Maximilian Fellner, BSc.

Specification of a Visual Programming Language by Example

Master's Thesis

Graz University of Technology

Institute for Software Technology

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

Graz, November 2013

Statutory Declaration

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

Graz, _____ _____

Date Signature

Eidesstattliche Erklärung[1]

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

Graz, am _____ _____

Datum Unterschrift

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

Zusammenfassung

Spezifikationen stellen sicher, dass alle Interessengruppen in einem Softwareprojekt gleichermaßen gut verstehen, welche Anforderungen es zu erfüllen gibt. Eine Dokumentation beschreibt genau, wie das fertige Produkt aussehen sollte. Präzise Spezifikationen helfen Mehrarbeit zu reduzieren, die durch Unklarheiten verursacht wurde. Außerdem stellen sie ein objektives Maß für den gesamten Fortschritt dar. Moderne Softwareprojekte haben kürzere Projektphasen und schnellere Iterationen, die es schwierig machen, eine nützliche und prägnante Spezifikation aufzubauen und richtig instand zu halten. Der Ansatz von "Spezifikation durch Beispiele" ist ein Versuch, mit der Hilfe von agilen Methoden Software Spezifikation direkt in den Arbeitsfluss zu integrieren. Verhaltensgetriebene Entwicklung (BDD) ist eine konkrete Erscheinungsform der Prozessmuster dieses Ansatzes und eine weit verbreitete Methodologie, welche mehrere nützliche Software Werkzeuge mit einschließt. In der vorliegenden Arbeit wird ein neuartiger Ansatz zur Spezifikation einer visuellen Programmiersprache unter Verwendung von Cucumber, einem populärem BDD Werkzeug, vorgestellt. Die visuelle Programmiersprache Catrobat wird für mehrere unterschiedliche mobile Betriebssysteme von gesonderten Entwicklerteams aktiv entwickelt. Die Plattformunabhängigkeit dieses Softwareprojekts verlangt nach einer automatisch verifizierbaren Spezifikation, die von allen Interessengruppen geteilt werden kann. Ausführbare Spezifikationsdokumente, die mittels Cucumber verfasst werden können, erfüllen diese Anforderung. Die Verwendung dieses Werkzeugs hat darüber hinaus auch noch zusätzliche Vorteile. Um ein solches Argument zu untermauern, stellt diese Arbeit die zugrunde liegenden Konzepte von BDD vor, betrachtet Beispiele von allgemeiner Softwarespezifikation und erklärt schließlich anhand von realen Beispielen, warum Cucumber für die plattformübergreifende Spezifikation einer visuellen Programmiersprache geeignet ist.

Abstract

Specifications assure that all stakeholders in a software project understand equally well which requirements need to be fulfilled. Documentation describes exactly what the finished product should look like. Precise specifications help to reduce extra work caused by ambiguities. They also provide an objective measure for overall progress. Modern software projects have shorter project phases and faster iterations, which make it difficult to build up and properly maintain a useful and concise specification. The approach of "specification by example" is an attempt at integrating software specification directly into the development workflow with the help of agile techniques. Behavior-driven development (BDD) is a manifestation of these process patterns and a widely used methodology which includes several useful software tools. In the present work, a novel approach of specifying a visual programming language by using Cucumber, a popular BDD tool, is introduced. The visual programming language Catrobat is being actively developed for multiple different mobile operating systems by separate teams of developers. The cross-platform nature of this software project requires an automatically verifiable specification which can be shared by all stakeholders. Executable specification documents that can be composed by means of Cucumber meet this requirement. Beyond that, the application of this tool also has additional benefits. In order to support such an argument, this work introduces the underlying concepts of BDD, looks at examples of software specification in general, and finally explains on the basis of real-world examples why Cucumber is suitable for a cross-platform specification of a visual programming language.

Contents

List of Figures

1 Introduction

Good practices are oftentimes only an afterthought for programmers. Although the field of software development has over the last few decades brought forth a plethora of methodologies, they are not always considered. One such good practice would be specification. Designating requirements and documenting the details of their implementation is essential for a successful software project. Notwithstanding its importance, this work is commonly regarded as just a necessary burden. But specification can actually be a very helpful tool that supports the development process and solves many problems outright.

During the past few decades, the software development community has focused mainly on technical practices in order to ensure results of high quality and to *build* the product right. But it is equally important to build the *right product*. This task requires different approaches and techniques however.

The visual programming language Catrobat is a software project which poses many challenges. Implementations of this language are being simultaneously developed for different mobile computing devices which are running varied operating systems. The high goal of Catrobat is to give users a solid and consistent experience regardless of the underlying platform. Like every other programming language, Catrobat too requires a semantical specification in order to provide an official and reliable guideline for implementors.

Without a common specification that is shared by all stakeholders of the project, separate development teams are likely to drift apart. Functional gaps which result from ambiguities are a great risk that can cause delays and other issues. For a specification to be really useful it should also be verifiable automatically. One possible solution is the use of test suites, which will be discussed in some of the

following sections. A disadvantage of such test suites is the reliance on a written specification from which the tests are derived. Thus, a probably even better approach would be to somehow combine the specification documents and the tests.

For these reasons, the subsequent chapters will explore the concept of machine executable specifications and the practice of behavior-driven development. Finally, the last chapter will explain how the behavior-driven test framework Cucumber has been used to specify parts of the Catrobat programming language.

2 Machine-executable specifications

2.1 Terminology

In other literature, a number of terms used in this work are sometimes used synonymously. Peculiar technical terms can oftentimes be mystifying and might confuse different concepts. In order to prevent confusion, the following terminology will be used consistently.

Specification by example (SbE) is a set of process patterns that assist in the creation and modification of software products. The term was coined by Gojko Adzic in his 2011 book *Specification by Example: How Successful Teams Deliver the Right Software* [2]. This approach aims to ensure that the right product, as defined by the business stakeholders, is delivered by collaboratively creating a specification that comprises numerous machine-executable examples. Ultimately, the goal of SbE is to create a living, structured documentation.[1] There can be different manifestations of this concept; the two most important ones are explained in the following.

Acceptance test-driven development (ATDD) is a method that derives software tests from collectively conceived requirements. Acceptance tests ought to capture the business intent of certain software features.[2] They are easily put to use in an existing test-driven development paradigm but there is no technical framework that can support the correct employment of process patterns.

[1] http://specificationbyexample.com/key_ideas.html (accessed 2013-10-05)
[2] http://testobsessed.com/2008/12/acceptance-test-driven-development-atdd-an-overview

Behavior-driven development (BDD) is a software development process and agile methodology based on test-driven development. It mandates software units be specified in terms of the desired behavior of the units. The technique outlines a formal format for behavioral specification which is adopted from classical user stories. The software behavior is specified in a specialized, domain-specific language that can be understood by all stakeholders. BDD also places certain demands on the software tools that automate reading the specification and executing the associated test code with the appropriate parameters.[3]

2.2 Specification by example

With the increasing dependency of society on information technology, the surge of personal computing devices and the necessity for higher quality and more secure systems, the demands on software development keep rising. In this industry the ability to deliver the right product in the shortest possible time is quickly becoming a key competitive advantage. In response, agile development has in recent years defined the notion of quick, iterative refinement while working in effectively organized teams.

Keeping a software system's documentation up-to-date under these conditions has become more and more challenging. As a result, creating and maintaining documentation is now often considered wasteful and obstructive. Unfortunately, this can lead to confusion among developers and stakeholders and result in copious amounts of lost time. Instead of building the *right product,* most programmers have just focused on how to build it right. However, for a software product to be successful, both requirements need to be fulfilled (Figure 2.1).

In his book, Gojko Adzic introduces *specification by example* (SbE) as a set of process patterns that "allows teams to define expected functionality in a clear, objective, and measurable way. It also speeds up feedback, improving the development flow and preventing interruptions to planned work" [2]. In practice, the essence of

[3]http://dannorth.net/introducing-bdd (accessed 2013-10-05)

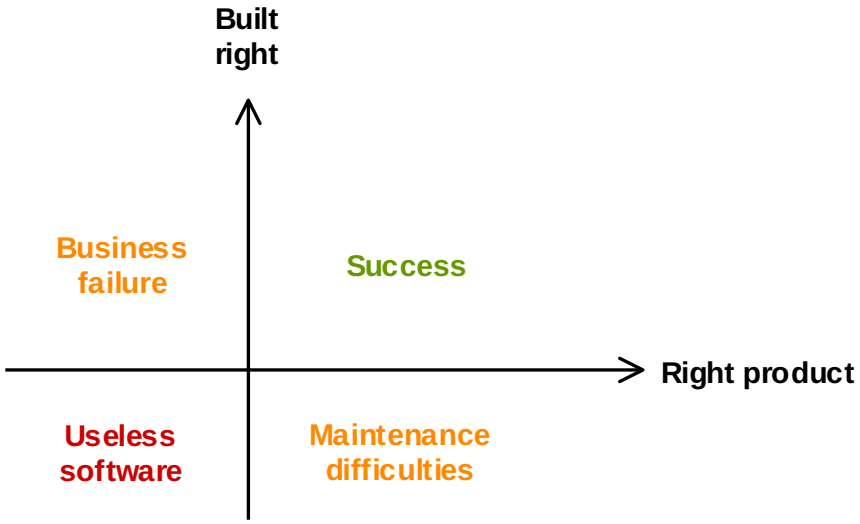

Figure 2.1: Success of a software product, adapted from *Adzic* [2]

this approach is to create a living documentation system through automating the verification of a collaboratively authored specification. For a better understanding, Adzic goes on to introduce two different models that actualize the principles of SbE.

Acceptance test-driven development (ATDD) is an advanced form of test-driven development where unit tests are directly derived from criteria specified by the stakeholders or customers. The focus of this technique lies on the automated tests and on defining straightforward targets for development [26], [17]. The testing matrix by Gerard Meszaros in Figure 2.2 shows the different kinds of software tests.

Acceptance tests (or *customer tests*) are at the business-facing end of the scale, because their purpose is to ensure that the product is acceptable to the customer. Their distinguishing trait is that the behavior specified by the test is understandable by an end user.

The second model is known by the name of behavior-driven development (BDD). It focuses on the interaction of the software system with its stakeholders and the interplay of the system's components with each other. Unlike acceptance tests,

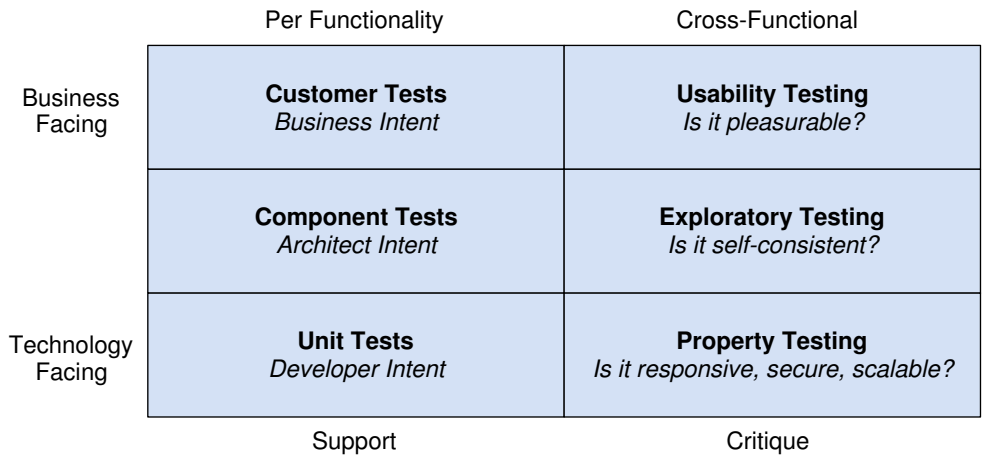

Figure 2.2: The testing matrix, adapted from *Meszaros* [22]

this methodology defines a definitive workflow for describing a software in terms of its expected behavior instead of its structure. The behavioral specification is accomplished with *stories*, related to *user stories* from the domain of object-oriented analysis and design. User stories, or story cards, are also a fundamental element of Extreme Programming (XP) to describe software requirements. XP was introduced by Kent Beck [6] as one of the first agile development methods.

A BDD story is composed of three distinct elements: a **title** that is unambiguous and concise, a **narrative** which explains what a stakeholder wants from the system, and **acceptance criteria** which comprise examples of specific cases of the narrative.

Stories are written in a so-called *ubiquitous language* that is shared by everyone involved with the development of the software product. Its syntax and grammar are understandable by both developers and non-technical stakeholders. The language also has specialized tooling support in order to make a specification written in this format executable.

The commonality shared by both approaches is the way the scope of the software system that has to be built is derived from business goals and subsequently illustrated via key examples. These examples make up a written specification that can be verified by an automated system.

Specifying software by the means of examples was suggested as early as 1972 [25]. The renewed interest in the technique by the agile software development community has nevertheless been relatively recent. Writing an example in a natural or ubiquitous language is usually faster than implementing a feature in code. Using examples for specification is also reasonable, because experience has shown that it makes it easier to avoid ambiguity and redundancy [1].

The underlying requirements for examples are that they need to be precise but also comprehensive enough to describe the entire scope of a certain feature. Furthermore, examples should be easily understandable and realistic, i.e., they should not be abstracted or simplified, but rather use authentic data like the software system would in a real-world use case.

2.3 Behavior-driven development

Like test-driven development with acceptance tests, behavior-driven development is a manifestation of SbE. In this case however, the underlying ideas are realized with a precise workflow and by providing useful software tools that make the whole process tangible for both engineers and business stakeholders.

2.3.1 History of BDD

BDD's origins are tightly interwoven with the evolution of a number of software tools. The British software engineer Dan North first introduced behavior-driven development in the year 2006 [23]. The initial idea for this at the time unprecedentedly agile technique came from a tool created by a coworker of North. The tool would automatically translate the names of JUnit classes and methods into a structured text document.

North's recognition of the necessity for expressive naming conventions to describe the behavior of single units in traditional test-driven development subsequently

led to the creation of a specialized language which is easily understood but also executable by a computer. Soon thereafter, this concept of a "ubiquitous language" was introduced in the Java test-framework JBehave.[4]

Later on, North reimplemented JBehave in the Ruby programming language and called it RBehave. This software was eventually integrated into another testing tool, RSpec, as a so-called "story runner." It only supported stories written in Ruby at first, but support for plain text was added later on, thus making the tool more accessible and expressive [8].

2.3.2 Ubiquitous language and story

Software requirements are most easily formulated in natural language. However, technical and non-technical people tend to use different jargons which can lead to difficulties in communication. Furthermore, programming languages have a smaller vocabulary than natural language and thus make it unintuitive to express specified requirements. Eric Evans suggested developing a common language that can bridge this gap in *Domain-Driven Design: Tackling Complexity in the Heart of Software* [10]. He called this language a *ubiquitous language*.

A ubiquitous language is a model-based language which is designed to describe the components of a model and the the rules that govern it. Furthermore, such a language allows non-technical domain experts and developers to communicate with each other comfortably and efficiently [30].

A fundamental characteristic of BDD is the structured format of behavioral specifications in the form of stories. This concept is directly inspired by the practices of agile software development and has many similarities with conventional user stories. In the case of BDD, a story is usually written in a domain specific or ubiquitous language which must be automatically executable by a software tool in order to verify the specification as if it was a test.

[4] http://jbehave.org (accessed 2013-10-05)

Even though BDD does not dictate the appearance and organization of a story, the scheme presented by Dan North [24] is now being widely used, and has been implemented mostly unchanged in a number of tools. Listing 2.1 shows the basic template for such a story which essentially comprises two major parts.

It begins with a narrative which explains **who** is the main stakeholder or character, **what** this person demands from the system, and the reason **why** or the benefit the persons hopes to gain from the proposed functionality. This composition forces the writer to consider the usefulness of a feature and whether the feature provides the appropriate benefit.

```
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 2.1: Structure of a story, adapted from *North* [24]

Secondly, the acceptance criteria describe specific cases, or examples, of the narrative as **scenarios**. A scenario begins with an initial condition, followed by a certain event, and concludes with an anticipated outcome. The initial state is defined by a **given** that explains the required context for the scenario. The **when** keyword marks the beginning of the event description that really illustrates the gist of the feature. After the event has occurred, the system is either left in an altered state or

in the same condition as before. The desired outcome is expressed after the **then** keyword.

2.3.3 Similarities with TDD

Behavior-driven development is in many ways a more thorough fulfillment of Kent Beck's original theory of test-driven development. For example, in his book *Test-driven Development: By Example* [7] Beck introduces the idea of manually updating a list of tests, crossing off those that have already passed and adding new ones as they come to mind. Most BDD tools actually automate this process through the notion of *pending* examples. This is one of usually three independent conditions for an individual test (the others being *passing* and *failing*) which indicates a not yet implemented piece of code.

Even though there are some idealogical and structural differences, many concepts apply to BDD as they do to TDD. In classical unit testing, a test case defines the *fixture* (test context) to run multiple tests. For example, a class containing multiple test methods would be called a test case. Such a class can also contain additional methods to initialize and clean up the fixture for each individual test run. For instance, the JUnit library for the Java programming language provides extensive facilities following this approach [9]. In BDD however, the fixture is commonly the executable source code that is directly connected to the ubiquitous language by the test framework. The notion of a test case is replaced by the story and a test is an example instead.

2.4 RSpec

In 2005 the behavior-driven test framework RSpec[5] for the Ruby programming language was created by Steven Baker after he had discussed elemental ideas with Dan North and other programmers. Although a comprehensive definition of

[5]http://rspec.info (accessed 2013-10-05)

BDD was still a work in progress at the time, RSpec's focus was on the behavior of software components already from the very beginning [8]. The framework does not employ an independent ubiquitous language, one that would have to be interpreted or compiled, but instead creates the structure of a story entirely with Ruby code.

Another difference to other BDD tools is that RSpec is most suitable for describing individual objects in a software rather than the system as a whole.

The BDD test tool is well respected in the Ruby development scene and is widely used in many software projects. Because of its versatility and behavior-driven approach, the RubySpec (Section 2.6) project also uses RSpec as their test framework of choice to create an executable specification and documentation of the Ruby programming language.

2.4.1 Structure and application

The Ruby programming language makes it relatively easy to create expressive syntactical constructs. Like other Ruby programs, e.g., the web framework Rails, RSpec also uses this circumstance to create a form of domain-specific language (DSL) by using poignant method names and *blocks* (closures).

```
describe [object] do           (example group, test case)
  describe|context [detail] do (code example, test method)
    it [behavior] do
      [expectation]            (expectation, assertion)
    end
  end
end
```

Listing 2.2: Structure of a RSpec *spec*

A set of RSpec test cases (also called *example groups*) that are contained within the same file is called a *spec*. Example groups can be nested hierarchically, but usually there are no more than two to four levels. The keywords *describe*, *context* and *it* all

invoke methods to create an instance of the class *RSpec::Core::ExampleGroup*. Their parameters are a string that describes the object, detail or behavior, and another example group or code block. The innermost example group in the hierarchy ultimately contains an executable block that contains the test code and assertions or *expectations* (Listing 2.2).

RSpec also provides a number of useful instruments that are reminiscent of traditional unit testing systems, like JUnit. There are *hooks*, methods that can run either before or after each single test case (*code example*) or all of them, and *expectations*, the equivalent of assertions. Just as the philosophy of BDD mandates, expectations have more colorful names than the functions of traditional TDD tools. The two methods *should* and *should_not* are also more powerful than simple assertions. Each method accepts a *matcher* or a special Ruby expression as an argument.

A matcher is an object that compares other objects using a special contract, not unlike the *Comparable* interface of Java. Matchers, however, not only perform comparisons between similar objects, but can also inspect the attributes and properties of an object. This makes expectations in RSpec very flexible and suitable for expressive statements in the code.

Even though RSpec's techniques are conceivably not much more than an elegant way of structuring regular tests, the purpose of this framework is rather to strongly encourage a different style of development. While it is certainly possible to misuse RSpec in a non-behavior-driven way, the usefulness of all kinds of software tests really depends on their correct utilization. This is also true of test-driven development.

2.5 Cucumber

Cucumber is a popular open source BDD test automation framework which follows the behavior-driven development approach.[6,7] It specializes in the specifica-

[6]http://cukes.info (accessed 2013-10-05)
[7]http://github.com/cucumber (accessed 2013-10-05)

tion of a software system as a whole, whereas RSpec focuses on individual objects inside the system. In this way, it is a more direct realization of the original BDD concepts.

Figure 2.3: Cucumber

Cucumber was originally written in the Ruby programming language by Aslak Hellesøy as a successor to the "story runner," an RSpec component developed by Dan North. By now there is also a fully featured version for the Java Virtual Machine (JVM), cucumber-jvm. Although a pure JavaScript version is currently in development as well, there is already support for modern web applications through a number of Ruby libraries. *The Cucumber Book: Behaviour-Driven Development for Testers and Developers* [14] by Matt Wynne and Aslak Hellesøy is a handbook on the correct use of Cucumber, and it also covers many BDD fundamentals.

Gherkin is the name of the ubiquitous language employed by Cucumber. It very closely follows the story structure that was initially outlined by Dan North (Section 2.3), with the minor difference that stories are called *Features* instead.

The parser for this language bears the same name and has been implemented with the Ragel state machine compiler.[8] Because Ragel supports many different programming languages, Gherkin is presently available for Ruby, Java, JavaScript and .NET.

2.5.1 Features, scenarios and steps

A story written in Gherkin has a very well defined but easily readable structure. A file should contain one single *feature* which can consist of one or more *scenarios*. There is only a small number of keywords that mark the beginning of a new element inside the structure (Listing 2.1).

Cucumber improves on the original story structure with the introduction of *tables* and *scenario outlines*, which are basically templates for scenarios. In this manner

[8] http://www.complang.org/ragel (accessed 2013-10-05)

```
1  Feature: Title of the feature or story
2
3   Description of the feature or narrative of the story.
4   This part can contain arbitrary text.
5
6   Background: Is executed once before every scenario
7     Given some condition
8       And one more thing
9       But something else
10
11  # This is a comment.
12
13  Scenario: A concrete example, illustrating the acceptance criteria
14    Given ...
15      And
16      But
17     When
18     Then
19
20  Scenario Outline: Template with placeholders, requires a table
21    Given I have <something>
22      And I also have <number> <thing>
23
24    Examples:
25      | something   | number | thing   |
26      | a monkey    | 3      | bananas |
27      | a rabbit    | 1      | carrot  |
```

Listing 2.1: Cucumber feature containing all possible elements

a lot of redundancy occurring from the duplication of acceptance criteria can be prevented.

By using variables inside steps, a scenario is executed once for every row in the given table. During each iteration the variables are substituted with their associated values in the respective column.

As an example, we can imagine a fictional website where a user can adopt the role of a *manager* or an *admin*. Depending on the role a user is being granted, he receives one of two different messages. Without tables, two separate scenarios would have been necessary to specify this behavior (Listing 2.2).

Another benefit of Cucumber is the concept of *backgrounds*. These are special sce-

```
Scenario Outline: Confirmation message
  Given I have a registered user account
   When an Admin grants me <Role> rights
   Then I should receive a confirmation message with the text:
    """
    You have been granted <Role> rights. <details>. Please be responsible.
    -The Admins
    """
  Examples:
    | Role     | details                                                  |
    | Manager  | You are now able to manage all accounts of your group    |
    | Admin    | Your are now able to manage any user account on the system |
```

Listing 2.2: Cucumber scenario outline using a table

narios that are executed once before every regular scenario. They are used to create the initial conditions required for every example, similar to the *before* method in JUnit. This is another way how redundancy and the amount of required text can be reduced (Listing 2.3).

```
Feature: Digital wallet

  Background:
    Given I am a registered user
      And I am logged in to my digital wallet

  Scenario: Withdrawing money from a wallet
    Given I have 10.00 dollars in my wallet
     When I click on the "withdraw money" button
     Then I should be redirected to the transactions page

  Scenario: Withdrawing money from an empty wallet
    Given I have 0.00 dollars in my wallet
     When I click on the "withdraw money" button
     Then I should see a dialog with the message:
       """
       Sorry, but your wallet is empty.
       """
```

Listing 2.3: Cucumber feature with a background and two scenarios

The hierarchical technology stack of Cucumber shows how the framework is integrated into a software project and how the different parts fit together. There are

two distinct segments, one of them on the business facing side and another one on the technology facing side (Figure 2.4).

Business facing means that those sections are used to specify the system and they are also utilized to communicate with non-technical personnel. The upper half is furthermore an allegory for the central BDD concept of how a project can be fully described by a well structured, executable specification. In the case of Cucumber the specification is the sum of the features, which consist of scenarios, which in turn are made up of individual steps.

The technology facing portion typically only concerns developers. The step definitions are directly imbedded into the code of the system. In other words, they are the "glue" that connects the system with the specification.

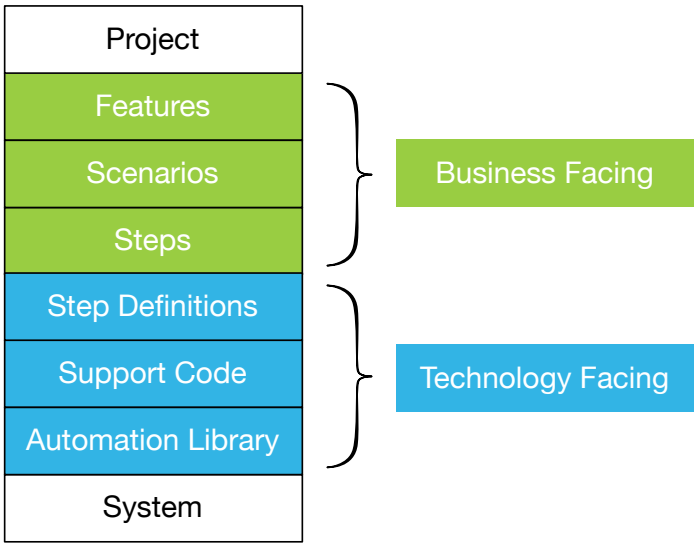

Figure 2.4: The Cucumber testing stack, adapted from *Hellesoy and Wynne* [14]

2.5.2 Step Definitions

In Cucumber, a step definition is the native code behind each step of a scenario. Step definitions are methods implemented in Ruby, Java or any other of the supported programming languages. Different implementation details notwithstand-

ing, step definition methods are always matched to their corresponding steps inside the feature files with the help of unique *regular expressions*. The regular expressions placed inside the code either as string arguments (Ruby, C++, etc.) or annotations (Java, Scala, etc.) of the methods.

As an example, Listing 2.4 shows the implemented step definitions for the feature of the fictional digital wallet from Listing 2.3 in Ruby code. In the Ruby variant of Cucumber, the keywords Given, When, Then, etc., are built-in methods of the framework that take a regex and a code block (closure) as their two arguments.

```
1  Given(/^I am a registered user$/) do
2    # Create a test user object.
3  end
4
5  And(/^I am logged in to my digital wallet$/) do
6    # Enter the credentials of the test user.
7  end
8
9  When(/^And I click on the "(\w+)" button$/) do |button|
10   # Perform a click on the UI element.
11 end
12
13 Then(/^I should see a dialog with the message: (\w+)$/) do |page|
14   # Check if we are on the correct page.
15 end
```

Listing 2.4: Cucumber step definitions for the wallet feature in Ruby

At runtime, the Cucumber framework parses the source code files that contain the step definitions and dynamically loads the methods within. When the tool then reads in the feature files it can execute the correct code corresponding to the steps of each scenario of every feature.

In practice the Cucumber step keywords are interchangeable because only the regular expression is evaluated. Nevertheless it is probably a very good idea to use the keywords appropriately in order to achieve better readability of the features and to properly organize the step definitions.

2.5.3 Cucumber and RSpec

Because Cucumber focuses on the behavior of a software system as a whole and RSpec is more suitable for specifying individual objects, the two frameworks can be very well used in concert with each other. The complete BDD cycle shows the approach of working "from the outside in" (Figure 2.5).

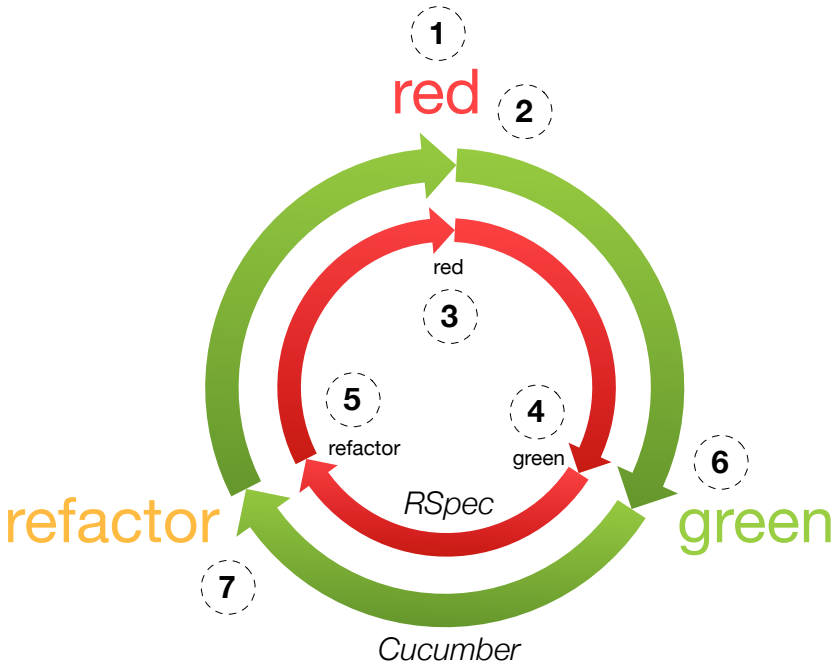

Figure 2.5: BDD cycle, adapted from *Chelimsky et al.* [8]. Working from the outside in, the following phases must be repeated for each scenario inside a feature:

1. Introduce a new scenario
2. Write a failing Cucumber step definition
3. Write a failing RSpec example
4. Test if the example passes
5. Refactor
6. Test if the scenario passes
7. Refactor

The development of a new feature starts out by creating a new *scenario* and the first exemplar in a set of still failing *step definitions* with Cucumber. Next, RSpec is used to write an *example* that, when fulfilled by the actual implementation in production code, should also satisfy the step definition. This process is repeated until the whole scenario is complete. After that, new scenarios can be added as needed before the feature is regarded as expressive enough.

2.5.4 Cucumber-JVM

The implementation of Cucumber for the *Java Virtual Machine* supports many programming languages and software packages that run on a standards-compliant JVM. Java, Groovy and Scala are probably the most popular examples. With the modules for JRuby and Jython, even Ruby and Python code can be tested with Cucumber on the JVM. Furthermore, support for Android applications was added only recently by Maximilian Fellner and the Cucumber team (Section 3.4). This makes it very comfortable for users to run Cucumber directly on their target devices, just like regular tests.

The JVM version of Cucumber works with exactly the same feature files, scenarios and steps as the Ruby implementation. The only perceptible difference is, of course, that step definitions are written in Java or any of the other supported languages. Regarding features and ease of use, there are no disadvantages to the original variant of Cucumber. Employing one of the many submodules, features can even be executed through JUnit, which makes it a lot easier to integrate them into an existing unit testing environment.

When using the Java variant of Cucumber, step definitions must be placed as public methods inside one or more classes. They also have to be annotated with one of the annotations @Given, @When, @Then, or @And. At runtime, Cucumber-JVM dynamically loads and executes the annotated methods using the reflection mechanism of Java. The regular expressions inside the annotations work the same as they do in Ruby; they can even be copied and reused from an existing project

in that language. One must be aware, however, that special regex symbols need to be escaped with two "\" characters in Java (Listing 2.5).

```
1 @Given("^I have (\\d+) slices of cucumber$")
2 public void I_have_slices_of_cucumber(int slices) {
3   // Do something with the slices
4 }
```

Listing 2.5: Cucumber step definition written in Java

The method names of the step definitions need to be unique in accordance with Java language requirements but can otherwise have any possible name. It is usually common practice to build a name from the text of the Cucumber step. In addition to step definitions, the so-called glue code classes can also contain two methods annotated with @Before or @After. These are *hook* methods that are used to prepare or clean up the fixture, the instance of the step class, before and after every scenario or example.

2.6 Ruby Spec

The popular Ruby programming language has been implemented a number of times in different versions. The reference implementation by inventor Yukihiro Matsumoto, Matz's Ruby Interpreter or **MRI**, is written in C.

Rubinius is a bytecode virtual machine written in C++ that uses LLVM to compile bytecode into machine code at runtime. In this implementation of Ruby, the bytecode compiler and most of the core classes are actually written in Ruby themselves.

JRuby is a virtual machine for the JVM written in Java. Ruby code can be interpreted directly and compiled into bytecode just-in-time or ahead-of-time. There are further implementations, including .NET and Smalltalk versions.

In order to provide a way to test and verify the correctness of a Ruby implementation, **RubySpec**[9] strives to create a complete, executable specification using RSpec. Running popular and complex Ruby software, e.g., the web framework Rails, has been a benchmark for Ruby interpreters and virtual machines, but this approach does not cover testing the full features of the programming language. The testing paradigm of RubySpec is somewhat similar to bootstrapping; a Ruby implementation must be at the very least able to even execute a single test, and from that point on the ability to pass simple tests becomes the precondition for the more complex ones.

```
1  require File.expand_path('../../../spec_helper', __FILE__)
2  require File.expand_path('../fixtures/classes', __FILE__)
3
4  describe "Kernel#==" do
5    it "returns true only if obj and other are the same object" do
6      o1 = mock('o1')
7      o2 = mock('o2')
8      (o1 == o1).should == true
9      (o2 == o2).should == true
10     (o1 == o2).should== false
11     (nil == nil).should == true
12     (o1 == nil).should== false
13     (nil == o2).should== false
14   end
15 end
```

Listing 2.6: Ruby spec for the "==" operator

The *spec files* which contain RSpec tests are organized in direct correlation with the official Ruby language documentation.[10]. Because Ruby is a strongly object-oriented language, most features of the programming language can be described in terms of objects and their methods. Listing 2.6 shows an example of a test for the equality operator "==" of Ruby. The complete executable specification is being actively developed as an open source project, but is still incomplete at this time. Section 4.3 will introduce a similar test-driven approach to programming language specification for the programming language ADA.

[9] http://rubyspec.org (accessed 2013-10-05)
[10] http://www.ruby-doc.org (accessed 2013-10-05)

2.7 Other BDD tools

2.7.1 Concordion

The acceptance test framework Concordion[11] is an open source tool that was at first implemented in Java, but is now also available for .NET, Python, Scala and Ruby.

Software requirements and specifications are written without any special structure as HTML documents in natural language. In order to connect the documents to executable source code, the markup needs to contain so-called "instrumentations" (Listing 2.3). These are special attributes that are placed inside the HTML tags and are invisible when the document is being viewed in a browser. A Java fixture class can process the instrumentations that accompany the specification, and connects the specification to the system under test.

```
<p>
When <span concordion:set="#name">Bob</span> logs in, a greeting
<span concordion:assertEquals="greetingFor(#name)">Hello Bob!</span>
should be displayed.
</p>
```

Listing 2.3: Example of an instrumenation, adapted from concordion.org

One advantage of Concordion is that the specification documents are typically more readable than when using other frameworks because they can be freely structured and visually enhanced using cascading style sheets (CSS).

HTML is, however, also more cumbersome to write and maintain than pure code or a specialized ubiquitous language. Furthermore, because the connection between the specification and the executable code is provided by attributes hidden in the markup, it is not immediately clear which are the essential parts and which parts are only for illustration purposes.

[11] http://www.concordion.org (accessed 2010-10-05)

2.7.2 FitNesse

FitNesse[12] is a collaborative wiki web server where the individual wiki pages are executable tests. This open source framework is a Java application that can be used on a local machine or as a service on a server. With an extensive list of plugins provided by users, FitNesse supports many of the major programming languages like Java, .NET, Ruby, Python, C++, etc..

Tests are expressed as tables of input data and expected output data. This table style is also called a "decision table," where each row represents a complete scenario (Listing 2.4).

```
|eg.Division|
|numerator|denominator|quotient?|
|10       |2          |5        |
|12.6     |3          |4.2      |
```

Listing 2.4: Example of a FitNesse wiki markup, adapted from fitnesse.org

Decision tables are executed live on the system under test when clicking the "test" button on a wiki page. The contents of the table are matched to a Java fixture class which the underlying system uses to connect with the system under test. The results of the execution are then also displayed on the same page. FitNesse provides a second class of table that uses a compact RPC (remote procedure call) system called *SLIM* to directly call functions in the system under test.

By restricting the test format to tables, FitNesse is somewhat more limited than comparable systems. But the integration of a fully featured wiki system makes the framework also increasingly flexible and very suitable for a more complete documentation that can be read and edited by many different people.

[12] http://fitnesse.org (accessed 2010-10-05)

2.7.3 JBehave

Initiated by Dan North, JBehave[13] is one of the earliest BDD test frameworks. It adopts the scenario-based story structure and the step keywords *Given, When, Then* that are also part of Cucumber (Section 2.5). JBehave is a pure Java implementation and also does not support any other programming languages, but it can be integrated quite easily into an existing workflow.

JBehave consists of a *core* and a *web* distribution, the latter one being an extension which provides support for web-related access and functionality. A separately available *Selenium integration module* allows the user to drive the verification of web application behavior using Selenium, an automation framework for web browsers.

2.7.4 Robot Framework

The open source, generic test automation tool Robot Framework[14] is intended for acceptance testing and acceptance test-driven development. The development of the core framework is supported by Nokia Siemens Networks.[15] Robot Framework supports a keyword-driven and data-driven testing approach with a tabular test data syntax. It is implemented in the Python programming language and can be extended natively with both Python and Java. Software in other languages can also be tested using an XML-based RPC interface.

The framework allows test cases to be formatted in a relatively permissive way as simple plain text or in HTML, focusing only on keywords. The supported file extensions are thus *.txt* and *.robot* for plain text files, *.tsv* for tab-separated files, *.html, .htm* and *.xhtml* for HTML files, and finally *.rst* or *.rest* for reStructuredText.

[13] http://jbehave.org (accessed 2010-10-05)
[14] http://robotframework.org (accessed 2010-10-05)
[15] http://nsn.com (accessed 2013-10-05)

As an example, Listing 2.7 shows a test for the login feature of a web service. The keywords used in this example are defined in a separate file, *resource.txt*, which can be found in Listing A.8.

```
*** Settings ***
Documentation     A test suite with a single test for valid login.
...
...               This test has a workflow that is created using keywords in
...               the imported resource file.
Resource          resource.txt

*** Test Cases ***
Valid Login
    Open Browser To Login Page
    Input Username    demo
    Input Password    mode
    Submit Credentials
    Welcome Page Should Be Open
    [Teardown]    Close Browser
```

Listing 2.7: Robot framework login feature, adapted from robotframework.org

Robot Framework produces structured reports for all test cases inside well readable HTML documents. An example of such a test report can be seen in a screenshot in Figure 2.6. Of course test cases do not have to be written as plain text files. Another approach supported by the framework is to use tables inside HTML documents. In this case no special markup is required, only the headings of the test data tables are considered. Four different kinds of table can be used with the following names required in their first column: *Setting*, *Variable*, *Test Case* and *Keyword*. The headings *Value*, *Action* and *Argument* which follow in subsequent columns specify additional information. Robot Framework recognizes only these words and ignores all other contents of an HTML document.

Keywords can actually consist of multiple words and any set of characters, and can even be combinations of other keywords. Robot Framework comes with a set of built-in keywords and can be effortlessly extended with custom words by the user. Eventually, the lowest level words have to be implemented in actual source code, called *test libraries*. Keywords can come from multiple user-provided

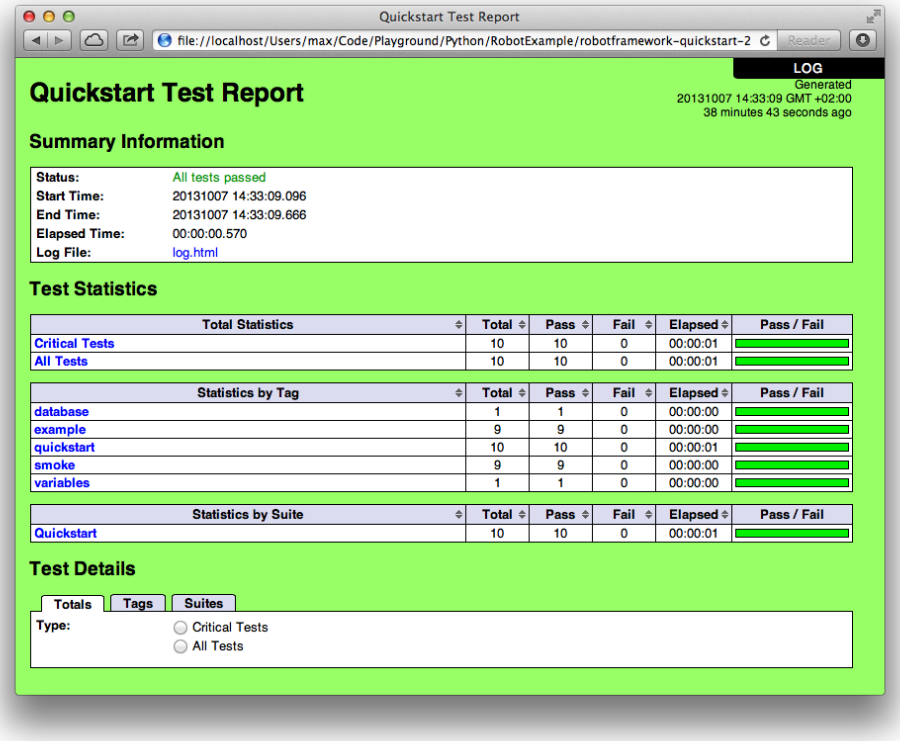

Figure 2.6: Robot framework example test report

or external libraries, a handful of which is already included together with Robot Framework.

Robot Framework is exceedingly versatile and suitable for testing every aspect of a software system ranging from user interfaces to command line programs. The keyword-driven approach makes the framework very flexible, but also more difficult to configure and maintain than other systems which provide a one-to-one binding between test case files (or stories) and the test code.

2.7.5 More BDD tools

SpecFlow[16] is an open source BDD test framework for software built with .NET. It uses the Gherkin parser and ubiquitous language of the Cucumber project (Section 2.5) but allows step definitions to be written in the C# programming language. The framework can be installed as a plugin for the Microsoft Visual Studio IDE, and supports syntax highlighting and the execution of feature files alongside an existing workflow with common tools.

TextTest[17] is a text-based functional testing framework for the Python programming language with a focus on user interface testing. The framework itself also includes a graphical user interface for the creation and execution of tests. Tests are written in a plain text format which can be automatically generated by another tool called *StoryText*. This tool can record actions performed by a user on the screen and translate into a simple domain specific language. The recordings can then be replayed as test cases by TextTest. A third tool, *CaptureMock*, applies this approach to mocking classes.

Twist[18] is a commercial product developed and distributed by the company ThoughtWorks, Inc. This test framework for the Java programming language comes with a custom IDE based on Eclipse. It uses a specialized domain specific language for test cases that supposedly allows for improved collaboration between engineers and non-technical stakeholders. The product and professional support are available from the company for an annual price of USD 99.00.

[16] http://www.specflow.org (accessed 2010-10-05)
[17] http://texttest.sourceforge.net (accessed 2010-10-05)
[18] http://www.thoughtworks.com/products/twist-agile-testing (accessed 2010-10-05)

2.7.6 Comparison of BDD tools

The following table gives an overview of the BDD or acceptance testing tools and frameworks introduced in the previous sections.

Name	Supported Languages	Test Focus	Specification format
RSpec	Ruby	single objects	Ruby code
Cucumber	Ruby, JVM languages, Javascript	whole system	ubiquitous language
Concordion	Java, .NET, Python, Ruby, Scala	whole system	HTML
FitNesse	Java, .NET, Python, Ruby, C/C++	whole system	wiki markup
JBehave	Java	whole system	ubiquitous language
Robot	Java, Python	acceptance tests	keywords
SpecFlow	.NET	whole system	ubiquitous language
TextTest	Python	acceptance tests	plain text
Twist	Java	acceptance tests	ubiquitous language

While every framework has its own particular strengths and weaknesses, their concrete sets of features and overall software architecture are the two most important properties when considering a cross-platform employment on modern mobile devices. In fact, only Cucumber and the Gherkin syntax have been modified and integrated many times over in a number of different tools and frameworks for the testing of mobile applications.

The list of software projects using Cucumber includes Calabash (Section 3.2), Frank (Section 3.3), iCuke,[19] and Zucchini.[20]

[19] http://github.com/unboxed/icuke (accessed 2010-10-05)
[20] http://www.zucchiniframework.org (accessed 2010-10-05)

3 Testing mobile applications

3.1 Challenges and motivation

The increasing importance of mobile computing platforms over traditional personal computers is undeniable. On the one hand, computers like smartphones and tablets, but also laptops and desktop computers, are increasingly being used for interaction with the World Wide Web. In a sense, the Web has become its very own platform for which programmers can develop software using the same tools regardless of the hardware a consumer chooses to use. On the other hand, these mobile and wirelessly connected devices are becoming more and more capable of fulfilling the same purpose for consumers as their larger precursors. For many people a smartphone and a tablet can be the only computers they require in order to fulfill their everyday needs. Certainly these mobile devices are nowadays much more likely to become the very first general purpose computers in children's lives, because they are relatively cheap and very easy to use.

Software for smartphones, tablets and the like can of course be similarly complex as software for desktop computers. While Microsoft Windows, Apple OS X, and some distributions of Linux to a lesser degree, have established themselves as de facto standards, and also support overlapping technologies, mobile operating systems are more diverse. Java, Objective-C and C# are the programming languages used for user space software in the major mobile operating systems Android, iOS and Windows Phone, respectively. The system kernels and application programming interfaces (APIS) are also vastly different on every platform. For this reason, cross-platform development poses many challenges for programmers. Another issue that complicates mobile application development is that smartphones

and tablets almost exclusively use ARM-based and not desktop-class x86 CPUs. As a result, applications eventually need to be transferred onto a physical device or an emulator, although cross compilers exist. In consequence, software for mobile devices cannot usually be tested on the same computer a programmer uses for development. Furthermore, many useful tools and test frameworks may not be available to a developer because the required technologies do not exist on a smartphone or tablet operating system. At the same time, the nature of mobile applications asks for completely novel testing paradigms altogether. Mobile operating systems feature complex visual patterns of touch-based user interaction, and the devices they run on contain many different sensors, the data of which is often incorporated into use cases of applications.

But for just these reasons, behavior-driven development can potentially be very useful for testing and specifying software on mobile devices. Firstly, the holistic view of specification by example makes it easier to translate requirements of visual and touch-driven user experiences into actual working code. And secondly, the concept of an independent ubiquitous language lends itself well to the cross-platform demands of mobile application development. The following sections will review several behavior-driven tools and frameworks for this purpose.

3.2 Calabash

The Danish company Lesspainful Apps[1] provided a cross-platform user interface test automation service for Android and iOS applications. The company was acquired by the US software startup Xamarin in 2013, and the original services are now provided as part of the new product *Xamarin Test Cloud*.[2]

A UI test framework based on Cucumber (2.5) with the name *Calabash* continues to be actively developed as an open source project on the *social coding* website Github.[3] Calabash is available for both the iOS and Android mobile operating

[1] http://www.lesspainful.com (accessed 2013-09-22)
[2] http://xamarin.com/test-cloud (accessed 2013-10-05)
[3] http://github.com/calabash (accessed 2013-10-05)

systems and consists of two major components: a Ruby library (i.e. *gem*) for the local development machine, and a server for the remote mobile device or emulator. The client and server communicate using a JSON based protocol to trigger and parametrize certain predefined actions (Figure 3.1).

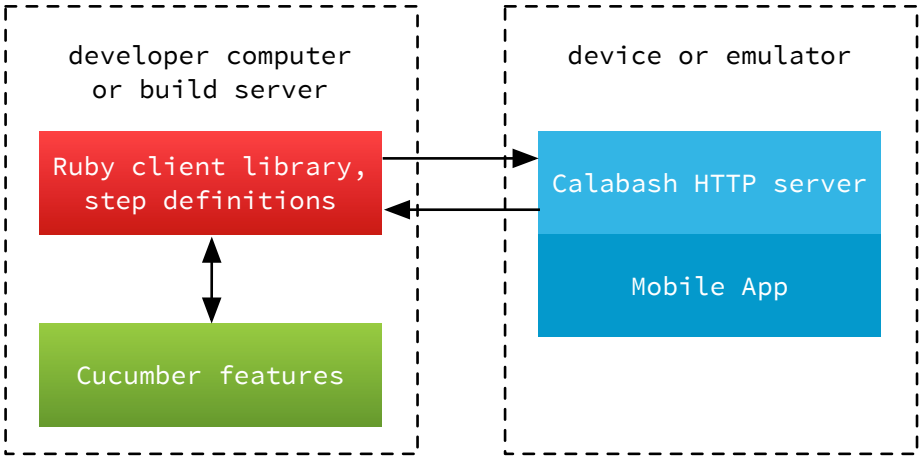

Figure 3.1: Calabash system architecture

Using this framework, step definitions must be written in Ruby and be placed on the developer's computer or on the build server together with the feature files. Calabash provides a number of built-in operations for extensive testing of the user interface (UI) and interaction design (UX) of mobile applications across iOS and Android. Amongst others, these steps include operations to press buttons, scroll lists, swipe across the screen and assertions for displayed text. Because the provided operations are relatively abstract and equally available on both operating systems, the same Cucumber features can be used to test two implementations of the same mobile application.

However, because the actual native code behind the steps is implemented in the server component of Calabash, no custom steps for directly interacting with the code of the mobile application can be implemented by the user. Because of the open source nature of the framework it would in theory be possible to add such modifications in a custom fork of the source repository. But the added complexity makes this approach hardly practical in most cases.

3.3 Testing with Frank

The open source project *Frank* is another user interface test framework that supports the development with Cucumber exclusively on iOS.[4] Similar to Calabash (Section 3.2), a number of predefined, "canned" operations can be rearranged and combined to create customized step definitions. Some ready-made steps for interaction and verification of screen contents are also already included.

Frank also uses a client-server architecture with the server component being compiled into the mobile application being tested. Likewise, the framework also features a JSON based wire protocol called *Frankly* for bidirectional communication. Testing on a physical device is possible, but requires additional software and a special configuration process. Additionally, Frank even can be used for desktop applications running under Apple OS X.

3.4 Cucumber Android

After originally being written in Ruby, the BDD test framework Cucumber has also been implemented in Java for the JVM (Section 2.5) and is thus compatible with a wide range of systems and programming languages. The open source operating system Android for smartphones, tablets and other platforms does not use a *Java Virtual Machine* but a custom implementation with the name *Dalvik*[5] which was optimized for use on mobile devices. Most parts of the API are identical to Java SE 6 (to such an extent that it warranted a lawsuit between the companies Oracle and Google [35]) but nevertheless there are several subtle differences.

Most importantly, dynamic class loading and Java's reflection mechanism work differently on Android,[6] which is critical for Cucumber. With the *Cucumber-Android*

[4]http://www.testingwithfrank.com (accessed 2013-10-05)
[5]http://code.google.com/p/dalvik (accessed 2013-10-05)
[6]http://android-developers.blogspot.com/2011/07/custom-class-loading-in-dalvik.html

module[7] Cucumber can now be integrated natively with the Android test framework[8] for the direct execution of step definitions on the Dalvik VM whilst supporting all the available features.

3.4.1 Android fundamentals

Android is an open source operating system developed by Google. It uses a monolithic, modified Linux kernel and includes a Java application framework based on Apache Harmony.[9] Using the freely available Android SDK, user space applications are packaged into Android packages (*.apk*) using a two-step compilation process. Java Code is at first compiled into an intermediate bytecode format which is then translated into bytecode in the Dalvik Executable-Format (*.dex*) for the Dalvik VM.[10] Apk files typically contain both uncompiled resources (e.g., image and audio data) and compiled resources [16].

Android Applications can enclose components of up to four different types [15]:

- **Activities** are processes which are in possession of a user interface for a screen. Only a single Activity can be in the foreground for the user to interact with at any given time. Activities are intricate elements of an application's overall lifecycle.
- **Services** are components that run in the background and do not provide a user interface. They are started by any of the other components and multiple services can be active simultaneously.
- **Content providers** have the capability to manage and provide sets of shared application data. They are created automatically by the system.
- **Broadcast receivers** respond to system-wide broadcast announcements and are only active during the short period when the announcement is being made. They are typically used to start specific Activities.

[7] http://github.com/cucumber/cucumber-jvm/tree/master/android (accessed 2013-10-05)
[8] http://developer.android.com/tools/testing/testing_android.html (accessed 2013-10-05)
[9] http://harmony.apache.org (accessed 2013-10-05)
[10] http://source.android.com/devices/tech/dalvik (accessed 2013-10-05)

When testing an Android application, the test classes and associated resources are usually contained inside their own separate test project, which will be compiled into an independent .apk file. Both the application being tested and the test package are placed on an physical device or an emulator. The *InstrumentationTestRunner* is a class provided by the Android framework which handles the dynamic loading and execution of test code, and is a subclass of *Instrumentation*. An Instrumentation is a set of control functions of the system that control Android components independently from their normal lifecycle (Figure 3.2). When running tests, the class will be instantiated before any of the application code. An Instrumentation implementation is described to the Android test system through an *instrumentation* tag inside the file *AndroidManifest.xml* of the test project. The manifest contains essential information about the application that is required by the Android system. The system must have all of this information before it can run any code of the application.

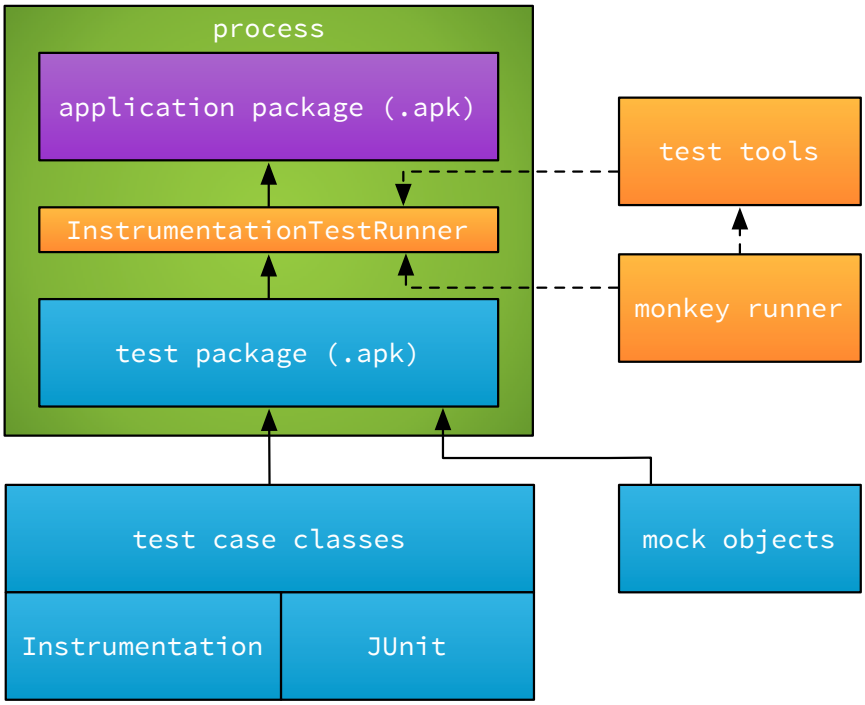

Figure 3.2: Overview of the Android test framework

3.4.2 Testing with Cucumber JVM

By replacing the *InstrumentationTestRunner* with another implementation, the test system can be altered and foreign code can be integrated. Cucumber-Android implements a custom variant of an Instrumentation in order to execute feature files with Cucumber JVM instead of regular unit tests. A simplified class diagram of Cucumber-Android can be seen in Figure 3.3.

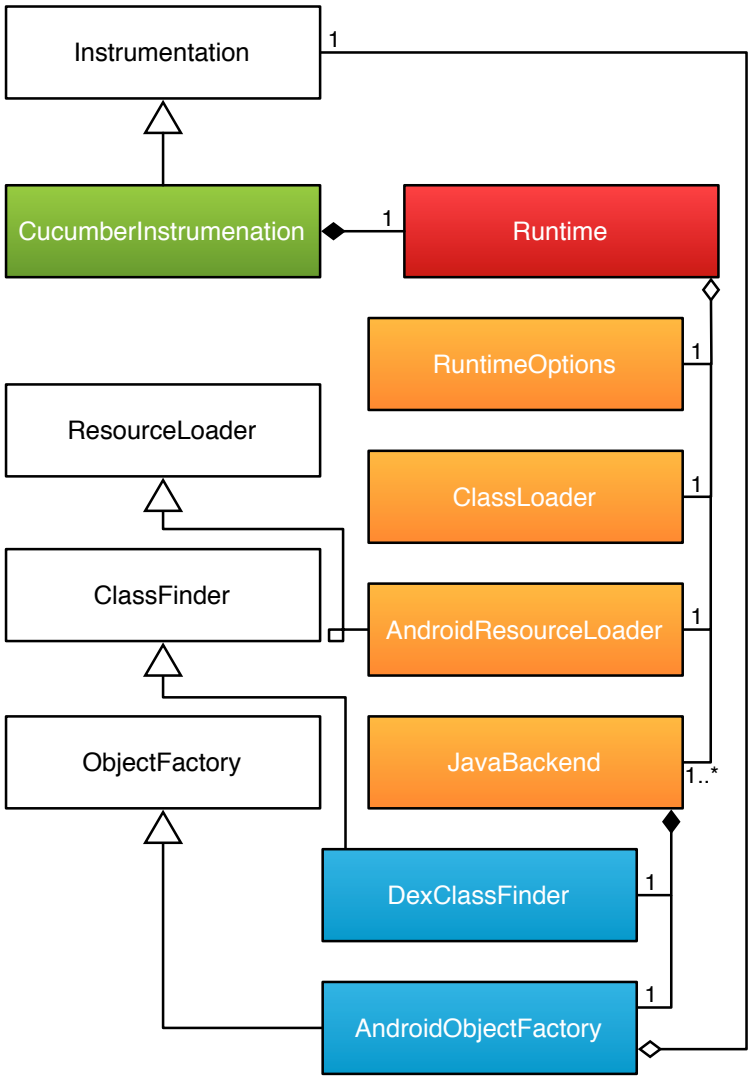

Figure 3.3: Class diagram of the Cucumber-Android module

At its core, the module includes the class *CucumberInstrumentation*. This is the only component that is being used to communicate between the Android test framework and Cucumber JVM. When creating a new test project, this class has to be utilized instead of the regular InstrumentationTestRunner. Inside the AndroidManifest.xml of the test project, the *instrumentation* tag needs to be configured in the following way:

```
<instrumentation android:name="cucumber.api.android.CucumberInstrumentation"/>
```

An Android test project for Cucumber JVM requires the libraries *cucumber-android*, *cucumber-core* and *cucumber-java*, all of which are available from Maven. Because Maven is not the recommended build environment for Android, it is advisable to download the required .jar files manually and place them inside the *libs* directory of the project, which is also the appropriate place for any additional libraries.

The class *AndroidResourceLoader* is responsible for loading the Cucumber feature files at runtime. For the loader to be able to find the files, they must be placed inside a directory called *features* inside the *assets* directory of the Android test project. Feature files are included as raw assets inside the test package and can thus be opened using the asset management mechanism of Android. Almost every feature of Cucumber JVM is also supported on Android. When the features are being executed, debug output is provided through the logging mechanism *Logcat*, and status code is reported back to the Instrumentation. This way, an IDE (e.g., Eclipse) can display successful and failed features as if they were unit tests and provide valuable feedback to the user.

Cucumber is capable of generating structured HTML reports which give an immediate overview of the failed and passed feature tests (Figure 3.4). However, because Cucumber Android is still in beta, the access to formatted test results is still a little bit cumbersome. At this time a developer has to copy the generated reports manually from the remote device using a command line tool included with the Android software development kit.

Future improvements to the code base of Cucumber will likely include more ad-

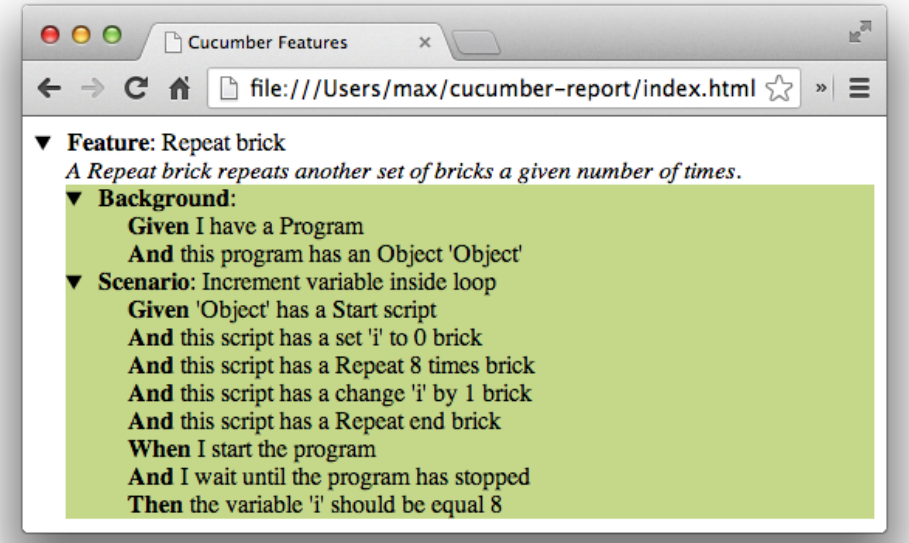

Figure 3.4: Cucumber example report for a successful feature test

vanced and comfortable reporting features, for example integration with a local service that can give real-time feedback about every test run. The source code and binary packages of the Cucumber JVM project, including the Android module, complete with instructions, tutorials and examples, are available from the project's website.[11]

3.5 Cucumber on other platforms

Cucumber has already been implemented in Ruby, for most programming languages running on the JVM and Javascript. Although C++ step definitions are supported, they still require Cucumber Ruby to run on a desktop computer. Calabash, Frank and similar testing frameworks on the other hand do not provide enough functionality, and are impractical for in-depth testing because of the

[11] http://cukes.info/install-cucumber-jvm.html (accessed 2013-10-05)

client-server-model. In order to support the other two major platforms, iOS and Windows Phone, a native implementation of Gherkin and Cucumber using the programming languages C and C++ is the only viable solution. To achieve such an objective, essentially two components would have to be considered:

Gherkin interpreter. The ubiquitous language of Cucumber is a relatively simple domain specific language, which encompasses only a handful of keywords and tokens. The story structure of Cucumber feature documents is also quite uncomplicated. As a result, the lexer and parser for this language have a very modest size of approximately 2800 source lines of code (SLOC) in Ruby. Furthermore, the Gherkin interpreter has been accomplished with the Ragel state machine compiler [31], and consequently it requires not very much effort to adapt and compile it for other programming languages. However, native support for C or C++ has not yet been implemented and would be the first step toward a functioning version of Cucumber on other mobile platforms.

Cucumber core modules. The core modules of Cucumber are responsible for generating an abstract syntax tree (AST) with the help of Gherkin. The objects of the AST connect the steps of each feature with their associated step definitions. Additionally, Cucumber contains a number of components which allow the user to generate formatted report documents from a test run. Altogether, the number of source lines of code of Cucumber's Java implementation amounts to approximately 6400. These modules would have to reimplemented from the ground up in C or preferably C++, because of its object oriented nature. The Ruby variant of Cucumber does in fact support step definitions that are written in C++ for testing C or C++ programs.[12] However, this extension requires a working Ruby installation and an additional C++ test framework, *Boost Test* or *GTest*. There are also macros for annotating step definitions which could probably be reused in a native C++ implementation of Cucumber.

[12]https://github.com/cucumber/cucumber-cpp (accessed 2013-10-05)

Cucumber has a distinct disadvantage in not being truly cross-platform yet. This certainly has to do with the fact that mobile applications have not been a concern of the open source project until very recently. Another factor which influenced the implementation languages of the test framework was likely the focus on web application development, which usually does not involve C or C++. The architecture of Cucumber is, however, relatively well designed and the amount of existing source code is not overwhelming. For this reason a reimplemenation in C++ should be an achievable goal, given that enough developers are actually interested in realizing this vision. The increased maintenance efforts of keeping a third variant of Cucumber up to date and feature complete would certainly be an additional burden for the whole project, but thriving open source communities can usually deal with such added complexity.

The challenges for a native HTML-5 version are similar. Fortunately, work in this regard has already progressed a little bit further. Besides, several extensions to Cucumber already exist which allow the testing of Javascript applications on the desktop.

4 Programming language specifications

4.1 Elements of programming languages

A programming language is an abstraction used by humans to efficiently convey instructions to a machine, specifically a computer. Programming languages are often built upon many levels of abstraction, as some languages themselves have been implemented in other programming languages, like C. When talking about formal programming languages, one must consider both the syntax (a set of rules regarding the possible combinations of symbols) and the semantics (the meaning and effects of statements) of a language. With the help of the previous two components, a programming language typically also encompasses a set of types (e.g., numbers, strings), and algorithms which can act upon those types.

A programming language syntax is commonly specified by means of a formal, *context-free grammar* (CFG) which defines how tokens, or lexemes, can be meaningfully combined. Individual tokens on the other hand can be very well described by *regular expressions*. Two accepted notation techniques for context-free grammars are the *Backus–Naur Form* and the *van Wijngaarden form*. When translating a computer program into machine instructions, a scanner and a parser must first perform a lexical and syntactical analysis of the program text, respectively. After that, a semantic analyzer examines whether the program conforms to the specified set of semantic rules. Semantics define the *meaning* of a programming language; this portion cannot be described by a context-free grammar [27]. In practice, semantics are specified either through the medium of mathematical formulae, natural language, reference implementations, or test suites.

4.1.1 Visual programming languages

Visual programming languages provide yet another level of abstraction to machine instructions. The syntax of such a language typically consists of graphical elements, or diagrams, in a two-dimensional space. The arrangement of these elements and their spatial relationships constitute the meaning of a visual program. Specialized formalisms for the syntactical and semantical specification of visual languages have been considered in the past [21]. However, the semantics of a visual language can be specified just in the same manner as a text-based language. It is possible to use the same approaches and tools, with the possible exception of a formal mathematical description. There should be no significant reason why natural language, reference implementations, or test suites would not be suitable for the semantical specification of a visual programming language.

4.2 Programming language standardization

In 1964, Alt [3] identified the general degree of complexity, the lack of pure manpower, and the lack of travel funds for participation in meetings as the three major obstacles preventing the widespread adoption of specification practices. At least the last two concerns have been all but wiped away by the advent of the Internet and the associated increase in worldwide communication. Although the challenge of specification has in the face of new developments not become any less complex, almost every major programming language these days was eventually specified in a written document that is now published on the World Wide Web.

A popular example would be the programming language **JavaScript**. This language was originally made available in the Netscape web browser. In 1996 Netscape submitted JavaScript to Ecma International, an international, membership-based, non-profit standards organization for IT systems based in Geneva, Switzerland. Today, ECMA-262[1] is the most widely used variant of the language. The specifica-

[1] http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf (acc. 2013-10)

tion document contains more than 200 pages with details on data types, built-in functions, syntax, and grammar.

But it is not always a consortium or organization that backs a standard. Although the original developer of **Java**, Sun Microsystems, attempted a standardization at ISO, and later Ecma International, the language remains only a de facto standard, controlled by the *Java Community Process*.[2]

For **C++** there is an official ISO/IEC standard, the latest one being 14882:2011.[3] Unlike Ecma, the ISO document is not freely available, but rather has to be purchased for the price of CHF 238,00. While this should not present a barrier to established compiler vendors, it still makes the standard barely accessible to the general public. Instead, hobbyists and other private developers have to look for unofficial alternatives.

Even though standardization has become common practice nowadays, it is not clear if the benefits have been fully realized yet. While the widespread availability of specification documents is one remaining issue, the format of natural language is becoming unwieldy in the face of the growing complexity and the richness of features of modern programming languages. It seems somewhat absurd that despite increasing automation and abstraction in the field of computing, the definition of computer languages themselves are still being maintained in archaic ways. Evidently, more modern approaches with the support of advanced software tools should be considered to specify programming languages and to automatically test implementations for their compliance with given standards.

[2] http://www.jcp.org/en/home/index (accessed 2013-10-05)
[3] http://www.iso.org/iso/catalogue_detail.htm?csnumber=50372 (accessed 2013-10-05)

4.3 Ada Conformity Assessment Test Suite

The Ada Conformity Assessment Test Suite (ACATS)[4] is an official and freely available test suite for checking the conformity of Ada processors with the Ada programming language standard. The most recent version available is ACATS 3.0 for the 2007 version of Ada. Also still available is version 2.6 for Ada 95. ACATS essentially constitutes a specification of the Ada programming language semantics by means of a test suite.

Ada is quite unique in that it has become the first programming language to provide a method for certifying the conformity of compilers according to an international standard. The International Organization of Standards specifies this approach to certification in the document "Ada: Conformity assessment of a language processor" (ISO/IEC-18009:1999) [4]. Perhaps it is not surprising that Ada is standardized in this way because it was originally developed by the United States Department of Defense, and in its beginning was targeted at *embedded systems*. Because of its safety-critical support features, the programming language today is widely used in critical systems like aerospace vehicles [33] [34].

ACATS 3.0 comprises over 4660 files and 3771 individual tests. It organizes the tests into six different classes; A, B, C, D, E and L, each of which reflects different testing requirements of language conformity testing. The test files themselves have alphanumeric names corresponding to their class and the location of the test objective inside the AIG (if they are legacy tests) or a section inside the Ada Standard. The ACVC Implementer's Guide (AIG) is an older document describing the test objectives used to produce test programs for ACVC (ACATS) versions [5].

Listing 4.1 shows an authentic ACATS test used to specify a property of the *String* type. It belongs to class C of tests, which check that executable constructs are implemented correctly and produce expected results.

Tonndorf [32] argues that the conformity assessments of Ada can be a model for other programming languages as well. First of all, the basic requirement besides

[4]http://www.ada-auth.org/acats.html (accessed 2013-10-05)

```
1  -- C26008A.ADA
2
3  -- CHECK THAT UPPER AND LOWER CASE LETTERS ARE DISTINCT WITHIN STRING
4  -- LITERALS.
5
6  -- JRK 12/12/79
7  -- PWN 11/30/94 SUBTYPE QUALIFIED LITERALS FOR ADA 9X.
8
9  WITH REPORT;
10 PROCEDURE C26008A IS
11
12         USE REPORT;
13
14 BEGIN
15         TEST ("C26008A", "UPPER/LOWER CASE ARE DISTINCT IN STRING " &
16               "LITERALS");
17
18         IF CHARACTER'('a') = 'A'  THEN
19                 FAILED ("LOWER CASE NOT DISTINCT FROM UPPER IN " &
20                         "CHARACTER LITERALS");
21         END IF;
22
23         IF STRING'("abcde") = "ABCDE" THEN
24                 FAILED ("LOWER CASE NOT DISTINCT FROM UPPER IN " &
25                         "STRING LITERALS");
26         END IF;
27
28         RESULT;
29 END C26008A;
```

Listing 4.1: Ada conformity test C26008A regarding String types

an international standard for a programming language would be to have a single worldwide certification system. Just like in the case of ACATS, such a system would comprise the following components:

1. Testing object, an identified language processor (e.g., a compiler).
2. Test objective, to verify compliance of the object with an ISO standard.
3. Tests conducted through a testing method (e.g., a test suite, ACATS).

In the case of Ada, the testing method includes both positive test cases and negative test cases, which are intended violations of the standard that have to be detected by an implementation. According to Tonndorf, the condition to have only a

single generally accepted test suite is paramount. This way, multiple implementations can be reliably compared. The testing itself is also supposed to be conducted exclusively by recognized or accredited organizations.

The author contends that test suites for other programming languages do not have the same objectivity, impartiality and completeness as ACATS because they are primarily developed and maintained by compiler vendors for their own purposes. However, community-driven open source projects like RubySpec (Section 2.6) show that a complete and systematic approach to conformity testing without bureaucracy is potentially feasible.

4.4 Vienna Development Method

In the 1970's a local branch of IBM, the Vienna Laboratory, developed a set of tools and techniques grounded in formal specification methods. For decades, the Vienna Development Method (VDM) has been one of the most established formal methods to develop software systems.

VDM and and the VDM Specification Language (VDM-SL), are supported by a wide range of academic and commercial tools. Over the years, practical use of VDM and the academical research centered around it have contributed notably to the development of fundamental systems like compilers, and to advances in computer science in general. In practice, VDM is mostly used for the development of large scale industrial systems and for security and safety critical software.

When designing a software system with this method, one begins by constructing an object-oriented model of data types and classes. A VDM specification is a mathematical model where data types represent the classes of input and output values. Functions and operations which work on values of these types model the functionality of a system. A document containing such information is written in the abstract syntax of the model-oriented specification language VDM-SL [13].

An example of VDML-SL can be seen in Listing 4.1. Here a message type is modeled as a sequence of characters. An invariant defines a maximum limit of 10,000 characters for a message. Additionally, a function *substring* on messages is specified. The function shall return a boolean value *true* if one message is a submessage of another [20]. Because VDM-SL allows the detailed definition of data types and functions regardless of implementation details, it supports a high level of abstraction.

```
Message = seq of char
inv m == len m <= 10000

substring: Message * Message -> bool
substring(s1,s2) == exists i,j in set inds s2 & s1 = s2(i,...,j)
```

Listing 4.1: Example of VDM-SL, adapted from *Larsen, Fitzgerald, and Brookes* [20]

In order to create actual source code for a piece of software specified with VDM-SL, two steps are necessary. First, data types and functions have to be refined into concrete types and algorithms using an algebraic and thus provable process. Secondly, this concrete specification must be translated into the source code of the targeted programming language. The correctness of this final step cannot be proven because it depends on the semantic of the programming language. It would theoretically be possible to automate this process at least partially.

4.4.1 Software tools

Overture IDE is a popular open-source tool based on Eclipse which is used for software development with the VDM Specification Language.[5] Alternate versions of the tool also support the object-oriented extension VDM++, and VDM-RT, which is used for specifying real-time and distributed systems. The community surrounding Overture aims to develop a common open-source platform that integrates a variety of tools to construct and analyze formal system models with the

[5] http://www.overturetool.org (accessed 2013-10-05)

help of VDM. The IDE includes a number of advanced features which facilitate the development process, like automatic static type checking and so-called *proof obligations* [19].

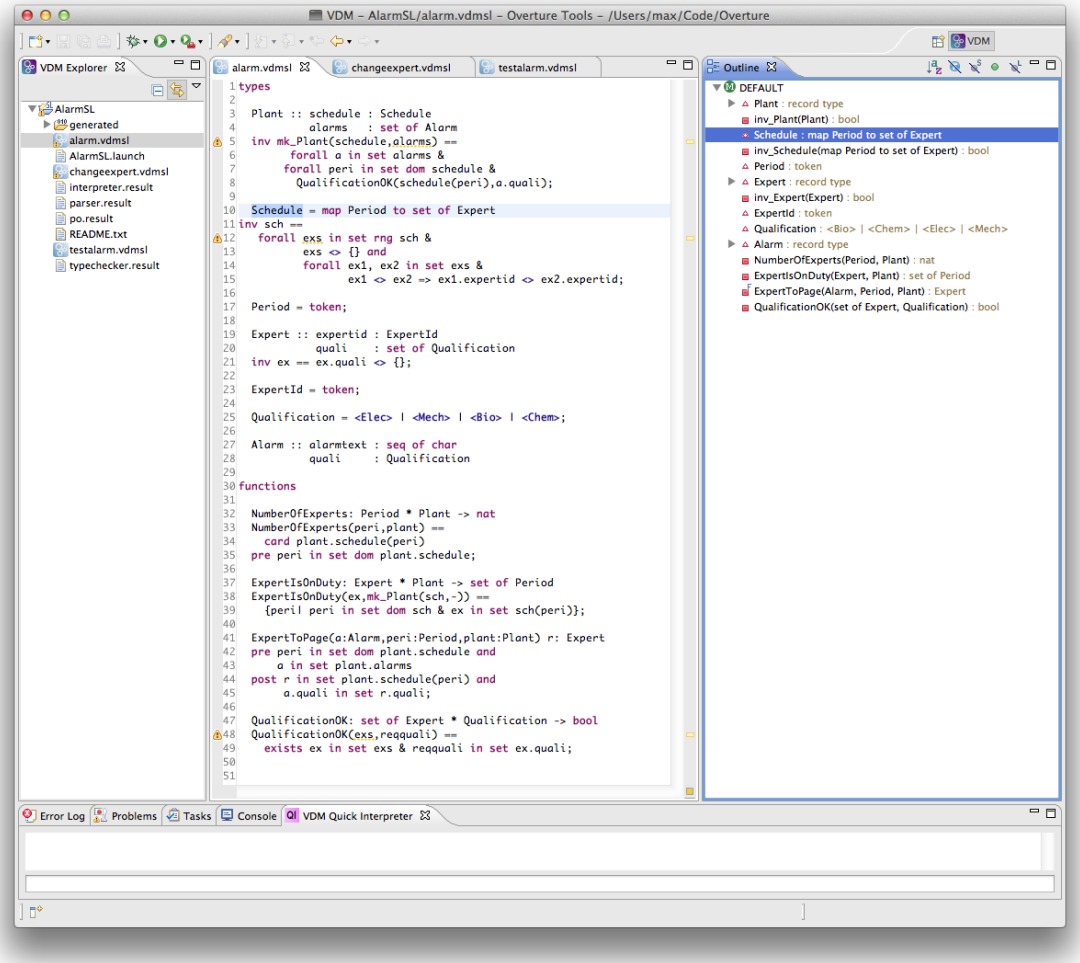

Figure 4.1: AlarmSL example project in the Overture IDE

Figure 4.1 shows an example that was inspired by an alarm system subcomponent developed by the Danish company IFAD A/S. Fitzgerald and Larsen originally introduced the example in their book *Modelling Systems: Practical Tools and Techniques in Software Development* [11]. The specified system is supposed to alert experts with different qualifications to operational faults in a chemical plant. The VDM-SL model of this system includes the necessary data types and functions, and

implements a number of requirements. Overture only allows for the creation and testing of formal models but it cannot automatically generate executable source code from these models. An automatic code generator has been proposed by Peter Jørgensen.[6]

VDMTools is a leading commercial tool for VDM and VDM++ that is being actively developed by the Japanese IT service company SCSK.[7] The software includes advanced features like a static checker and a code generator for both C++ and Java. Recently, an experimental reverse engineering module for Java was included in the tool suite to generate low-level VDM++ models from source code [12].

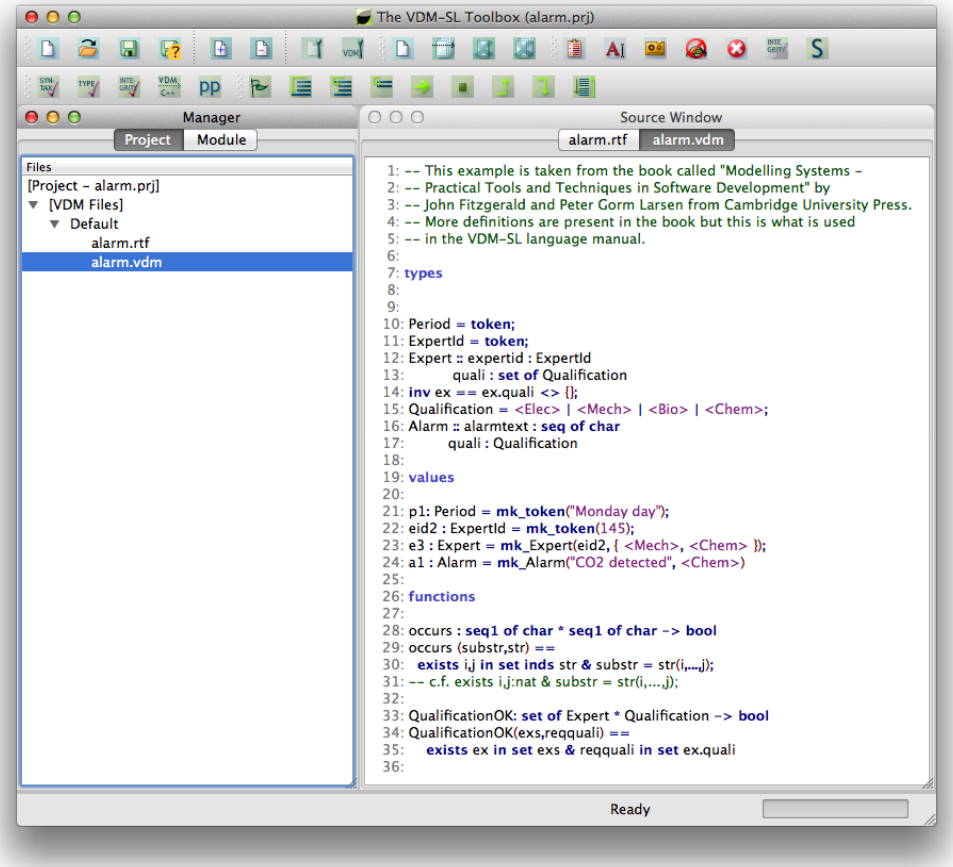

Figure 4.2: Alarm example project in VDMTools

[6]http://wiki.overturetool.org/images/1/1c/PeterJørgensen_CodeGen.pdf (accessed 2013-10-05)
[7]http://www.vdmtools.jp/en/index.php (accessed 2013-10-05)

VDMTools has been used in several large and important projects in the past. For example, the company FeliCa Networks developed the operating system for a new generation of integrated circuit chip with the help of VDM++ and VDMTools. The system was implemented in the programming languages C and C++ and after a VDM++ model. A specification document which spanned 677 pages and contained test cases formulated in VDM was created by a separate team of developers who were responsible for validation and verification. The VDMTools interpreter was used to execute these test cases, and it was also possible to generate information about test coverage. Using VDM++ and VDMTools helped to identify many issues in the requirements early on, and contributed to the success of the whole project [18].

5 Specification of a visual language with BDD

5.1 Specifying programming languages by example

The previous chapter gave an overview of the fundamentals of programming language standardization. It also explored the concepts of assessing compliance with a specification by means of automated tests and ways to formally specify software from the ground up. The focus of specification by example, to build the right software, is by its own definition aligned with the goal of written language standardization documents, to ensure consistent implementations. For this reason, the following sections will explore the suitability of BDD tools to take on the responsibility of replacing or at least supplementing such documents.

As it was already demonstrated by RubySpec in Section 2.6, a completely behavior-driven specification by means of acceptance tests is potentially feasible. But for the job of specifying software, and programming language implementations in particular, across multiple different platforms, an intermediate format may be more practical. Hence the ubiquitous language Gherkin of the Cucumber framework (Section 2.5) would be ideal for this task of cross-platform specification.

There are several conceivable advantages to this approach:

- **Natural language**. A ubiquitous language is close enough to the natural language of conventional specification documents that it can possibly constitute

all of the documentation. At the same time, executable specifications written in this format are able to fulfill at the very least the same purpose as traditional tests (e.g., ACATS).

- **Portability**. Contrary to test code, an intermediate format is portable across multiple platforms as long as respective interpreters for this format exist. In the case of Gherkin, the native code of step definitions still has to be ported, but it is easier to translate individual steps than whole tests.
- **Process**. Because SbE is a set of process patterns that are the foundation of an agile development process, the development of the programming language itself becomes agile when using this method for specification purposes.
- **Psychology**. Programmers may shy away from writing a formal specification when they come up with a new feature in their minds because they would rather start writing code immediately. On the other hand they might also be hesitant to write any code before the feature has not been extensively documented. Specification by example breaks down the psychological barriers between those two worlds by using practical and executable examples that can be easily composed.

5.1.1 Applying the story structure

In agile development, the purpose of a story is generally to capture the requirements for a single software component or to define a concrete use case from the perspective of an end user [6]. Behavior-driven development uses the story format for an outside-in approach to software development by defining the expected behavior of the whole system or individual objects.

Cucumber uses an extended variation of Dan North's original story structure (Section 2.3.2) written in the ubiquitous language of Gherkin. When specifying a programming language, each Cucumber feature should correspond to a single property of an element in the programming language. For example, a string type has many properties, ranging from the characteristics of the underlying raw

data, to the operations that can be performed on the string. The elements of programming languages were briefly introduced in Section 4.1. Depending on the complexity and the abstraction level of the language, the features will have to be more or less detailed. A decent guideline for the degree of detail are existing test suites like RubySpec (Section 2.6) or ACATS (Section 4.3).

There are plausibly two different ways how a programming language can be described using an exemplary Cucumber story:

One way is to define *program examples* complete with inputs and the expected output or behavior of the whole system. The program structure and expected behavior must be described using the Gherkin grammar of acceptance criteria (Given, When, Then). Input and output data can additionally be represented as formatted strings or as strings within tables. This approach is more suitable for specifying control structures and algorithms.

The second concept is to directly *describe* singular elements of the language (e.g., objects, primitive types) and their expected behavior under certain conditions. This technique is best used for specifying the type system. However, in higher level programming languages often most elements, even functions, are actually objects. In these cases this descriptive method can also be used to reason about such objects. RubySpec (Section 2.6) is a test framework which follows this approach for the most part.

5.1.2 Specifying visual languages

Visual programming languages (Section 4.1.1) use graphical elements in lieu of a textual syntax. Nevertheless, they are governed by the very same laws as regular programming languages. However, depending on the concrete implementation, input can be a challenge for testing. Even if an interpreter does not provide a way to process visual programs in an alternate text-based or binary format, there must at least be some sort of user interface to interact with it. In such a case the test framework evidently needs to include a library for the automated testing

in graphical user interfaces. When testing third-party implementations without direct access to the code base, this is likely the only possible approach.

If the software system that implements a visual language, including the interpreter, has a well-defined and accessible API, the test framework can alternatively directly interface with the system. This should also be the preferred technique for programmers who are writing their own code and who use BDD in their development process. The following sections will look at a practical example where this course of action has been taken.

5.2 The Catrobat programming language

Catrobat is an open source, cross-platform, mobile, visual programming language. It was inspired and is heavily influenced by Scratch,[1] a visual programming environment for children and teenagers, developed by the MIT Media Lab.

Different from Scratch, Catrobat from the very beginning aimed to be optimized for mobile devices, i.e., smartphones and tablets. It was at first implemented in Java as the Android application *Catroid*. Meanwhile, versions for iOS and Windows Phone, as well as a version for the web based on HTML 5, are also in development. The Android app Catroid has already been released to the public on the Google Play Store under the name *Pocket Code*.[2] It is at this time still the most feature complete implementation of the visual programming language Catrobat.

5.2.1 Scratch

Scratch was originally written in the Smalltalk programming language and implemented on the Squeak virtual machine. Recently it was completely rewritten

[1] http://scratch.mit.edu (accessed 2013-10-05)
[2] http://play.google.com/store/apps/details?id=org.catrobat.catroid (accessed 2013-10-05)

from the ground up in *Actionscript* using Adobe Flash, and it is now available as a web application on many compatible browsers.

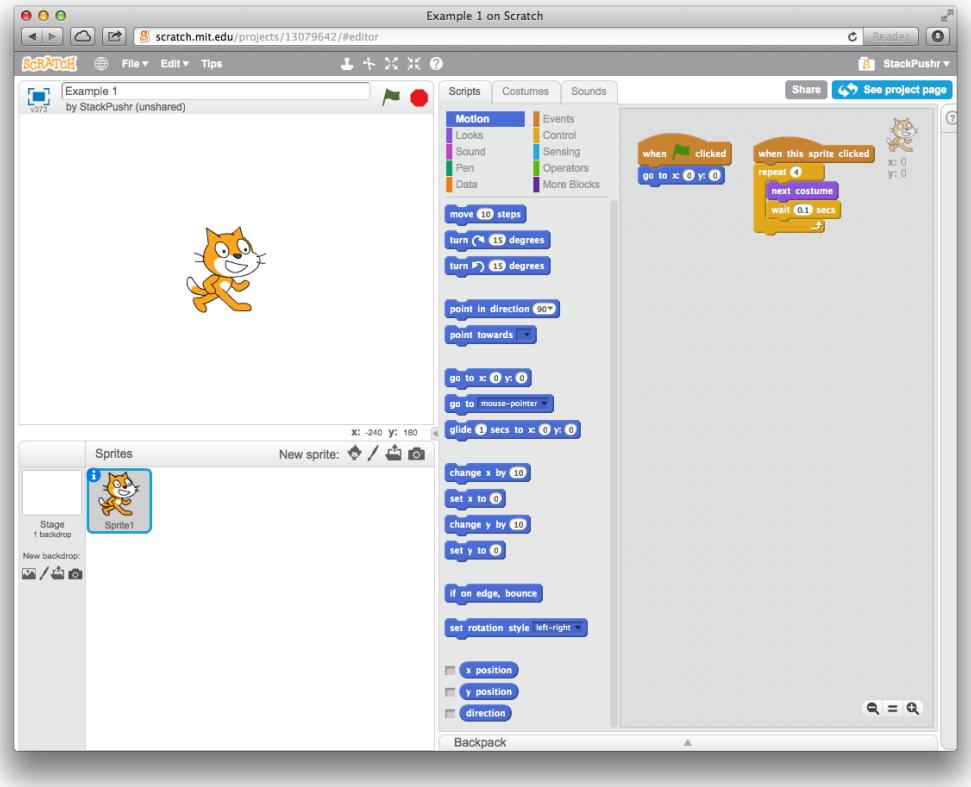

Figure 5.1: Scratch version 2.0

In a nutshell, the concept of Scratch is to create interactive animations with the help of rearrangeable *blocks*. The blocks are organized into lists called *scripts* which need to have a specialized *event* block at the top. Scripts, on the other hand, belong to a single *sprite*, an object of sorts that contains scripts, *costumes* (images) and *sounds* (audio files).

Scratch differentiates between nine different types of blocks in total. The most important ones, however, are *events* and *control* blocks. Events, with the exception of broadcast blocks, are always the first elements in a script. They are message responders that react to the program being started, a key being pressed, or other predefined events. If scripts are like subroutines, then event blocks basically define

the identifiers and parameters of functions. Control blocks determine the control flow inside a script with iteration and selection statements. The other categories all contain blocks that either perform some kind of action or blocks that evaluate statements and return a primitive type (i.e., string, number, boolean).

To the user, every script is an asynchronous process in Scratch. The concurrency model is fully transparent, which means that there is no way for the user to directly control individual threads. Only the *broadcast and wait* block gives a little bit of control to the user by letting him decide to halt the execution of the current script until any message receivers are finished. Furthermore, scripts are comparable to function objects, which can have just a single instance (singleton). As a result, the same script cannot be executed multiple times in parallel. Because Scratch 2.0's implementation language Actionscript is a single threaded programming language, there can be no actual data races in the background. However, there are still frequently unexpected outcomes. For example, Figure 5.2 shows a program with three scripts. From the visual syntax alone it is unclear what the output is going to be. Here is what really happens behind the scenes:

When starting the program, the start script is the first to begin executing. The asynchronous broadcast reaches the the other scripts in the order they were added to the program. Because the rightmost script was added last, it overwrites the value that was set by the leftmost script. Even though the wait block is set to *0*, it introduces a small delay. As a result, the output of this program is actually *"hello, people."*

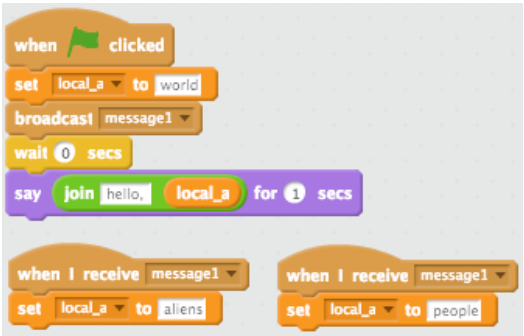

Figure 5.2: Concurrency artifact in Scratch 2.0

When running the same program on the older version 1.4 of Scratch, the output *"hello, world"* is generated instead. The reason for this is that the wait block causes no delay in that version, and so the first script continues before the other two can finish. Another peculiarity of the older version is that the order in which scripts are arranged on the screen influences the order in which they will be executed.

It is understandable that introducing advanced synchronization mechanisms would unnecessarily overcomplicate the user-friendly design of Scratch. Nevertheless, the discrepancy between the new and the old implementation is very likely the result of undefined, or unspecified, behavior.

5.2.2 Catroid

Catroid was the first visual programming environment for Catrobat. It is predominantly aimed at children and teenagers with the goal of increasing their interest in computer science and to promote logical thinking and programming skills. The visual aspect of the system is supposed to make programming more attractive and easier to understand for the target audience. Because of the rapid growth in the sector of mobile computing, the increasing proliferation of smartphones and tablets amongst children, and the intuitive interaction design of these devices, Catroid was created for the mobile Android platform [29], [28]. The software is being actively developed by the Catrobat open source project on Github.[3]

Figure 5.3: Script view in Pocket Code (Catroid) 0.9.4

Similar to Scratch, Catroid is also centered around the scripting and audiovisual animation of graphical objects, or sprites, on the screen. This makes the language very accessible to the prime target audience of children and young teenagers. The

[3]http://github.com/Catrobat (accessed 2013-10-05)

programming environment is not limited to this purpose however. On Android, the Pocket Code application also allows users to control Lego Mindstorms© toy robots and the Parrot AR Drone© with their personally created visual programs.

5.2.3 Language concepts

Catrobat borrows from Scratch the notion of block elements or *bricks*, as they are also called. These are atomic elements that represent specific statements of the programming language. For example, there are control flow blocks for structured programming but also more specialized blocks which directly alter a graphical object on the screen. Although it is implemented in a different programming language and with a different architecture, Catrobat also retains the principal language concepts and program composition from Scratch. A program comprises one or more objects and possibly a set of global variables. An object can possess local variables and typically also has two sets of specialized attributes, namely *looks* (images) and *sounds* (audio files), for the audiovisual animations. Most importantly, an object contains a list of *scripts*. They are the code portion of a Catrobat program and contain the bricks which in turn incorporate the logic of the entire program. Scripts essentially behave like subroutines because they are triggered by different external or internal events (Figure 5.4).

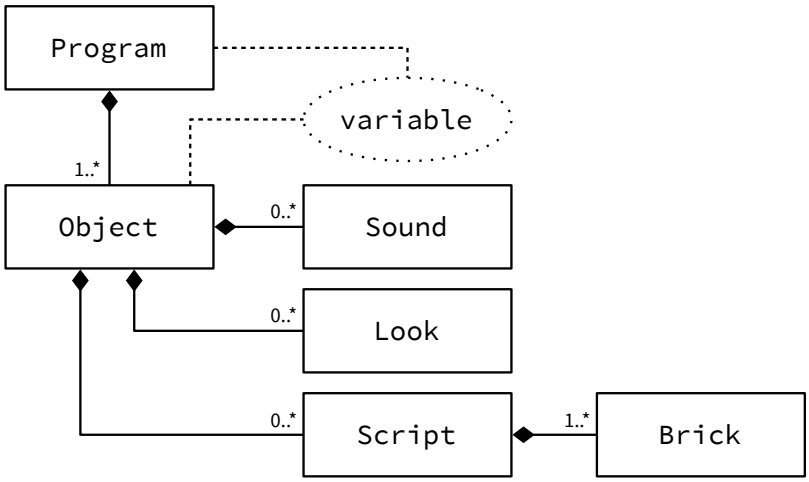

Figure 5.4: Composition of elements in the Catrobat language

Catrobat has five discernible types of bricks in total:

1. **Control**. These are either event bricks that respond to messages, broadcast bricks to send messages, or control flow bricks (e.g., loops, conditionals).
2. **Motion**. These are bricks to alter the position of an object (sprite) on the screen either instantly or by using a predefined animation.
3. **Sound**. This category includes bricks that control the playback of audio files associated with an object or change the system's volume level.
4. **Looks**. These bricks change the appearance of the visual representation of objects. A different look can be chosen from the object's internal list, or the overall visibility of the object can be changed.
5. **Variables**. This last category contains bricks to initialize variables or to change their values.

Unlike Scratch, Catrobat does not yet include a particular set of bricks for arithmetic operations. Instead, the integrated development environment of Catroid (Pocket Code) provides a module called *forumla editor*. This is a special input method that is made available to the user whenever a literal value can be entered. The formula editor is also responsible for managing global and object-local variables during the runtime of the program.

Broadcast bricks and when scripts

There are two different kinds of broadcast bricks. The regular, non-blocking *broadcast* brick sends a message to every receiver, and the containing script immediately continues executing or stops if there are no further bricks. The synchronous, blocking *broadcast wait* brick sends a message and causes the containing script to block until every responding script has finished execution.

Like in Scratch, every script is triggered by a certain event. A *when brick* defines the reception of a specific message as an event. Scripts that begin with a when brick are called *when scripts* and they are responders to broadcast bricks.

5.3 Specification of Catroid with Cucumber

One of the primary goals of the Catrobat project is to provide dependable functionality and a consistent overall experience by ensuring that visual programs behave exactly the same on every implementation platform. This is one of the reasons that Cucumber with its platform independent ubiquitous language was selected at as the tool of choice to create an executable documentation for the programming language. The following examples show how some of the primary features of Catrobat have been specified in a behavior-driven way using Cucumber scenarios and steps.

5.3.1 Creating a program

For many of the individual specifications, the basis is a simple Catrobat program with a single object. Although this foundation could be set up and prepared as part of the supporting test code, it is important to document as much of the implementation as possible by using executable Cucumber features. This makes it much easier to maintain the code base and it also makes the specification more useful to implementing programmers.

There are two steps necessary to crate a basic program (Listing 5.1). In the first step an empty program container is created. The underlying step definition is shown in Listing 5.2. In this example a parameter for the program name is omitted, but it would be no problem at all to create a second definition to support this potential requirement.

```
1 Background:
2   Given I have a Program
3   And this program has an Object 'Object'
```

Listing 5.1: Steps for program creation

A program is an instance of the class *Project* in the Catroid implementation of Catrobat. Other types as well may have different names in different implemen-

tations. However, by using steps with a high granularity or resolution for every individual aspect of Catrobat, it is unproblematic for an implementation to use distinct denominations. By any means the test code inside step definitions should be short, concise and do only one specific thing. How well this requirement can be fulfilled strongly depends on the modularity of the application under test. If the application does not have a well designed API, the test code has to work around any deficiencies.

```
1  @Given("^I have a Program$")
2  public void I_have_a_program() throws IOException {
3    ProjectManager pm = ProjectManager.getInstance();
4    pm.initializeNewProject("Cucumber", getContext(), /*empty*/ true);
5    Project project = pm.getCurrentProject();
6  }
7
8  @Given("^this program has an Object '(\\w+)'$")
9  public void program_has_object(String name) {
10   int lookId = org.catrobat.catroid.R.drawable.default_project_mole_1;
11   ProjectManager pm = ProjectManager.getInstance();
12   Project project = pm.getCurrentProject();
13   Sprite sprite = Util.addNewObjectWithLook(getContext(), project, name, lookId);
14   Cucumber.put(Cucumber.KEY_CURRENT_OBJECT, sprite);
15 }
```

Listing 5.2: Step definitions for program creation

After creating an empty program, the second step is to add an Object to it. In the step definition a new object is instantiated from the *Sprite* class of Catroid and is assigned the provided name. The declaration and initialization of objects is not done directly by the use of Catrobat statements (bricks), but rather by manually adding them to a project inside Catroid's IDE. For this reason the step does not have to be any more explicit. In the corresponding step definition the object is created and added to the program using supporting test code from a utility class. As the name implies, this step should follow the instantiation of the program and therefore it is assumed that a valid instance is available through the *ProjectManager*.

Unfortunately, the following steps cannot rely on a similar mechanism for access to the created object. This is why a shared global variable, a static instance of a Java

collection in this case, must be used to store a reference to this object. Although a local member attribute to the containing class would also be an option, step definitions usually reside in different classes and need a way to share information with each other.

In every feature where having a program that contains a single object is the fundamental requirement, a *Background* (Section 2.5) is used to define the two necessary preceding steps. With the help of this construct, these steps are automatically executed before every example and need not be additionally included in each Scenario.

5.3.2 Specification of a loop

Like in every programming language, a loop is also an essential control structure in Catrobat. Until recently, this functionality used to be only documented by a relatively complex unit test (Listing 5.3). The test method in question first initializes a set of local variables. Then it creates a sprite object and an associated script object to which the necessary brick elements are finally added. The actual execution of the script happens inside the *while* loop, where a method is called which belongs to a backend graphics library that is employed by Catroid.

A *Repeat brick* and the corresponding *Repeat end* brick are the two elements which make up a loop in the Catrobat language. By definition, any bricks that are positioned between those two are repeatedly executed for a given number of iterations. In this case the bricks enclose another brick which has the purpose to modify a position attribute of the sprite. Albeit functional, this test cannot be regarded as a useful specification mainly for two reasons: Firstly, the Java code is far too difficult to read in order to get an impression of the loop functionality, at least in a short amount of time. Secondly, the test as a whole requires considerable efforts to be translated into other programming languages on other platforms.

By defining the feature at hand with Cucumber instead, both the intent and expected behavior can be made much clearer. Furthermore, the whole example is

```
1  public void testRepeatBrick() throws InterruptedException {
2    final int loops = 4, deltaY = -10;
3    Sprite sprite = new Sprite("sprite");
4    Script script = new StartScript(sprite);
5
6    RepeatBrick repeatBrick = new RepeatBrick(sprite, loops);
7    LoopEndBrick loopEndBrick = new LoopEndBrick(sprite, repeatBrick);
8    repeatBrick.setLoopEndBrick(loopEndBrick);
9
10   script.addBrick(repeatBrick);
11   script.addBrick(new ChangeYByNBrick(sprite, deltaY));
12   script.addBrick(loopEndBrick);
13
14   sprite.addScript(script);
15   sprite.createStartScriptActionSequence();
16
17   while (!sprite.look.getAllActionsAreFinished()) {
18     sprite.look.act(1.0f);
19   }
20
21   assertEquals(loops * deltaY, sprite.look.getYInUserInterfaceDimensionUnit());
22 }
```

Listing 5.3: Original Repeat brick test

broken up into smaller, reusable step definition methods which are much simpler to port. Listing 5.4 shows a complete example for a behavior-centric specification of the Repeat brick. The example only specifies the basic functionality of a loop, namely repeating a given set of bricks for a number of times. As will be discussed later on, there are also performance related properties which are dealt with in separate feature files.

Below the name of the feature there is a explanatory line of text that describes the intent of a Repeat brick. This part is called the *narrative* in Dan North's original story structure (Section 2.3.2), and while it does not contain any executable statements, it is still important to give a brief summary of the specified feature and its scenarios. If this portion of text tends to get too long and unwieldy, then it is most likely that the feature and the described intent are too generic. A Repeat brick, for example, has many desired properties, each of which can and should be specified in an individual Cucumber feature file.

```
1  Feature: Repeat brick
2  A Repeat Brick should repeat another set of bricks a given number of times.
3
4  Background:
5    Given I have a Program
6    And this program has an Object 'Object'
7
8  Scenario: Increment variable inside loop
9    Given 'Object' has a Start script
10   And this script has a set 'i' to 0 brick
11   And this script has a Repeat 8 times brick
12   And this script has a change 'i' by 1 brick
13   And this script has a Repeat end brick
14
15   When I start the program
16   And I wait until the program has stopped
17   Then the variable 'i' should be equal 8
```

Listing 5.4: Cucumber feature for a Repeat brick

As mentioned before, the *Background* is a reoccurring element that declutters scenarios by grouping together essential steps, which are executed before every example. Here it already becomes apparent that the Repeat brick feature contains more logic than the original unit test. The reason for this is that the behavior-driven approach of Cucumber stimulates the composition of holistic models which look at the system as a whole. While the essence of every feature is principally defined by the scenarios, a Background gives the necessary context for a practical use case.

Like the feature itself, every *Scenario* can have a short and descriptive name which should be carefully selected. In this example a script is constructed that will be automatically executed when the whole program is started. The script contains bricks to declare and initialize a variable that is then incremented by one within a loop. The step definition to initialize variables makes use of Catroid's internal *formula editor*, which is a separate module. The underlying code can be found in Listing A.5. In the end an assertion is made that the variable should equal the amount of total iterations. Because of the whole-system approach, which considers a complete program, this Cucumber specification effectively uses more Java

code behind the scenes than the original unit test while still being shorter in length and exhibiting far greater clarity.

```
1  @And("^this script has a Repeat (\\d+) times brick$")
2  public void script_has_repeat_times_brick(int iterations) {
3    Sprite object = (Sprite) Cucumber.get(Cucumber.KEY_CURRENT_OBJECT);
4    Script script = (Script) Cucumber.get(Cucumber.KEY_CURRENT_SCRIPT);
5
6    Brick brick = new RepeatBrick(object, new Formula(iterations));
7    Cucumber.put(Cucumber.KEY_LOOP_BEGIN, brick);
8    script.addBrick(brick);
9  }
10
11 @And("^this script has a Repeat end brick$")
12 public void script_has_repeat_end_brick() {
13   Sprite object = (Sprite) Cucumber.get(Cucumber.KEY_CURRENT_OBJECT);
14   Script script = (Script) Cucumber.get(Cucumber.KEY_CURRENT_SCRIPT);
15
16   LoopBeginBrick begin = (LoopBeginBrick) Cucumber.get(Cucumber.KEY_LOOP_BEGIN);
17   Brick brick = new LoopEndBrick(object, begin);
18   script.addBrick(brick);
19 }
```

Listing 5.5: Step definitions for a Repeat brick

Listing 5.5 contains the two step definitions for creating a Repeat brick and its counterpart, a Repeat End brick. The Catroid IDE is implemented in such a way that the user does not need to worry about the Repeat End brick because it is automatically added or removed together with the first brick of the loop. Of course, the Cucumber feature cannot rely on the convenience of the IDE, and thus an appropriate step definition is required. Although it would theoretically be possible to automatically add the Repeat End brick after a given number of consecutive bricks and to omit the additional step, this would be rather unintuitive and actually make the specification less concise. It is also worth saying that these two step definitions in particular need to use the global state in order to share the instance of the first Repeat brick with the Repeat End brick.

Finally, within Listing 5.6 there is the step definition that creates a Change Variable brick by employing the internal formula mechanism of Catroid. The purpose of this brick is to change the value of a stored variable by a given amount, which is

```
1  @And("^this script has a change '(\\w+)' by (\\d+.?\\d*) brick$")
2  public void script_has_change_var_by_val_brick(String name, String value) {
3    Sprite object = (Sprite) Cucumber.get(Cucumber.KEY_CURRENT_OBJECT);
4    Script script = (Script) Cucumber.get(Cucumber.KEY_CURRENT_SCRIPT);
5    Project project = ProjectManager.getInstance().getCurrentProject();
6
7    UserVariable var = project.getUserVariables().getUserVariable(name, object);
8    if (var == null) {
9      var = project.getUserVariables().addSpriteUserVariableToSprite(object, name);
10   }
11
12   FormulaElement elem = new FormulaElement(ElementType.NUMBER, value, null);
13
14   Brick brick = new ChangeVariableBrick(object, new Formula(elem), var);
15   script.addBrick(brick);
16 }
```

Listing 5.6: Step definition for a Change Variable brick

stated as a literal value.

5.3.3 Running Catrobat programs

Considering that Catroid is a mobile Android application which implements a visual programming language, there is inevitably some framework code required to interact with a user interface. Robotium[4] is such a framework that extends the existing Android test facilities, and gives improved control over various screen elements. The *Solo* class of Robotium is used inside the step definition for the invocation of a program (Listing 5.7) to navigate through different views, and to eventually initiate program execution.

Because a Catrobat program itself is really only a container, its execution conventionally begins with the start of the first script and ends with termination of the last script. By this definition the program must contain at least one *Start* script which will be automatically signaled at the beginning. In order to determine the start and end of the scripts, a set of two different wait locks and callback methods

[4]https://code.google.com/p/robotium (accessed 2013-10-05)

```
1  @When("^I start the program$")
2  public void I_start_the_program() throws InterruptedException {
3    programWaitLock = new Semaphore(programWaitLockPermits);
4    addScriptEndCallbacks();
5
6    Solo solo = (Solo) Cucumber.get(Cucumber.KEY_SOLO);
7    assertEquals(MainMenuActivity.class, solo.getCurrentActivity().getClass());
8    solo.clickOnView(solo.getView(R.id.main_menu_button_continue));
9    solo.waitForActivity(ProjectActivity.class.getSimpleName(), 3000);
10   assertEquals(ProjectActivity.class, solo.getCurrentActivity().getClass());
11   solo.clickOnView(solo.getView(R.id.button_play));
12   solo.waitForActivity(StageActivity.class.getSimpleName(), 3000);
13   assertEquals(StageActivity.class, solo.getCurrentActivity().getClass());
14
15   synchronized (programStartWaitLock) {
16     if (!programHasStarted) {
17       programStartWaitLock.wait(10000);
18     }
19   }
20 }
```

Listing 5.7: Step definition for program invocation

is used inside the Java code of the step definition classes. Special bricks of a type that is declared inside the test code are added to the top and bottom of each script. Such a brick then has the ability to call a method which is also inside the test code to signal the start or end of a script. These "callback bricks" are respectively added either inside the step definitions for starting the program (Listings 5.7, A.2), or for creating a start script (Listing A.1).

5.3.4 Behavior of script invocation

In Catrobat, Scripts begin running in response to an event, which is the same behavior as in Scratch. An event can be the start of the whole program, an external input event on the hardware, or some kind of internal event. The so-called *Broadcast* bricks and *When* scripts are the two components of the latter example. Broadcasts are signals or undirected messages, which are sent into the Catroid runtime. They do not contain information about the sender or any other data

than a string, which is just the name of the message itself. A When script can generally either respond to a pre-defined event, e.g., a touch input, or to a broadcast message. While this model is reminiscent of function calling, it has more similarities with the dynamic approach of programming languages like Smalltalk and Ruby.

```
1  Scenario: Broadcast brick sends message in program with two When scripts
2
3    Given 'Object' has a Start script
4    And this script has a Broadcast 'hello' brick
5    And this script has a Wait 200 milliseconds brick
6    And this script has a Print brick with '-S-'
7
8    Given 'Object' has a When 'hello' script
9    And this script has a Wait 100 milliseconds brick
10   And this script has a Print brick with '-A-'
11
12   Given 'Object' has a When 'hello' script
13   And this script has a Wait 300 milliseconds brick
14   And this script has a Print brick with '-B-'
15
16   When I start the program
17   And I wait until the program has stopped
18   Then I should see the printed output '-A--S--B-'
```

Listing 5.8: Cucumber scenario for a Broadcast brick

When specifying the properties of Broadcast bricks and When scripts, there are two fundamentally important modes of behavior that need to be documented. The first one is the simple convention that every When script for given message A will start running as the direct result of a broadcast of the message A. Listing 5.8 specifies a scenario where two When scripts react to the same message broadcast.

In this example timings are used to force the scripts into a deliberate order. This is important to ensure that no side effects that are uncorrelated with the documented behavior affect the outcome of the test. The method of printed output was chosen because text strings can be very easily compared to each other, and they do a good job at illustrating the entire scenario. The **Print brick** is actually not part of the Catrobat programming language, but has been implemented inside the test code.

At the time of writing Catroid did not yet include any functionality that would have been suitable for this feature.

```
1  Scenario: Program with two start scripts and one When script
2
3    Given 'Object' has a Start script
4    And this script has a Broadcast 'hello' brick
5
6    Given 'Object' has a Start script
7    And this script has a Wait 100 milliseconds brick
8    And this script has a Broadcast 'hello' brick
9
10   Given 'Object' has a When 'hello' script
11   And this script has a Print brick with '-A-'
12   And this script has a Wait 300 milliseconds brick
13   And this script has a Print brick with '-B-'
14   And this script has a Wait 300 milliseconds brick
15   And this script has a Print brick with '-C-'
16
17   When I start the program
18   And I wait until the program has stopped
19   Then I should see the printed output '-A--A--B--C-'
```

Listing 5.9: Cucumber scenario for the restarting of When scripts

The second essential behavior that When scripts are expected to have comes to light in a specific situation. When scripts actually behave like *singleton objects* in a way that they are only instantiated once and do not get duplicated when they receive a message. This means that sending the same message twice in quick succession can have one of two different effects. Either a reacting When script queues up the messages it receives and runs from start to finish, or it immediately restarts itself for every message. The second type of behavior is the one that has been implemented in Scratch and Catroid. Within Listing 5.9 there is is a scenario that specifies this feature with the help of printed output. Again, timings are used to enforce a deterministic sequence of execution that can be reliably verified every time.

5.3.5 Concurrency and wait locks

Because broadcasts are undirected messages they are generally asynchronous in both Scratch and Catroid. This means that a script will continue right away to run any following bricks that come after a Broadcast brick was sent. However, this behavior may not always be desired. For example, a Start script may invoke a number of separate initialization scripts that should finish before the program continues with the first script. For this reason there is another kind of Broadcast brick that blocks the parent script until every reacting When script has ceased execution. This *Broadcast Wait* brick thus effectively allows to enforce the synchronization of parts of the control flow. In Listing 5.10 there is another Cucumber scenario which illustrates the expected behavior. If in this example a conventional Broadcast brick was used instead, then the Start script would be the first to finish.

```
1  Scenario: Broadcast Wait brick sends message in program with two When scripts
2
3    Given 'Object' has a Start script
4    And this script has a BroadcastWait 'hello' brick
5    And this script has a Print brick with '-S1-'
6
7    Given 'Object' has a When 'hello' script
8    And this script has a Wait 100 milliseconds brick
9    And this script has a Print brick with '-W1-'
10
11   Given 'Object' has a When 'hello' script
12   And this script has a Wait 200 milliseconds brick
13   And this script has a Print brick with '-W2-'
14
15   When I start the program
16   And I wait until the program has stopped
17   Then I should see the printed output '-W1--W2--S1-'
```

Listing 5.10: Cucumber scenario for a Broadcast Wait brick

Concurrency, of course, always introduces a lot of complexity into a programming language, even if the possibilities are strictly limited like they are in Catrobat. Here, correct and verifiable specification is the key to a well-designed and well-implemented system. In fact, a discrepancy between Catroid and Scratch was only

discovered by chance, but the correct behavior has since been well documented and exactly specified with the help of Cucumber. One of the many advantages of the ubiquitous language Gherkin is that it allows developers to discuss concepts at by means of tangible examples. The example for the aforementioned concurrency issue is revealed in Listing 5.11.

```
1  Scenario: Broadcast Wait brick is unblocked when the same message is resent
2
3    Given 'Object' has a Start script
4    And this script has a BroadcastWait 'hello' brick
5    And this script has a Print brick with '-S1-'
6
7    Given 'Object' has a Start script
8    And this script has a Wait 200 milliseconds brick
9    And this script has a Broadcast 'hello' brick
10
11   Given 'Object' has a When 'hello' script
12   And this script has a Print brick with '-W1-'
13   And this script has a Wait 400 milliseconds brick
14   And this script has a Print brick with '-W2-'
15
16   When I start the program
17   And I wait until the program has stopped
18   Then I should see the printed output '-W1--S1--W1--W2-'
```

Listing 5.11: Cucumber scenario for the blocking behavior of Broadcast bricks

The behavior in question arises from a situation where a Broadcast Wait brick is blocking a script because it is waiting for the reacting When scripts to finish. Ordinarily there would be only one possible outcome, that is to say the waiting script would simply finish after the other scripts. But what happens when there is an additional script with a Broadcast brick that sends the same message another time? Given that the behavior of restarting When scripts described in Section 5.3.4 still holds true, the very When scripts that the waiting script is anticipating to finish are indeed restarted. Now one possibility would be that because the When scripts did not actually finish (because they were restarted), the Broadcast Wait brick continues to block its containing script. The other possibility, which is the one that has been implemented within Scratch and Catroid, is to unblock the Broadcast Wait brick and let the waiting script proceed instantly. It becomes

clear that stating intricate circumstances like these only in unstructured natural language is quite inappropriate. With Cucumber, however, even very complex problems can be stated clearly.

5.3.6 Considering uneven performance

Another topic of great importance is the specification of performance. In order to achieve the goal of providing the same experience across multiple platforms, Catrobat programs also need be consistent in the amount of time they need for certain operations. In most cases it is considered better if a task is carried out as well and fast as possible in a programming language. However, the visual nature of Catrobat involves specific actions which directly interact with animated objects on the screen. One common task, for example, is to modify the position attribute of an object's sprite in such a way that it appears to move in a gliding motion. Specifically, one would increment or decrement the coordinate values inside a loop. Now the arising problem is that if the time for one iteration of a Repeat brick would be only bounded by hardware speed, then the animated sprite would move faster on better hardware and more sluggishly on slower hardware. Since new generations of hardware tend to continuously improve, it is thus sufficient to define a lower temporal bound for one loop iteration (Listing 5.12).

This Cucumber feature relies completely on native features of the Catrobat programming language to specify the expected behavior of a Repeat brick. The scenario involves two scripts which start running at the same time and continue to run concurrently. One of the scripts contains a loop which continuously increments a variable. The other script waits two seconds and then checks the value of the variable. If a single iteration takes at least 20 milliseconds (0.02 seconds), then the variable should be less than or equal to 100. In other words, only 100 iterations at the most should have been possible within two seconds. An assertion for a lower bound of iterations is omitted because there are separate features for specifying the general functionality and expected minimum performance of a Repeat brick.

```
1  Scenario: No more than 100 iterations in 2 seconds
2
3    Given 'Object' has a Start script
4    And this script has a set 'i' to 0 brick
5    And this script has a set 'k' to 0 brick
6    And this script has a Wait 2 seconds brick
7    And this script has a set 'k' to 'i' brick
8
9    Given 'Object' has a Start script
10   And this script has a Repeat 400 times brick
11   And this script has a change 'i' by 1 brick
12   And this script has a Repeat end brick
13
14   When I start the program
15   And I wait until the program has stopped
16   Then the variable 'k' should be less than or equal 100
```

Listing 5.12: Cucumber scenario for Repeat brick delay

In most cases the test will produce a result of almost exactly 100 iterations, depending on the hardware. When the Cucumber tests are executed on virtualized systems with an emulator, however, it is possible that the amount is much lower. The Java code of the step definitions consists only of basic assertions, visible in in Listing A.4.

5.4 Lessons learned

The previous sections have demonstrated how an executable specification which is written in the ubiquitous language Gherkin can be used to document and test the visual programming language Catrobat as it is implemented in the Android application Catroid. The behavior-driven approach of specification by example definitely has a number of advantages, but in order to realize its full potential for cross-platform applications there is still some work to be done.

5.4.1 Advantages of specifying Catrobat by example

The Android module of Cucumber JVM has already been successfully integrated with the existing code base of Catroid. Several features have been written since, and numerous practical advantages have been identified.

Verifiable documentation. Feature files have indeed turned out to be extremely valuable as written documentation. Before the introduction of BDD with Cucumber, story cards pinned on a whiteboard and other loose documents were the only foundation for the large code base of Catroid. The established unit tests are already mostly derived from these story cards as a result of the TDD process. Nevertheless, they are not very suitable for gaining a complete understanding of the whole system. Gherkin features on the other hand are not only very readable documentation, but most importantly, they are verifiable by automated testing. The more natural syntax of Gherkin is also an advantage of Cucumber over model-based testing. Because specification by example also means to think about the requirements first and write them down, depending on the discipline of the developers the resulting documentation should always be complete.

Agile development process. A common misunderstanding about BDD is that customers or other non-technical stakeholders are supposed to write stories in the ubiquitous language. While this can be possible, programmers should instead integrate writing feature files into their development process. Because the Catroid development team already uses a modified form of Kanban, it is very easy to do just that. With the integration of Cucumber, a new story card can be accompanied by a Gherkin feature which illustrates concrete use cases in the form of scenarios. When submitting bug reports or feature requests to Github, a written example can now be used to better explain the correct behavior of the system. Best of all, any Gherkin scenarios can be directly tried out on Catroid without having to abstract them into unit tests first.

Testing requirements. The measure of code coverage states how large that portion of a software is which has been throughly tested. But the larger a system is, the more individual components it contains. The different ways these components interact with each other, and how they interact with the user comprise the software functionality as a whole. The functional requirements can potentially be tested by integration tests or acceptance tests. Cucumber features with their combinable and reusable steps are much more efficient for this task however. With the ubiquitous language Gherkin, every relevant behavior can be stated precisely and with a concise structure. Another advantage is the increased turnaround time during development. Existing Cucumber features can be modified without having to recompile the code base. Once a significant number of step definitions has been accumulated, new features can be composed without even having to add new glue code.

Easier cross-platform development. Although Cucumber has not yet been put to use on the other implementation platforms of Catrobat, the process will be very similar. Until now the development teams of the iOS, Windows Phone and web versions of the visual programming language did not have any other guideline than the existing Java code based on the Android version. Even without employing Cucumber themselves, those teams will for the first time be able to rely on a platform independent, behavior-centric documentation in the form of feature files. Critical features like the loop delay or the blocking behavior of bricks are documented and specified in a plain and readable language. As soon as Cucumber is available on the other platforms, the complete set of feature files can be shared by all the different development teams. They will then only need to implement step definitions for every step in order to be able to automatically verify the specification. In such an environment synergetic effects will allow the teams to benefit from each others' work, as the underlying feature files can be developed together across all platforms.

5.4.2 Necessary future improvements and limitations

Although the specification of Catrobat by example has initially shown many promising advantages, BDD with Cucumber is certainly no golden bullet. Like every methodology of software development, specification by example is also limited by how thoroughly it is exercised. If the demanded features are not formulated specifically enough, it still cannot be ensured that the software will meet the requirements. This can be especially difficult when trying to design a programming language, visual or not. The developers must take great care that they consider all the aspects of the system and then come up with user stories and key examples for the expected behavior. In Section 5.1.1 different approaches to applying a story structure to a visual programming language were introduced that helped to make the process easier. Similarly, the quality of the test code inside step definitions also heavily depends on the abilities of the programmers.

Another limitation at this time is that the cross-platform aspect of the Catrobat specification could not yet be fully realized. The reason is that Cucumber and Gherkin must first be natively implemented on other mobile platforms than Android. Other mobile test frameworks which also use Gherkin but do not execute Cucumber natively on the target platform cannot directly interact with the application code, and thus are no substitute. In Section 3.5 it was briefly discussed what steps have to be taken to achieve a native implementation of Cucumber on the operating systems Windows Phone and iOS.

Finally, even the most comprehensive test and specification framework can be limited by some of the inherent properties of the hardware of mobile devices. The inconsistency of performance or the varying availability of certain features can be a challenge for a uniform test base across devices. Because the advantage of using Cucumber significantly depends on the ability to use the same Gherkin feature files on all platforms, any discrepancies must be accounted for in some way. When considering only one operating system platform, the differences between mobile devices are usually hardware specific properties like screen size and resolution, CPU performance, and RAM size. With some effort, these differences can

be accounted for inside the test code and by carefully selecting test criteria for the Cucumber features, e.g., using a lower bound for the time limit of a loop iteration (Section 5.3.6).

Coping with missing hardware components is rarely necessary but can be rather complicated. Because Catrobat features abstractions for hardware sensor measurements at the language level, e.g., device rotation, a complete specification of the visual programming language must also include tests for these components. If a physical device or emulator instance lacks the proper hardware to deliver the expected sensor data, certain tests would probably fail. This issue can potentially be mitigated by using mocking frameworks inside the test code, which can stand in for the missing hardware.

When it comes to providing a consistent set of Cucumber files as the definite specification of Catrobat across not only different hardware but also different operating systems, more challenges arise. While the data format delivered by sensor measurements is in most cases equal across devices running the same operating system, different operating systems may specify different formats for values of device orientation, coordinates of touch input, etc. If the Gherkin feature files contain information about the expected format for such data, it is an obligation for the programmers to implement any necessary transformations inside the step definitions. However, the Cucumber test framework provides no help in itself for such issues.

5.4.3 Conclusion

Proper specification is the precondition for an all-round successful software project. The focus of many developers lies often exclusively on building the software right, but not on building the right product for the given requirements. Behavior-driven development and specification by example are sometimes criticized for not adding anything valuable to a development process, but instead for only increasing the management overheads. Software tools like Cucumber make it easier to realize the concept of specification by example with less effort, but they cannot prevent

failure or ensure success. Although the approach definitely has some limitations, in the instance of Catroid and Catrobat actual real life benefits could be demonstrated.

The challenge of specifying a visual programming language on diverging mobile platforms truly demands new and different approaches. Using Cucumber to compose system-centric specification documents which are executable is a very powerful solution to many problems such as this challenge poses. One reason why the underlying methods are so suitable in this case is that BDD enforces a holistic vision of the software that inspires developers to think about architecture and behavior first before writing any code. The other reason is that the tools which are part of the test framework Cucumber, like the platform-independent ubiquitous language Gherkin, are ideal for use with diverse mobile applications.

In conclusion, the practical examples of a dynamic and executable specification for Catrobat suggest that this approach will be of great value for the continued development of this visual programming language. Hopefully the presented ideas will further improve software quality and project efficiency to help Catrobat reach its respectable goals.

A Appendix

A.1 Listings

```
@Given("^'(\\w+)' has a Start script$")
public void object_has_start_script(String object) {
  programWaitLockPermits -= 1;
  Project project = ProjectManager.getInstance().getCurrentProject();
  Sprite sprite = Util.findSprite(project, object);
  StartScript script = new StartScript(sprite);

  script.addBrick(new CallbackBrick(sprite, new CallbackBrick.BrickCallback() {
    @Override
    public void onCallback() {
      synchronized (programStartWaitLock) {
        if (!programHasStarted) {
          programHasStarted = true;
          programStartWaitLock.notify();
        }
      }
    }
  }));

  sprite.addScript(script);
  Cucumber.put(Cucumber.KEY_CURRENT_SCRIPT, script);
}
```

Listing A.1: Step definition for a Start script (Section 5.3)

```
private void addScriptEndCallbacks() {
  Project project = ProjectManager.getInstance().getCurrentProject();
  for (Sprite sprite : project.getSpriteList()) {
    for (int i = 0; i < sprite.getNumberOfScripts(); i++) {
      sprite.getScript(i).addBrick(new CallbackBrick(sprite, new BrickCallback() {
        @Override
        public void onCallback() {
          programWaitLock.release();
        }
      }));
    }
  }
}
```

Listing A.2: Method to append a callback brick (Section 5.3)

```
@And("^I wait until the program has stopped$")
public void wait_until_program_has_stopped() throws InterruptedException {
  // While there are still scripts running, the available permits should be < 1.
  programWaitLock.tryAcquire(1, 60, TimeUnit.SECONDS);
}
```

Listing A.3: Step definition for determining program termination (Section 5.3)

```
@And("^this script has a set '(\\w+)' to '(\\w+)' brick$")
public void script_has_set_var_to_var_brick(String a, String b) {
  Sprite object = (Sprite) Cucumber.get(Cucumber.KEY_CURRENT_OBJECT);
  Script script = (Script) Cucumber.get(Cucumber.KEY_CURRENT_SCRIPT);
  Project project = ProjectManager.getInstance().getCurrentProject();

  UserVariable varA = project.getUserVariables().getUserVariable(a, object);
  if (varA == null) {
    varA = project.getUserVariables().addSpriteUserVariableToSprite(object, a);
  }

  FormulaElement elemB = new FormulaElement(ElementType.USER_VARIABLE, b, null);

  Brick brick = new SetVariableBrick(object, new Formula(elemB), varA);
  script.addBrick(brick);
}
```

Listing A.4: Step definition for assigning one variable to another (Section 5.3)

```java
@And("^this script has a set '(\\w+)' to (\\d+.?\\d*) brick$")
public void script_has_set_var_to_val_brick(String a, String b) {
  Sprite object = (Sprite) Cucumber.get(Cucumber.KEY_CURRENT_OBJECT);
  Script script = (Script) Cucumber.get(Cucumber.KEY_CURRENT_SCRIPT);
  Project project = ProjectManager.getInstance().getCurrentProject();

  UserVariable varA = project.getUserVariables().getUserVariable(a, object);
  if (varA == null) {
    varA = project.getUserVariables().addSpriteUserVariableToSprite(object, a);
  }

  FormulaElement elemB = new FormulaElement(ElementType.NUMBER, b, null);

  Brick brick = new SetVariableBrick(object, new Formula(elemB), varA);
  script.addBrick(brick);
}
```

Listing A.5: Step definition for assigning a literal to a variable (Section 5.3)

```java
@Then("^the variable '(\\w+)' should be greater than or equal (\\d+.?\\d*)$")
public void var_should_greater_than_equal_float(String name, float expected) {
  Sprite object = (Sprite) Cucumber.get(Cucumber.KEY_CURRENT_OBJECT);
  Project project = ProjectManager.getInstance().getCurrentProject();

  UserVariable var = project.getUserVariables().getUserVariable(name, object);
  assertNotNull(var);

  float actual = var.getValue().floatValue();
  assertThat(actual, greaterThanOrEqualTo(expected));
}
```

Listing A.6: Step definition for comparing variables (Section 5.3)

```java
@And("^this script has a Wait (\\d+.?\\d*) seconds brick$")
public void script_has_wait_ms_brick(float seconds) {
  Sprite object = (Sprite) Cucumber.get(Cucumber.KEY_CURRENT_OBJECT);
  Script script = (Script) Cucumber.get(Cucumber.KEY_CURRENT_SCRIPT);

  int millis = Math.round(seconds * 1000f);
  Brick brick = new WaitBrick(object, millis);
  script.addBrick(brick);
}
```

Listing A.7: Step definition for a Wait brick (Section 5.3)

```
*** Settings ***
Documentation     A resource file with reusable keywords and variables.
...
...               The system specific keywords created here form our own
...               domain specific language. They utilize keywords provided
...               by the imported Selenium2Library.
Library           Selenium2Library

*** Variables ***
${SERVER}         localhost:7272
${BROWSER}        Firefox
${DELAY}          0
${VALID USER}     demo
${VALID PASSWORD}    mode
${LOGIN URL}      http://${SERVER}/
${WELCOME URL}    http://${SERVER}/welcome.html
${ERROR URL}      http://${SERVER}/error.html

*** Keywords ***
Open Browser To Login Page
    Open Browser    ${LOGIN URL}    ${BROWSER}
    Maximize Browser Window
    Set Selenium Speed    ${DELAY}
    Login Page Should Be Open

Login Page Should Be Open
    Title Should Be    Login Page

Go To Login Page
    Go To    ${LOGIN URL}
    Login Page Should Be Open

Input Username
    [Arguments]    ${username}
    Input Text    username_field    ${username}

Input Password
    [Arguments]    ${password}
    Input Text    password_field    ${password}

Submit Credentials
    Click Button    login_button

Welcome Page Should Be Open
    Location Should Be    ${WELCOME URL}
    Title Should Be    Welcome Page
```

Listing A.8: Robot framework resource file, adapted from robotframework.org

A.2 Acronyms

ATDD acceptance test-driven development

BDD behavior-driven development

CFG context-free grammar

regex regular expession

SbE specification by example

TDD test-driven development

XP Extreme Programming

Bibliography

[1] G. Adzic. *Examples make it easy to spot inconsistencies*. Online; accessed 2013-10-05. May 2009. URL: http://gojko.net/2009/05/12/examples-make-it-easy-to-spot-inconsistencies.

[2] G. Adzic. *Specification by Example: How Successful Teams Deliver the Right Software*. Manning Pubs Co Series. Manning, 2011. ISBN: 9781617290084. URL: http://books.google.at/books?id=5F5PYgEACAAJ.

[3] Franz L. Alt. "The standardization of programming languages." In: *Proceedings of the 1964 19th ACM national conference*. ACM '64. New York, NY, USA: ACM, 1964, pp. 22.1–22.6. DOI: 10.1145/800257.808893. URL: http://doi.acm.org/10.1145/800257.808893.

[4] Ada Resource Association. *An ISO Standard Guards the Ada Hen House*. Online; accessed 2013-10-05. 2013. URL: http://www.adaic.org/ada-resources/standards/ada-95-documents/acaa.

[5] Ada Conformity Assessment Authority and R. L. Brukardt. *The Ada Conformity Assessment Test Suite (ACATS) Version 3.0 User's Guide*. Online; accessed 2013-10-05. 2008. URL: http://www.ada-auth.org.

[6] K. Beck. *Extreme Programming Explained: Embrace Change*. An Alan R. Apt Book Series. Addison-Wesley, 2000. ISBN: 9780201616415. URL: http://books.google.at/books?id=G8EL4H4vf7UC.

[7] K. Beck. *Test-driven Development: By Example*. Kent Beck signature book. Addison-Wesley, 2003. ISBN: 9780321146533. URL: http://books.google.at/books?id=gFgnde%5C_vwMAC.

[8] D. Chelimsky et al. *The Rspec Book: Behaviour Driven Development With Rspec, Cucumber, and Friends*. Pragmatic Bookshelf Series. Pragmatic Bookshelf, 2010. ISBN: 9781934356371. URL: http://books.google.at/books?id=0rxoPgAACAAJ.

[9] Yoonsik Cheon and GaryT. Leavens. "A Simple and Practical Approach to Unit Testing: The JML and JUnit Way." English. In: *ECOOP 2002 — Object-Oriented Programming*. Ed. by Boris Magnusson. Vol. 2374. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2002, pp. 231–255. ISBN: 978-3-540-43759-8. DOI: 10.1007/3-540-47993-7_10. URL: http://dx.doi.org/10.1007/3-540-47993-7_10.

[10] E. Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison Wesley Professional, 2004. ISBN: 9780321125217. URL: http://books.google.at/books?id=xColAAPGubgC.

[11] J. Fitzgerald and P.G. Larsen. *Modelling Systems: Practical Tools and Techniques in Software Development*. Cambridge University Press, 2009. ISBN: 9780521899116. URL: http://books.google.at/books?id=PwbqT8g1nfAC.

[12] John Fitzgerald, PG Larsen, and S Sahara. "VDMTools: advances in support for formal modeling in VDM." In: *SIGPLAN Not.* 43.2 (Feb. 2008), pp. 3–11. ISSN: 0362-1340. DOI: 10.1145/1361213.1361214. URL: http://doi.acm.org/10.1145/1361213.1361214.

[13] Andrew Harry. *Formal Methods: Fact File: VDM and Z*. New York, NY, USA: John Wiley & Sons, Inc., 1997. ISBN: 0471958573.

[14] A. Hellesoy and M. Wynne. *The Cucumber Book: Behaviour-Driven Development for Testers and Developers*. Pragmatic Programmers. Pragmatic Bookshelf, 2012. ISBN: 9781934356807. URL: http://books.google.at/books?id=oMswygAACAAJ.

[15] Google Inc. *Android Developer Reference — Application Fundamentals*. Online; accessed 2013-10-05. 2013. URL: http://developer.android.com/guide/components/fundamentals.html.

[16] Google Inc. *Android Developer Reference — Building and Running*. Online; accessed 2013-10-05. 2013. URL: http://developer.android.com/tools/building.

[17] C. Larman and B. Vodde. *Practices for Scaling Lean & Agile Development: Large, Multisite, and Offshore Product Development with Large-Scale Scrum*. Pearson Education, 2010. ISBN: 9780321685087. URL: http://books.google.at/books?id=fqdTsH36TVYC.

[18] Peter Gorm Larsen and John Fitzgerald. "Recent industrial applications of VDM in Japan." In: *Proceedings of the 2007th internatioanal conference on Formal Methods in Industry*. FACS-FMI'07. London, UK: British Computer Society, 2007, pp. 8–8. URL: http://dl.acm.org/citation.cfm?id=2227886.2227894.

[19] Peter Gorm Larsen et al. "The overture initiative integrating tools for VDM." In: *SIGSOFT Softw. Eng. Notes* 35.1 (Jan. 2010), pp. 1–6. ISSN: 0163-5948. DOI: 10.1145/1668862.1668864. URL: http://doi.acm.org/10.1145/1668862.1668864.

[20] P.G. Larsen, J. Fitzgerald, and T. Brookes. "Applying formal specification in industry." In: *Software, IEEE* 13.3 (1996), pp. 48–56. ISSN: 0740-7459. DOI: 10.1109/52.493020.

[21] Kim Marriott, Bernd Meyer, and KentB. Wittenburg. "A Survey of Visual Language Specification and Recognition." English. In: *Visual Language Theory*. Ed. by Kim Marriott and Bernd Meyer. Springer New York, 1998, pp. 5–85. ISBN: 978-1-4612-7240-3. DOI: 10.1007/978-1-4612-1676-6_2. URL: http://dx.doi.org/10.1007/978-1-4612-1676-6_2.

[22] G. Meszaros. *XUnit Test Patterns: Refactoring Test Code*. A Martin Fowler signature book. Addison Wesley Professional, 2007. ISBN: 9780131495050. URL: http://books.google.at/books?id=y4JuQgAACAAJ.

[23] D. North. *Introducing BDD*. Online; accessed 2013-10-05. 2006. URL: http://dannorth.net/introducing-bdd.

[24] D. North. *What's in a story?* Online; accessed 2013-10-05. 2007. URL: http://dannorth.net/whats-in-a-story.

[25] D. L. Parnas. "A technique for software module specification with examples." In: *Commun. 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.

[26] K. Pugh. *Lean-Agile Acceptance Test-Driven-Development*. Net Objectives Lean-Agile Series. Pearson Education, 2010. ISBN: 9780321719447. URL: http://books.google.at/books?id=tB23eWcG9DEC.

[27] M.L. Scott. *Programming Language Pragmatics*. Elsevier Science, 2009. ISBN: 9780080922997. URL: http://books.google.at/books?id=GBISkhhrHh8C.

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

[29] Wolfgang Slany. "Catroid: a mobile visual programming system for children." In: *Proceedings of the 11th International Conference on Interaction Design and Children*. IDC '12. Bremen, Germany: ACM, 2012, pp. 300–303. ISBN: 978-1-4503-1007-9. DOI: 10.1145/2307096.2307151. URL: http://doi.acm.org/10.1145/2307096.2307151.

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

[31] A. Thurston. *Ragel State Machine Compiler User Guide*. Online; accessed 2013-10-05. 2013. URL: http://www.complang.org/ragel/ragel-guide-6.8.pdf.

[32] Michael Tonndorf. "Ada conformity assessments: a model for other programming languages?" In: *Ada Lett.* XIX.3 (Sept. 1999), pp. 89–99. ISSN: 1094-3641. DOI: 10.1145/319295.319310. URL: http://doi.acm.org/10.1145/319295.319310.

[33] D.A. Watt, B.A. Wichmann, and W. Findlay. *ADA: language and methodology*. Prentice-Hall international series in computer science. Prentice/Hall International, 1987. ISBN: 9780130040862. URL: http://books.google.at/books?id=O9kmAAAAMAAJ.

[34] Peter Wegner. "The Ada language and environment." In: *SIGSOFT Softw. Eng. Notes* 5.2 (Apr. 1980), pp. 8–14. ISSN: 0163-5948. DOI: 10.1145/1010792.1010793. URL: http://doi.acm.org/10.1145/1010792.1010793.

[35] Wikipedia. *Oracle v. Google — Wikipedia, The Free Encyclopedia*. Online; accessed 2013-10-05. 2013. URL: http://en.wikipedia.org/wiki/Oracle_v._Google.