suggested. This workflow follows the principles of TDD. Cucumber features are written before any changes to the existing system are made.

1. **Derive scope from goals:**
   This step is based on the first step of the key process patterns defined by Adzic [3]. For Catroid this may mean to observe some behavior in Scratch and decide to adapt the behavior of Catroid alike. Adapting the behavior of Catroid to conform to Scratch might also denote the "business value" for this step.

2. **Create Cucumber feature:**
   The next step would be to formulate a Cucumber feature containing one or more scenarios. When speaking about Catrobat programs, the text-based language, which was introduced in section 3.4 should be

used. During this stage of the workflow, some Cucumber step definitions might be reused, for example for adding bricks to a Catrobat program.

3. **Implement missing step definitions:**
   Some of the step definitions might be reusable from other Cucumber features. Step definitions which are sill missing have to be implemented.

4. **Convert the steps to Gherkin language:**
   When using the text-based language from section 3.4 like described in section 5.3 to specify the steps of a Catrobat program, the Gherkin keywords are missing. To add them back, adapt and use the script shown in listing 5.4. In order to automate as many tasks as possible and keep validation time as short as possible, this step could be carried out automatically once the Cucumber framework is fully integrated in the Catroid testing process.

5. **Run the Cucumber feature:**
   Running the Cucumber feature would probably result in a failure as the Catroid project has not been adapted yes. However, if specifying some already existing behavior, the test run might already succeed at this stage.

6. **Implement the specified behavior:**
   At this stage the code to make the Cucumber tests pass should be written.

7. **Run the Cucumber feature again:**
   Now the Cucumber feature should pass and the system should behave as specified.

8. **Run all Cucumber features:**
   To make sure that the new added code did not break something else this step is necessary. If all Cucumber features succeed at this stage the system behaves as specified and nothing else broke.

6.2. Further Examples of Using Cucumber for Specifying Catroid Elements

It has been demonstrated in chapter 5 that the broadcast mechanism of Catroid is easily specifiable by the means of examples using the behavior-driven testing framework Cucumber. This section provides additional examples of Cucumber features, specifying Catroid bricks, as a starting point for future specifications. The text-based Catrobat language defined in section 3.4 is used for specifying exemplary Catrobat programs.

6.2.1. `Hide` Brick

```
Feature: Hide brick

  A brick setting its Object invisible.

  Background:
    Given I have a Program
    And this program has an Object 'test object'

  Scenario: After executing the Hide brick the object should not be
            visible

    Given 'test object' is visible
    And 'test object' has the following script:
      when program started
      hide

    When I start the program
    And I wait for at least 200 milliseconds
    Then I should not see 'test object'
```

Listing 6.1: A Cucumber feature specifying the behavior of a `Hide` brick.

Listing 6.1 shows a Cucumber feature with one scenario specifying a Catrobat program with a `Hide` brick. This feature demonstrates, how Cucumber

can be used to specify the behavior of Catrobat bricks. Instead of describing the procedure of checking the visibility of an object in technical terms, the phrase can simple be stated as "`Then I should not see 'test object'`". Of course the technical representation is still present, but abstracted and put out of the way into the step definitions.

6.2.2. `If` Brick

```
Feature: If brick

  An If brick decides which path of execution to follow depending on a
  condition.

  Background:
    Given I have a Program
    And this program has an Object 'test object'

  Scenario Outline: An If brick decides on which path to take base on
                    the value of a user variable

    Given 'test object' has the following script:
      when program started
        set variable '<variable>' to '<value>'
        if '<condition'> is true then
          print 'if path'
        else
          print 'else path'
        end if

    When I start the program
    And I wait until the program has stopped
    Then I should see the printed output '<path>'

    Examples:
      | variable     | value | condition          | path      |
      | myVariable   | 5.3   | myVariable > 5.3   | else path |
      | yourVariable | 2     | yourVariable = 2.0 | if path   |
      | ourVariable  | 4.3   | ourVariable < 1    | else path |
```

Listing 6.2: A Cucumber feature specifying the behavior of an `If` brick.

The Cucumber feature in listing 6.2 demonstrates conditional execution in Catroid. The concept of user variables is also exemplified. Via a scenario outline several configurations of input values can be tested (the concept of scenario outline is introduced in section 4.5.2).

6.2.3. `PlaceAt` Brick

```
Feature: PlaceAt brick

  A brick placing an Object at a given position

  Background:
    Given I have a Program
    And this program has an Object 'test object'

  Scenario Outline: Place an Object at a given position

    Given 'test object' the following script:
      when program started
        place at X: <xPosition>, Y: <yPosition>

    When I start the program
    And I wait until the program has stopped
    Then 'test object' should be at position <xPosition> <yPosition>

    Examples:
      | xPosition   | yPosition   |
      | 200         | 100         |
      | 2147483647  | 2147483647  |
      | -2147483648 | -2147483648 |
```

Listing 6.3: A Cucumber feature specifying the behavior of a `PlaceAt` brick.

A feature specifying a `PlaceAt` brick can be viewed in listing 6.3. This feature uses a scenario outline to test several sets of input values. It demonstrates the use of corner cases to properly specify the behavior of the `PlaceAt` brick for values corresponding to the Java constants `Integer.MAX_VALUE` and `Integer.MIN_VALUE`.

6.2.4. `WhenIReceive` Script

```
Feature: Restart WhenIReceive script

  A WhenIReceive script should be restarted when the message
  is broadcast again from within the script.

  Background:
    Given I have a Program
    And this program has an Object 'test object'

  Scenario: A WhenIReceive script is restarted when the broadcast message
            is sent again from within the script

    Given Object 'test object' has the following scripts:
      when program started
        broadcast 'print a immediately and send broadcast message again'
      when I receive 'print a immediately and send broadcast message again'
        print 'a'
        broadcast 'print a immediately and send broadcast message again'

    When I start the program
    And I wait for at least 1 second
    Then I should see at least 10 times 'a'
```

Listing 6.4: A Cucumber feature specifying the behavior of a `WhenIReceive` script.

The feature shown in listing 6.4 specifies the behavior of a `WhenIReceive` script when the same broadcast message is sent again from within the script. Because the broadcast message is sent from within the `WhenIReceive` script the script gets restarted immediately. This sending and restarting process would continue endlessly.

6.3. Advantages and Disadvantages of Using Cucumber as a Behavior-Driven Framework in Catroid

The preceding chapters discuss many aspects of using Cucumber as a behavior-driven framework in Catroid. This section sums up the advantages and disadvantages of specifying Catroid by example.

One of the main advantages is, that Cucumber scenarios can be written and read by every single person involved with Catroid. Every developer, tester, designer, and manager can easily maintain existing scenarios as well as create new ones. The scenarios are written in a ubiquitous language, which every member of the Catroid team understands. New team members are able to understand the system faster by reading the scenarios. The text-based Catrobat language defined in section 3.4 is used when specifying Catrobat programs.

The necessity of writing new step definitions will decrease over time because existing step definitions can be reused in further scenarios. The specification of the behavior of a certain brick is defined once and can be used over again.

The external quality of Catroid can he held upright by using Cucumber scenarios as examples. This means that the behavior of Catroid can be adjusted to the behavior of Scratch. The examples describe how Scratch behaves in certain situations which means that Catroid should behave alike.

However, there are also drawbacks in the introduction of Cucumber into Catroid. First, the Cucumber framework can coexist with the already existent unit tests. But this requires a strict policy about what to test through unit tests and what to test via Cucumber scenarios. It also is tedious to maintain two testing systems in parallel. Another option would be, to establish Cucumber as solely testing system in Catroid. Yet this would mean to rewrite all the unit tests into Cucumber scenarios. Due to the large amount of existent tests this would require many hours of developers' time.

To sum up, a practicable solution would be to keep and maintain the unit tests of Catroid to guarantee a stable internal quality level of the project, and establish the Cucumber testing-framework additionally. This would also guarantee a high external quality level of Catroid. If the Cucumber tests and the unit tests overlap more and more over time, the number of unit tests could be reduced and eventually the Cucumber framework establishes as exclusive testing framework in Catroid.

6.4. Future Work

As has been shown in the previous chapters the behavior of the broadcast system of the Catrobat language is easily specifiable by the means of Cucumber features. A living documentation has evolved from examples. The examples are mostly text-based Catrobat programs which can easily be read and understood by all people involved in Catrobat. Following those examples, other parts of Catrobat might also be straightforward specifiable by the means of examples.

The Catrobat team might be interested in expanding the language in the future. Self defined blocks, like in Snap!, would be an exciting enhancement to have in the Catrobat language. The specification of such an enhancement could be compassed by the means of examples. The examples could be written in Gherkin and tested with the already present Cucumber testing framework.

As steps from different features can share the step definitions, the organization of the step definitions into files has to be reconsidered. It could be an option to have one file per brick.

The Cucumber testing framework was already introduced into Catrobat. The integration into the Jenkins test environment was also set up and tested manually. What is, however, still missing is the automatic execution of the Cucumber features upon each Jenkins test run. It needs to be considered, that adding the Cucumber tests might extend the testing period. But the Cucumber test suite could be executed on an emulated device and run in

parallel to the already existing unit tests. Putting this proposed set up into practice, the additional tests would not prolong the testing period at all.

A noble goal for future work would also be to implement the Cucumber framework for other platforms. Currently there exist implementations of the Catroid programming language, besides the Android version, for Microsoft Windows Phone and Apple iOS. There is also an HTML5 version in development. Cucumber features could be specified once and serve as reference for the implementation of the steps for all platforms. Hence, the same behavior of the language could be verified on all different implementations of Catrobat.

Bibliography

[1] Gojko Adzic. *Bridging the Communication Gap. Specification by Example and Agile Acceptance Testing*. Neuri Limited, Jan. 2009. ISBN: 9780955683619.

[2] Gojko Adzic. *Examples make it easy to spot inconsistencies*. May 2009. URL: http://gojko.net/2009/05/12/examples-make-it-easy-to-spot-inconsistencies/. (visited on 2014-07-14).

[3] Gojko Adzic. *Specification by Example: How Successful Teams Deliver the Right Software*. Manning Publications, 2011. ISBN: 9781617290084.

[4] David Astels. *Test-Driven Development: A Practical Guide*. Prentice Hall, 2003. ISBN: 9780131016491.

[5] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999. ISBN: 9780201616415.

[6] Kent Beck. *Test-Driven Development: By Example*. Addison-Wesley, 2002. ISBN: 9780321146533.

[7] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. *Manifesto for Agile Software Development*. 2001. URL: http://agilemanifesto.org/. (visited on 2014-07-14).

[8] Gilad Bracha. *Pluggable Type Systems*. OOPSLA Workshop on Revival of Dynamic Languages. Oct. 2004.

[9] Luca Cardelli. "Type Systems." In: *Handbook of Computer Science and Engineering*. Ed. by Allen B. Tucker. CRC Press, 1997. Chap. 103.

[10] David Chelimsky, Dave Astels, Zach Dennis, Aslak Hellesøy, Bryan Helmkamp, and Dan North. *The Rspec Book: Behaviour-Driven Development with Rspec, Cucumber, and Friends*. Pragmatic Bookshelf, 2010. ISBN: 9781934356371.

[11] Alistair Cockburn. *Crystal Clear A Human-Powered Methodology for Small Teams*. Addison-Wesley Professional, 2004. ISBN: 9780201699470.

[12] Ian Dees, Matt Wynne, and Aslak Hellesøy. *Cucumber Recipes: Automate Anything with BDD Tools and Techniques*. Pragmatic Bookshelf, 2013. ISBN: 9781937785017.

[13] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2003. ISBN: 9780321125217.

[14] Steve Freeman and Nat Pryce. *Growing Object-Oriented Software, Guided by Tests*. Addison-Wesley Professional, 2009. ISBN: 9780321574442.

[15] Robert Harper. *Practical Foundations for Programming Languages (Draft)*. 2nd ed. Cambridge University Press, 2014. ISBN: 9781107029576.

[16] Brian Harvey and Jens Mönig. *Snap! Build Your Own Blocks*.

[17] Annemarie Harzl, Vesna Krnjic, Franz Schreiner, and Wolfgang Slany. "Comparing Purely Visual with Hybrid Visual/Textual Manipulation of Complex Formula on Smartphones." In: *DMS*. Knowledge Systems Institute, 2013, pp. 198–201. ISBN: 1891706349. URL: `http://dblp.uni-trier.de/db/conf/dms/dms2013.html#HarzlKSS13`.

[18] Aslak Hellesøy and Matt Wynne. *The Cucumber Book: Behaviour-Driven Development for Testers and Developers*. Pragmatic Bookshelf, 2012. ISBN: 9781934356807.

[19] Elizabeth Keogh. "BDD: A Lean Toolkit." In: *Lean Software & Systems Conference*. Atlanta, GA, USA, 2010.

[20] Lasse Koskela. *Test Driven: Practical TDD and Acceptance TDD for Java Developers*. Manning Publications, 2007. ISBN: 9781932394856.

[21] Craig Larman and Bas Vodde. *Practices for Scaling Lean & Agile Development: Large, Multisite, and Offshore Product Development with Large-Scale Scrum*. Addison-Wesley Professional, 2010. ISBN: 9780321685117.

[22] Ioan Lazăr, Simona Motogna, and Bazil Pârv. "Behaviour-Driven Development of Foundational UML Components." In: *Electronic Notes in Theoretical Computer Science* 264.1 (2010). Proceedings of the 7th International Workshop on Formal Engineering approaches to Software Components and Architectures (FESCA 2010), pp. 91–105. ISSN: 1571-0661. URL: http://www.sciencedirect.com/science/article/pii/S1571066110000666.

[23] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. "The Scratch Programming Language and Environment." In: *ACM Transactions on Computing Education* 10.4 (Nov. 2010), 16:1–16:15. ISSN: 1946-6226. DOI: 10.1145/1868358.1868363. URL: http://doi.acm.org/10.1145/1868358.1868363.

[24] Steve McConnell. *Code Complete*. 2nd ed. Microsoft Press, 2004. ISBN: 9780735685819.

[25] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. 3rd ed. John Wiley & Sons, Nov. 2011. ISBN: 9781118031964.

[26] Dan North. *Introducing BDD*. Mar. 2006. URL: http://dannorth.net/introducing-bdd/. (visited on 2014-07-14).

[27] Dan North. *What's in a Story?* Feb. 2007. URL: http://dannorth.net/whats-in-a-story/. (visited on 2014-07-14).

[28] David Lorge Parnas. "A Technique for Software Module Specification with Examples." In: *Communications of the ACM* 15.5 (May 1972), pp. 330–336. ISSN: 0001-0782. DOI: 10.1145/355602.361309. URL: http://doi.acm.org/10.1145/355602.361309.

[29] Kenneth Pugh. *Lean-Agile Acceptance Test-Driven Development: Better Software Through Collaboration*. Addison-Wesley Professional, 2011. ISBN: 9780321714084.

[30] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. "Scratch: Programming for All." In: *Communications of the ACM* 52.11 (Nov. 2009), pp. 60–67. ISSN: 0001-0782. DOI: 10.1145/1592761.1592779. URL: http://doi.acm.org/10.1145/1592761.1592779.

[31] Wolfgang Slany. "A mobile visual programming system for Android smartphones and tablets." In: *2012 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. Sept. 2012, pp. 265–266. DOI: 10.1109/VLHCC.2012.6344546.

[32] Wolfgang Slany. "Catroid: A Mobile Visual Programming System for Children." In: *Proceedings of the 11th International Conference on Interaction Design and Children*. IDC '12. New York, NY, USA: ACM, 2012, pp. 300–303. ISBN: 9781450310079. DOI: 10.1145/2307096.2307151. URL: http://doi.acm.org/10.1145/2307096.2307151.

[33] Carlos Solís and Xiaofeng Wang. "A Study of the Characteristics of Behaviour Driven Development." In: *2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*. Aug. 2011, pp. 383–387. DOI: 10.1109/SEAA.2011.76.

[34] Laurence Tratt. "Dynamically Typed Languages." In: *Advances in Computers* 77 (July 2009). Ed. by Marvin V. Zelkowitz, pp. 149–184.

Appendix A.

Acronyms

ATDD acceptance test-driven development

BDD behavior-driven development

TDD test-driven development

SbE specification by example

UI user interface

XP Extreme Programming

CI continuous integration