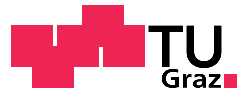

Benjamin Wullschleger, BSc

Writing and Testing an Audio Engine for Pocket Code on iOS

Master's Thesis

to achieve the university degree of

Master of Science

Master's degree programme: Information and Computer Engineering

submitted to

Graz University of Technology

Supervisor

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

Institute for Softwaretechnology
Head: Prof. Dr. Franz Wotawa

Graz, April 2020

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Date

Signature

Abstract

With the rise of smartphones in the last decade, operating systems of mobile devices have become increasingly powerful. This includes their capability of audio processing, which has been continuously improved and expanded. Many of today's mobile applications make heavy use of the operating system's audio processing capabilities, one of them being Pocket Code.
Pocket Code is a mobile integrated development environment (IDE) for teenagers and uses the visual programming language Catrobat. It offers an easy way to create programs and games and provides an audio engine that includes various audio playback, processing and recording features. Unfortunately, the audio engine of Pocket Code's iOS version has proven to be erroneous and unreliable, leading to many bugs in the application. The weaknesses and the planned implementation of additional audio features therefore raised the need for a complete redesign of the audio engine. This thesis documents this redesign process. It outlines the original state and weaknesses of the audio engine, defines the specifications of existing and new audio features, evaluates various potential audio processing frameworks on iOS, and describes the architecture of the redesigned engine.
To prevent further bugs in the future, automated tests are implemented for the audio engine. The thesis therefore explores some general concepts of automated testing and describes the testing framework provided by Apple. As automated testing of audio functionality is non-trivial, the thesis also researches and highlights different testing strategies specifically tailored to the testing of audio features. It then proceeds by looking at how these strategies can help the automated testing of Pocket Code's redesigned audio engine.

Contents

Abstract iii

1 Automated code testing: An introduction 1
- 1.1 The different levels of software testing . . . 3
  - 1.1.1 Unit testing . . . 4
  - 1.1.2 Integration testing . . . 6
  - 1.1.3 System testing . . . 7
  - 1.1.4 Acceptance testing . . . 8
- 1.2 The benefits of automated testing . . . 8
- 1.3 Properties of good automated tests . . . 10

2 Good practises and patterns for testable code 14
- 2.1 Types of test doubles . . . 15
  - 2.1.1 Dummy . . . 15
  - 2.1.2 Stub . . . 16
  - 2.1.3 Fake . . . 16
  - 2.1.4 Test spy . . . 17
  - 2.1.5 Mock . . . 17
- 2.2 Creating test doubles . . . 18
  - 2.2.1 Extract interface . . . 18
  - 2.2.2 Subclass and override . . . 20
  - 2.2.3 Using isolation frameworks . . . 20
- 2.3 Dependency injection . . . 22
  - 2.3.1 Constructor-based dependency injection . . . 23
  - 2.3.2 Setter-based dependency injection . . . 23
  - 2.3.3 Dependency injection with factory classes and methods 24

3 Xcode testing environment 26
- 3.1 Creating a test setup for Xcode . . . 26

3.2 Defining test cases 27
3.3 Asserts 28
3.4 Asynchronous testing 31
3.4.1 Testing asynchronous code with expectations 32
3.5 Third party testing tools for Xcode 38
3.5.1 Isolation frameworks 38
3.5.2 Other frameworks 39

4 Pocket Code **42**
4.1 What is Pocket Code? 42
4.2 Structure of a Pocket Code Project 44
4.2.1 Project level 44
4.2.2 Object level 44
4.2.3 Script level 45
4.2.4 Brick level 45

5 Audio processing on macOS and iOS **48**
5.1 Core Audio 48
5.2 High-level audio classes 50
5.3 AVAudioEngine 50
5.4 AudioKit 54

6 Original Pocket Code iOS audio engine **56**
6.1 Overview of Pocket Codes's audio features 56
6.1.1 Sound bricks 57
6.1.2 Speech bricks 59
6.1.3 Other audio features 60
6.2 Audio engine architecture 60
6.2.1 Audio engine components 60
6.2.2 Audio manager 63
6.3 Problems and limitations 65
6.4 Automated testing 68

7 New Pocket Code audio engine **71**
7.1 New audio features 71
7.1.1 Sound bricks 72
7.1.2 Effect bricks 72

7.1.3 Music bricks . . . 74
7.2 Choosing the right audio framework . . . 77
7.3 Building Pocket Code's audio graph . . . 79
7.3.1 Audio player . . . 79
7.3.2 Speech synthesizer . . . 81
7.3.3 Mixer . . . 81
7.3.4 Software instrument . . . 82
7.3.5 Volume . . . 85
7.3.6 Pitch effect . . . 85
7.3.7 Pan effect . . . 86
7.3.8 Combining nodes to a PCObject subgraph . . . 87
7.3.9 Combining subgraphs to a full graph . . . 89
7.4 Audio engine architecture . . . 89
7.4.1 CBAudioEngine . . . 90
7.4.2 CBSubgraph . . . 94
7.4.3 Other architectural elements . . . 96

8 Automated audio testing 98
8.1 The difficulties of audio testing . . . 98
8.2 Audio testing strategies in literature . . . 101
8.3 Bit-exact testing . . . 103
8.3.1 Bit-exact testing with hashes . . . 104
8.4 Parametric testing . . . 105
8.5 Audio fingerprinting . . . 107
8.5.1 General audio fingerprinting framework . . . 109
8.5.2 Audio fingerprinting framework for automated testing 110
8.5.3 Fingerprint extraction . . . 111
8.5.4 Similarity measures . . . 113
8.6 Open source audio fingerpinting libraries . . . 115
8.6.1 Chromaprint . . . 115
8.6.2 pHash . . . 119

9 Automated audio testing in Pocket Code on iOS 123
9.1 Unit testing . . . 123
9.2 Bit-exact testing . . . 126
9.2.1 Offline manual rendering . . . 126
9.2.2 Using offline manual rendering in Pocket Code . . . 128

9.3 Integration tests with audio fingerprinting 132
9.3.1 Capturing the audio output of test scenarios with audio taps 133
9.3.2 Choice of the audio fingerprinting framework 135
9.3.3 Creating and running audio fingerprinting tests in Pocket Code 135
9.3.4 Findings from using audio fingerprinting in Pocket Code 139

10 Conclusion and future work 143

Bibliography 147

List of Figures

1.1 Different levels of software testing 3

3.1 Xcode folder structure for automated tests 27
3.2 Assigning a test file to a test target 27
3.3 Test navigator 28

4.1 Pocket Code scripts and bricks 43
4.2 Pocket Code objects 43
4.3 Scripts, looks, and sounds sections 46
4.4 Brick selection menu 46

5.1 The architecture of Core Audio 49
5.2 AVAudioEngine processing chain 52
5.3 Processing graph 53

6.1 Sound bricks in Pocket Code 57
6.2 Speech bricks in Pocket Code 57
6.3 Original implementation of the Pocket Code audio engine . . 61

7.1 Visualization of *AKPlayer*'s/*AVAudioPlayerNode*'s functionality 80
7.2 Visualization of *AKMixer*'s/*AVAudioMixerNode*'s functionality 82
7.3 Sample instrument keyboard layout 84
7.4 Visualization of *AKSampler*'s functionality 85
7.5 Visualization of *AKVariSpeed*'s/*AVAudioUnitVarispeed*'s functionality 86
7.6 Visualization of *AKPanner*'s functionality 87
7.7 Functional diagram of a subgraph 88
7.8 Connected nodes forming a subgraph 88

7.9 Connecting subgraphs to a full graph 89
7.10 Visualization of the redesigned audio engine 91

8.1 Visualisation of a signal processed by a time variant effect . . 101
8.2 General audio fingerprinting processes 110
8.3 Audio fingerprinting process for automated testing 111
8.4 Chroma feature visualization 117
8.5 Filters used in Chromaprint 118
8.6 PRH and pHash flowchart 121

9.1 Inserting a node recorder into an audio graph 134

List of Listings

2.1 Creating a mock object with Mockito 21

3.1 Basic structure of an Xcode test class 29
3.2 Example of an asynchronous test case 36
3.3 Example of an predicate based asynchronous test case 38
3.4 A behaviour driven test with Quick 40
3.5 Asserting with Nimble and XCTest 41

9.1 Offline rendering with AudioKit 128
9.2 Bit-exact test case in Pocket Code 130
9.3 Audio fingerprinting test case in Pocket Code 139

List of Abbreviations

API Aplication Programming Interface
BER Bit Error Rate
BPM Beats Per Minute
FFT Fast Fourier Transform
IDE Integrated Development Environment
MIDI Musical Instrument Digital Interface
PCLook Pocket Code Look
PCM Pulse Code Modulation
PCObject Pocket Code Object
PCProject Pocket Code Project
PCSound Pocket Code Sound
PRH Philips Robust Hashing
SUT System Under Test
TTS Text-to-speech
XML Extensible Markup Language

1 Automated code testing: An introduction

Code testing is a concept as old as computer programming itself. Every software developer throughout the history of software development has written or performed a test case in one or the other way. In the era of programming with punched cards, testing the algorithms often involved the use of pen and paper, sketching and outlining the different control flows of the program to avoid unintended behaviour. An error during the execution of the program often lead to a major time loss since computational resources were very limited at that time[1]. Nowadays, a code test could be as simple as setting a breakpoint to inspect the state of a running program or creating a temporary button and output label in the user interface to execute a specific method and displaying its output. This kind of manual code testing works pretty well in small and non-commercial projects. However, in the increasingly large and complex code bases of today's commercial software projects, manual testing becomes a time-consuming and error prone task.
Typically, an application changes continually over its lifecycle. It grows as new features are added, which in turn leads to new tests that need to be performed. Unfortunately, the new features cannot be viewed in isolation, they almost always influence and affect previously written code. A 10% code change might have such widespread impact, that 100% of the application's features have to be retested[2]. At some point, manual testing can no longer keep up with this increasing workload. Bugs start to get unnoticed by testers and will be shipped to the customers with the final product. These bugs can lead to a significant financial loss for the distributor or user of the application. In 2002, the National Institute of Standards and Technology

[1] [Osh14], p. xvii.
[2] [Hay04], p. 5.

estimated the financial losses of the U.S. economy caused by software bugs at around 60 billion dollars. The study also found that a third of these costs could have been avoided with an improved testing infrastructure[3]. With the increasing digitalization, these figures are likely to be much higher today.
To overcome the difficulties of manual testing, software engineers have come up with various solutions to automate this process. These solutions, although different in their implementations, essentially all evolve around the same definition of automated testing which can be reduced to the following two (slightly modified) sentences found in [Osh14]:

"An automated test is a piece of code that invokes another piece of code and checks the correctness of some assumptions afterwards. If the assumptions turn out to be wrong, the test has failed."[4]

One of the most significant contributions to automated testing has been made by Kent Beck with the creation of a testing framework called "SUnit" for the programming language Smalltalk which he described in [Bec99]. The automated testing strategies of SUnit have been adopted by testing frameworks of many other programming languages and are still widely used today[5]. Nonetheless, still today, developers, clients and project managers often struggle with the concept of automated testing. Automated testing results in additional work for developers, not only in writing the tests themselves but also in writing the code that has to be tested. To make code testable, a developer needs to plan ahead and follow some specific concepts or else ends up with code that is very hard to test. Oftentimes, developers are not willing or able to put in this extra time for writing test cases, especially when they are working under time pressure to finish a certain task. Clients and project managers also often neglect the long-term benefits of automated tests by only seeing the additional costs caused by their creation.

[3] [Nat02].
[4] [Osh14], p. 4.
[5] [Fea04], p. 48.

1.1 The different levels of software testing

Since software development is a multi-faceted and complex process, it usually involves more than just one type of testing. Typically, a software product will be tested on different levels and with different types of tests and tools. Each level has its specific testing goals[6]. Starting with testing small units of code, each subsequent level tests bigger parts of the system for correct functionality and integration until at last the system gets tested as a whole. This hierarchical testing structure can be seen in figure 1.1. While the number of testing levels may vary in literature, the four levels depicted in figure 1.1 are some of the most commonly found in software projects and will be looked at in more detail in this chapter.
In the following explanations, the term "system under test" (SUT) is introduced, which is frequently used throughout this thesis. It always refers to the code that is being tested in a test case, regardless of whether it is just a single method or many different classes interacting with each other.

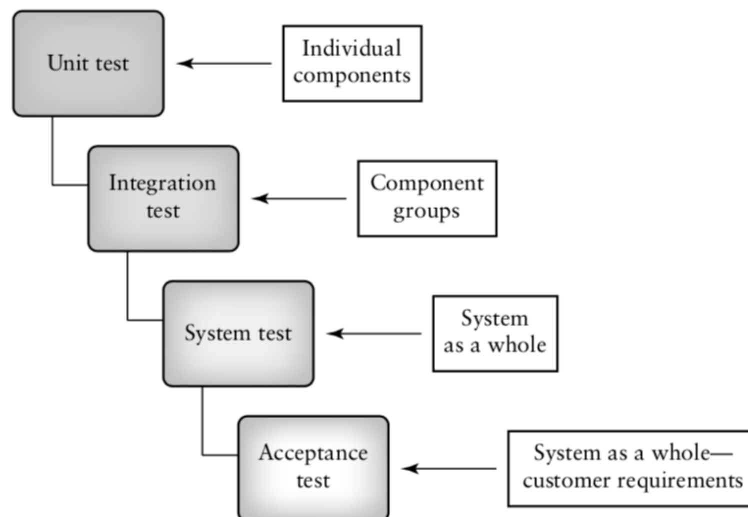

Figure 1.1: The different levels of software testing[7]

[6][Buro3], p. 133.
[7][Buro3], p. 134.

1.1.1 Unit testing

The name "unit test" already says a lot about this level of testing and especially highlights the subjects of these tests: individual units. The Oxford Dictionary gives the following definition of a unit:

"An individual thing or person regarded as single and complete but which can also form an individual component of a larger or more complex whole"[8].

The "larger or more complex whole" can be interpreted as the system or a subsystem that is developed during a software development process. The units on the other hand, are tightly interconnected and form the building blocks of this system. The fact that a unit is regarded as "single and complete" implies two things:

1. Although connected with other units, a unit is an independent building block and can therefore also be tested independently from other units.
2. A unit consists of code, that produces some sort of a meaningful and testable end result, which in turn contributes to the functioning of the whole system.

However, the exact definition of a unit in the context of unit testing varies. Some simply describe it as the smallest testable part of an application, others define it as a single method or a single class. In reality, it does not make much sense to work with such a static definition. Even if a test might cover the smallest testable part of an application, it does not necessarily mean that it is a meaningful test. A specific method or class might only deliver a meaningful result in conjunction with another method or class. A good example of a smallest possible but meaningless test would be the test of a getter or setter method. While it is possible to test a getter or setter method, such a test will not add much to the improvement of code quality since those methods do not contain any logic.
A more pragmatic definition of a unit can be found in [Osh14]. Roy Osherove defines it as a "unit of work", which is the sum of actions that lead to a

[8] [Oxf].

single end result after the invocation of a public method. By this definition, a unit can span a single method or multiple classes[9].

Independent from the exact definition of a unit, a unit test's main goal is to uncover functional and structural defects of the unit under test[10]. A test has to fulfill several properties to be considered a unit test:

1. **Isolation:** The unit under test should be tested independently and isolated from other units[11]. In reality, almost every unit is connected to other objects and components. To still achieve sufficient isolation during a test, there are specific techniques to eliminate those dependencies which will be covered in chapter 2. A test that communicates with a database, makes network calls or reads from the file system is generally not considered a unit test because it depends on other components[12]. The degree of isolation directly impacts the following properties.
2. **Error localization:** As previously mentioned, unit tests are designed to test small units of work. This guarantees that an error can be quickly localized whenever a test fails[13]. The more dependencies the code under test has, the harder it gets to localize the error in case of a failed test. Tests with many dependencies can also be useful, but usually fall into the category of integration testing which is discussed in the next section. To facilitate the localization of errors, automated tests usually provide a text output that compares the expected with the actual behaviour at the end of a failed test case.
3. **Speed:** Unit tests are meant to be executed frequently during development to detect possible errors that resulted from recent code changes. If tests take too long, developers might not run them often enough or not at all, which in turn negatively affects code quality. Unit tests should run fast or else cannot be considered to be unit tests. Even tests that run longer than 100 milliseconds are already considered to be slow[14]. Considering that big software projects build up tens of

[9] [Osh14], p. 4.
[10] [Bur03], p. 133.
[11] [Fea04], p. 12.
[12] [Fea04], p. 14.
[13] [Fea04], p. 12.
[14] [Fea04], p. 13.

thousands of unit tests over time, it becomes clear that running a full test suite would otherwise become a very time consuming endeavour. Tests with many dependencies tend to be larger and slower than tests with isolated units.

1.1.2 Integration testing

As described earlier, units are the building blocks of bigger systems. Units interact with and depend on each other to provide the system's intended functionality. Once all necessary units have been written and tested, they can be assembled to form a higher-level system. Of course, a system consisting of units that all passed their corresponding unit tests is not automatically guaranteed to work. In fact, one little mistake inside or at the interface of a unit can have a negative impact on the entire system. This is where integration testing comes into play.
In [Het88], integration testing is defined as "an orderly progression of testing in which software and/or hardware elements are combined and tested until the entire system has been integrated."[15]. With this definition, some properties can be derived that clearly differentiate unit tests from integration tests[16]:

1. While unit tests validate the correctness of the structure and internal logic of a component, integration testing can detect weaknesses at the interfaces of system components[17].
2. Unit tests try to eliminate dependencies to isolate a component from external influences. For integration testing, units now integrate some of their real dependencies to form increasingly larger subsystems. If a test makes use of techniques like threading, accesses the file system or a real database or uses the system time, it is most likely an integration test.
3. Using real dependencies increases the risk of creating inconsistent tests. The content of a real database can change, the timing of threads

[15][Het88], p. 15.
[16][Osh14], p. 7.
[17][Buro3], p. 152.

can differ slightly, and the system time will be different every time a test executes.

4. Due to the fact that integration tests comprise multiple interacting components, they also cover a bigger part of the system's code base than a unit test. Additionally, the individual components have to be set up properly before a test can be executed. Due to these circumstances, integration tests are usually much slower than unit tests.
5. Unlike unit tests, it might be hard to find the reason for a failed integration test. When many components of a system work together and contribute to a final result, the root of an error does not become apparent immediately.

Most of the above properties are somewhat negative, which could give the misleading impression that unit testing should be preferred over integration testing. This, of course, is not the case. Both, unit tests and integration tests, are equally important for a good testing infrastructure and cannot replace one another. While unit tests are designed to test small units of work, other results can only be observed and tested when multiple units are combined and therefore rely on integration testing. The above-mentioned disadvantages of integration tests are a direct result of the higher complexity of the SUT. It is therefore important to clearly separate integration tests from unit tests[18].

1.1.3 System testing

System testing starts once all the different parts of a system are fully integrated. This includes the actual software, the hardware it runs on, and any external hard or software that a system depends on. Although this level of testing also intends to uncover possible defects, it is much more focused on assuring performance, usability and reliability of the system. Depending on the setup of the system, testing on this level can include the use of special software and hardware to perform the tests and measure their results[19].

[18] [Osh14], p. 7.
[19] [Bur03], p. 134.

Often, system testing is not executed by the software developers themselves, but by separate testing personnel or team leaders[20].

1.1.4 Acceptance testing

Once system testing has been completed, testing can move on to an additional level called acceptance testing, which is needed if a software is custom made for a client[21]. It is used to verify that all requirements and specifications are met that were agreed on with the client and that the client is paying for and that the final product is ready to be used. Acceptance tests are often a subset of the system tests and verify the correctness of important business transactions of the system. They are usually performed by the client itself or by an external company hired by the client[22]. As acceptance tests as well as system tests are often not written and performed by software developers and follow different procedures than unit and integration tests, they will not be discussed any further. The term "automated testing" therefore only refers to unit and integration tests in all following chapters.

1.2 The benefits of automated testing

Apart from verifying code and detecting errors in software, automated tests bring additional benefits to software projects. This section gives a quick overview of those benefits.

Improved documentation and specification Automated tests can be seen as a living documentation of a software's intended behaviour and specifications. They describe the system's behaviour and results to different inputs and are continuously updated as the software evolves. If a software developer is unable to come up with a test for a specific piece of code, it can mean that the specifications are not clear enough and need some further

[20] [Het88], p. 11.
[21] [Bur03], p. 135.
[22] [Het88], p. 11.

refinement. A well written test case can give a developer who is unfamiliar with a particular part of a software much faster insight into the functionality of the tested code compared to looking at the production code itself[23].

Improved design If a test case is hard to write, the cause for this is often a badly designed SUT with high coupling or low cohesion[24]. Coupling and cohesion are two attributes used to determine how simple or complex code is designed. Cohesion describes to which extent all elements within a software unit are related together. Elements within a highly cohesive unit all belong together, serve a single purpose and therefore keep the design simple. Coupling on the other hand is an attribute that describes the extent of interaction and interdependence between two software units. Tightly coupled units are harder to reuse, modify or fix in case of errors. They are also harder to isolate, which makes automated testing more difficult. Writing code with a design as simple and clean as possible should be the goal of every developer, and such code is usually loosely coupled as well as highly cohesive. As multiple studies have shown, loose coupling and high cohesion can lower the proneness to error and increase maintainability as well as testability of software[25].
Writing automated tests forces developers to think ahead when writing production code and makes them factor in coupling and cohesion from the beginning. This leads to better software design and testable code[26]. Automated tests also encourage developers to write interfaces that are easy to use instead of just easy to implement, especially when tests are written before writing the production code.

Cost reduction Studies have shown, that errors found during development or unit and integration testing are a lot cheaper to fix than errors that are found during system testing or even on a productive system[27]. Generally, the earlier a bug is discovered, the lower are the costs it causes. On the other

---

[23] [GS17], p. 6.
[24] [BA05], p. 50.
[25] [TKB18], p. 175 + 179.
[26] [Mar08], p. 133.
[27] [Lin02], p. 305 -307.

hand, the creation and maintenance of test cases consume a lot of time and generate additional costs which might seem to exceed the costs of manual testing at first glance. Considering that test cases, regardless of whether they are manual or automated, are executed and repeated many times over the life cycle of a software project, automated test cases soon exceed manual tests in terms of cost effectiveness as they only require an initial setup and sporadic maintenance work to be done and operate independently after that.

Increased confidence and trust in code Having many meaningful automated tests increases trust in the product for clients, project managers and developers. From a developer's point of view, having confidence in a project's code base is very important. Automated tests establish trust as they act as a safety net when code is being changed and inform the developers when they are introducing bugs. Refactoring code therefore becomes much easier and effective, as developers no longer have to look at every little detail of code that might be affected by a change, but can rely on running automated test cases to tell them whether the refactoring has broken any existing logic or not. This is especially helpful for developers that are not fully familiar with the code they are working on. Without having tests, developers can be reluctant to make changes to code as they might fear to introduce bugs that stay undetected[28]. Frequently executed automated tests not only increase trust in code, but also lead to better code quality and an increase in development productivity.

1.3 Properties of good automated tests

To use the full potential and benefits of automated testing, test cases should have certain properties and follow some basic rules[29]. Some properties like isolation, error localization and speed have already been discussed in connection with unit testing. Although these properties are particularly important for unit tests, they can't be fully neglected for integration testing.

[28] [Mar08], p. 124.
[29] [Mar08], p. 123 - 132.

The below paragraphs describe a few other properties of automated tests that are equally important for unit and integration testing. Carelessly designed automated tests which lack these properties could become counter productive and might increase the cost of maintaining, executing and understanding a test suite.

Clean tests Having clean tests means, that test code should adhere to the same high coding standards as production code. Test code that is written carelessly, is badly designed or muddled, requires a lot more maintenance when it has to be changed due to evolving production code. Writing clean tests also means designing them to be readable. Repetitive code or code that does not contribute to a better understanding of a test case (although relevant for its correct execution) should be extracted into separate functions with meaningful names[30]. If possible, test cases should be structured according to the "Build-Operate-Check" pattern, which divides a test case into three distinctive sections: A section where all required setup work is done (build), a section that performs the operations that the test case has to verify (operate) and a section that verifies the results of those operations (check)[31].

Single concept Some experts on automated testing recommend to have only a single assert statement per test case[32]. This means each test case should only verify the correctness of a single piece of information. The advantage of such tests is, that they only come to a single, unambiguous conclusion which is easy to understand. Sometimes however, the correct behaviour of the SUT cannot be determined by just one piece of data and requires the verification of multiple (partial) results. If this is the case, an automated test can contain more than one assert statement, but it is still important to limit the number of asserts to a minimum. A better rule might be, that a test case should only test a single concept[33]. Test cases that do

[30][Mar08], p. 124 - 126.
[31][Mar08], p. 127.
[32][PLP12], p. 2.
[33][Mar08], p. 131.

not follow this rule and test multiple unrelated concepts simultaneously quickly get unreadable and complicate the search for the cause of errors.

Independence A test case should not depend on the result or state of any other test case. Test cases should always be independent such that they can be executed in arbitrary order. If this is not the case, a failed test could cause other tests, which depend on it, to also fail. This complicates the error diagnosis and might obscure some actual bugs in the affected test cases[34].

Repeatability Running a test case should require no prior knowledge, and should be repeatable on all environments the software runs on. Anyone should be able to run it at the push of a button and verify its result[35]. Test cases that require manual setup or only run on specific environments will lead to developers not running the tests at all.

Consistency Given that no code changes have been made, a test case should always produce consistent results for different runs[36]. Tests that are unreliable and sometimes produce false negative outcomes cause unnecessary investigations into bugs that don't exist. Most unreliable tests fall into the category of integration tests, as they often depend on external resources or systems. Such resources or systems do not always behave as expected. Their response time is different with every test execution, they might be temporarily unavailable or the data they provide might change unexpectedly. If the creation of test cases which are influenced by factors beyond the developers control cannot be avoided, such tests should still be designed to be as consistent as possible. Often, test suites can be configured to repeat failed tests a certain amount of times, which can eliminate the risk of producing false negative test results. However, test cases that produce false negative results on a regular basis and not just sporadically are still likely to cause more harm than good.

[34][Mar08], p. 132.
[35][Osh14], p. 6.
[36][Osh14], p. 6.

Self-validation The result of a test case should always be a boolean value indicating whether the test passed or not. The test case has to be able to produce this result itself, without any further help of a developer. A test that requires the developer to look through a log file or manually verify a certain document will be slow and error prone[37].

[37][Mar08], p. 132.

2 Good practises and patterns for testable code

Object oriented code is usually tightly interconnected such that a single class can be made up of objects of many other classes. Those objects are called dependencies, as the containing class depends on their functionality to perform its own task. However, for automated testing, dependencies can be a problem. Initialising the SUT can become a cumbersome process, as every dependency has to be initialized with an instance suitable for the test case. The dependencies in turn contain dependencies themselves which also have to be initialized, and so on. Often, the correct behaviour of dependencies is not of interest in a test case and verified in separate tests. However, the dependencies still need to supply some meaningful input to the SUT[1]. Other dependencies might not even be used for the execution of a test case. Initializing unused or irrelevant dependencies and the dependencies' dependencies would therefore be a waste of time and resources. Another problem are dependencies that rely on external systems or functionality, like databases and APIs (application programming interface). Such dependencies are not always under control of the developers[2] and might be unavailable in a test environment or supply inconsistent data between test runs.
As mentioned in section 1.1.1, it is therefore important to isolate the SUT from all the unused, irrelevant, inconsistent and unavailable dependencies to gain full control over the test environment. This ensures that the test cases are clean, consistent and as quick as possible. Isolation is achieved by substituting dependencies with fake implementations called test doubles[3]. Test doubles implement an alternative behaviour that differs from the

[1] [Kac13], p. 75.
[2] [Osh14], p. 49.
[3] [Kac13], p. 71.

original dependency to avoid the testing difficulties mentioned above. This chapter highlights some of the most common practises and patterns to create testable code by using test doubles.

2.1 Types of test doubles

Depending on the dependency that needs to be replaced and depending on a test double's desired behaviour, test doubles need to be equipped with different capabilities. There are several types of test doubles, all used for different purposes. Five of the most commonly used types are explained below. Despite having four distinct categories, test doubles can often be a mixture of those types under real conditions[4].

2.1.1 Dummy

A dummy is the simplest type of test double and is used to replace dependencies that are not used and irrelevant in a test case. As such, a dummy does not collaborate with other objects nor is there any verification performed on its state during test execution. It is usually passed as a parameter of a direct method call and its job is to prepare the environment for testing and simply exist where it needs to exist[5]. An example would be a test case that calls a method with several parameters of which one is never used during test execution. Nevertheless, an instance of this specific class needs to be passed to the method. This instance can be substituted by a dummy object which has the same structure as the original, but does not provide any functionality. The use of dummies is quiet rare in automated tests as the effort of creating a dummy object is often higher than just initializing an instance of the original class.

[4] [Kac13], p. 72.
[5] [Kac13], p. 71 - 74.

2.1.2 Stub

Stubs are similar to dummies. They are also used to substitute dependencies in test cases and to properly prepare the environment for testing. Just like dummies, a stub's state and the return values of its methods are never used for verification during test execution. What sets a stub apart from a dummy is the fact, that a stub provides indirect data to the SUT which is needed to successfully run the test case. The input data is said to be indirect, because it is not provided to the SUT by the test case itself, but by a collaborating object of the SUT (the stubbed class). The SUT usually retrieves the indirect input data by calling certain methods of the stub. The behaviour of the stubbed class and the input data that it provides to the SUT have to be defined during test setup. This means that the stub has to be instructed what values its methods have to return when they are called during test execution. The stub will then simply return the specified values upon invocation of those methods instead of executing the method's original code[6]. When using a stub in a test case, the author of the test case has control over the creation of the stub, the behaviour of its methods and its integration into the SUT, but cannot directly control the use of and access to the stub by the SUT during test execution.

2.1.3 Fake

A fake is a test double that provides almost the same functionality as the real dependency. However, compared to the real implementation a fake is a simpler and more lightweight version, making it faster to set up and use. One of the most common examples of a fake is an in-memory database that is used for testing purposes instead of a real database server. The in-memory database has exactly the same functionality as the database server, but uses much less resources, and provides better and more reliable response times during testing. A fake is similar to a dummy or stub as it is part of the test setup but no verification is performed on it directly. Fakes are only used in integration testing[7].

[6] [Kac13], p. 75.
[7] [Kac13], p. 72.

2.1.4 Test spy

Determining the success of an automated test often works by verifying the correctness of a return value of the SUT. However, not all SUTs provide a return value that clearly indicates success or failure of a test case. Imagine a method that does not have a return value at all, but processes some data which it then passed to another collaborator of the SUT. This collaborator then stores an output file which contains the processed data. Verifying the success of such a method needs a different approach. This is where test spies come into play. A test spy is a test double that is used to verify indirect outputs, or simply put, outgoing communications from the SUT to one of its dependencies (the test spy object)[8]. Such indirect outputs or communications are usually just simple method calls that the SUT invokes on the test spy. A test spy behaves like the original object, but additionally records all method calls that have been made to it by the SUT and saves those records for later verification by the test case. In its verification phase, the test case can then check whether the SUT has invoked the correct methods with the correct parameters on the test spy[9]. In the example made above, the dependency responsible for storing the output file would be substituted by a test spy and the test case would simply verify whether the right data has been passed to the test spy when the relevant method was invoked.

2.1.5 Mock

The term ”mock” is often falsely used as a general term to refer to all types of test doubles. In reality, a mock is a specific type of test double that is very similar to a test spy. Just like a test spy, a mock is also used to verify the indirect outputs of the SUT, but the way those outputs are verified works in a different way. As described in the previous section, the verification of method calls on a test spy is done after the SUT has been exercised. However, mock objects can verify the correctness of method calls while the SUT is exercised and therefore fail test cases earlier than a test spy would. This is achieved by delegating the verification of the indirect outputs directly to

[8][Kac13], p. 75 - 76.
[9][Mes07], p. 538 - 539.

the mock object, whereas test spies perform the verification within the test method. The differences in using test spies and mocks becomes apparent when looking at the structure of a test case. When working with a test spy, the test case usually performs the test setup first (initializing the SUT and its dependencies) and exercises the SUT afterwards (executing the functionality which has to be tested). Finally, after the SUT has been exercised, the test case verifies the recorded direct outputs of the test spy. A test case with a mock follows a slightly different flow. After setting up the SUT and its dependencies, the test case defines the expected behaviour of the mock, stating which direct outputs are expected to be observed. After that, the SUT is exercised while verifying the direct outputs at the same time[10].

2.2 Creating test doubles

After discussing the different types of test doubles, this section describes the most common techniques used to create them. To be able to substitute dependencies of the SUT with test doubles, those dependencies have to be "broken", meaning that a pattern has to be implemented that allows the SUT to easily exchange a specific dependency used in a production environment with a test double needed for testing purposes. The following sections will discuss three of the most commonly used techniques. The first two of them can be implemented manually, while the last one provides an approach to automate this process.

2.2.1 Extract interface

"Extract interface" is a technique used to separate the specification of a class (the interface) from its actual implementation. This allows multiple implementations to exist for the same interface which can replace each other as needed. An interface is a class that cannot be instantiated and does not implement any functionality. It simply lists a number of method declarations without giving any details about their implementations. To

[10] [Kac13], p. 278 - 279.

equip an interface with some actual functionality, it has to be implemented by another class. This implementation class then has to specify the real implementation details of all the methods that are listed in the interface[11]. The great thing about this technique is, that whenever a variable of the interface type is used in code, this variable is completely agnostic to the actual implementation class it holds. The interface variable can hold an object of any class that implements this specific interface, making it possible to use different implementations for one and the same variable. This is very helpful for creating test doubles, as a test double is nothing else than an alternative implementation of an existing class.
Breaking a dependency by extracting an interface works as follows[12]:

1. Create a new, empty interface for the dependency that has to be broken.
2. Make the class of the dependency implement the interface. This does not break anything, since the interface does not yet specify any method declarations.
3. Substitute all occurrences of the dependency class in the project's code with the interface class.
4. Add a method declaration to the interface for each public method of the original dependency. The method declarations which should be a part of the interface can also be determined by compiling the code when the interface is still empty or incomplete. The compiler will show an error for each missing method declaration in the interface.
5. Create a test double class that implements the interface, providing alternative or empty method implementations to change or nullify their original behaviour.
6. Inject the newly created test double into the SUT (see section 2.3) and assign it to a variable of the interface class which previously contained the original implementation of the dependency.

[11] [Low17], p. 302 - 306.
[12] [Fea04], p. 362 - 369.

2.2.2 Subclass and override

"Subclass and override" is one of the most heavily used techniques to break dependencies. It uses inheritance to get access to behaviour that needs to be changed and to nullify behaviour that is irrelevant or undesired. The behaviour is modified by subclassing the dependency's class and overriding the methods which need to be changed. Production code will continue using the original dependency, while the test code will use the subclass with the modified behaviour to stand in as the test double.
Using subclass and override to break dependencies, requires the following steps to be performed[13]:

1. Identify the smallest set of methods that have to be modified in the dependency's class, so that it can be used in the test environment.
2. Make those methods overridable in the dependency's class. This process varies between different programming languages, but often includes setting those methods to a non final state and setting their access modifiers to protected or public.
3. Create a subclass that applies the necessary modifications to the class of the dependency by overriding its methods.
4. The newly created subclass can now stand in as a test double. To do so, exchange the class of the dependency with the newly created subclass in the SUT.

2.2.3 Using isolation frameworks

Test doubles have become an essential part of software development. They are often used so extensively by automated tests, that their manual creation would become a complicated and time consuming process, especially for large-scale projects. Many programming languages therefore provide built-in or third party libraries to facilitate and speed up the process of creating test doubles. Those libraries are often referred to as mocking frameworks although most of them provide functionality to create all previously mentioned test doubles. The term "isolation framework" is therefore a better

[13] [Fea04], p. 401 - 404.

way to describe them. Isolation frameworks provide an easy way to create test doubles without writing the entire classes from scratch. Instead, test doubles are created by the isolation framework during test execution, based on a few lines of code describing their behaviour. One of the best-known isolation frameworks is *Mockito*[14], an open source library available for the Java language which is also used in Pocket Code's Android version. Without going into much detail, listing 2.1 shows a simple example of how test doubles are created with Mockito.

```
@Test
public void testNeedsFuel_returnsTrue() {
        Car myCar = mock(Car.class);
        when(myCar.needsFuel()).thenReturn(true);
        assertTrue(myCar.needsFuel());
}
```

Listing 2.1: Creating a mock object with Mockito[15]

By using Mockito's `mock()` method, this unit test creates a test double which can stand in for objects of type `Car`. It then specifies that the `needsFuel()` method always returns true when it is called on this object by using Mockito's `when()` and `thenReturn()` methods. Although such a simplified example would not be used in a real automated test, it still illustrates the easy syntax isolation frameworks offer to quickly create test doubles and specify their behaviour. Creating test doubles and modifying the return values of their methods are just two of the many features modern isolation frameworks provide. They can also be used to intentionally throw exceptions, analyse the outgoing communications of spies and mocks and provide many different methods to assert the results of automated tests to just name a few of the other possibilities[16].

To dynamically create test doubles during runtime, isolation frameworks use a concept called reflection, which can be defined as "the ability of a

[14][Moc].
[15][Kac13], p. 67.
[16][Kac13].

running program to examine itself and its software environment, and to change what it does depending on what it finds"[17]. Reflection lets a program inspect its own classes and objects, revealing the properties they store and methods they use. Based on those findings, the running program can then dynamically alter property values, invoke methods or even create new objects of the inspected classes[18]. Many isolation frameworks, including Mockito, use reflection to create test doubles by implementing a proxy pattern[19]. As described in [MW06], the general intent of the proxy pattern is to "control access to an object by providing a surrogate, or placeholder, for it"[20]. A proxy therefore substitutes another object, forwarding requests made to the proxy to the underlying, original object. The interfaces of the original and proxy objects are usually very similar or even identical. The proxy pattern can be useful when the original object does not live up to a specific purpose. This could be the case if the original object takes a long time to load or if messages to the object need to be intercepted or modified. By using reflection, isolation frameworks are able to dynamically create proxy objects for the dependencies that need to be exchanged. The proxy then stands in as a test double in the SUT. Method calls to the test double are thus handled by the proxy which forwards them to the original object. The proxy has the ability to pre- or post-process the forwarded calls, altering the test double's behaviour as necessary[21].

2.3 Dependency injection

Dependency injection is a very simple but also immensely important concept in software development and especially automated testing. It is used to create seams, which is a term for places in code where different functionality can be plugged in[22], and simply describes the process of inserting

[17] [FF05], p. 3.
[18] [NK05], p. 200.
[19] [Ach14], p. 142.
[20] [MW06], p. 117.
[21] [FF05], p. 74.
[22] [Osh14], p. 54.

a dependency into another object from outside the object[23]. This is an important piece of information for working with test doubles as it is not only important to know how they are created, but also how they can be injected into the SUT. There are several dependency injection techniques which are explained below.

2.3.1 Constructor-based dependency injection

One way of injecting a dependency into the SUT is by using the constructor of the class that requires the dependency. To do so, an additional parameter of the dependency's type has to be added to the constructor. Inside the constructor, the dependency is then assigned to an internal variable of the SUT for later use. If the test doubles are created by the "extract interface" or "subclass and override" approaches, the new constructor parameter has to be of the interface or superclass type. When instantiating a new object of a class that uses constructor based dependency injection, the constructor parameters must always be provided. This approach is therefore a great way to signify that a dependency is non-optional and always required by the SUT. A drawback of constructor-based dependency injection is, that constructors can quickly become cluttered and less readable if too many dependencies are injected this way[24].

2.3.2 Setter-based dependency injection

When injecting a dependency via setter-based dependency injection, the class that requires the dependency implements a setter method for each dependency that has to be injected. This setter can then be called from within a test case, passing it the test double before the SUT is executed. Similar to constructor-based dependency injection, the parameter of the setter method also has to be of the interface or superclass type when using test doubles created with an "extract interface" or "subclass and override" approach. An advantage of using the setter-based variant is, that the setter only has to be

[23][Mis17], p. 279.
[24][Osh14], p. 57 - 60.

called in situations where the object actually needs the dependency and not every time a new object is initialized. Setter-based dependency injection is therefore preferably used for optional dependencies of the SUT or when the SUT already contains a default instance of a dependency that does not lead to any problems during test execution[25].

2.3.3 Dependency injection with factory classes and methods

Rather than passing dependencies to the SUT via constructors or setters, the factory approach retrieves dependencies just before they are used for operation. Factory classes and methods are design patterns that can be used to actively request and retrieve instances of dependencies from within a system instead of passing them to the system from the outside[26].

Dependency injection using a factory class With this approach, the SUT requests a dependency from a factory class. By calling a specific method of this factory class, the factory will return a fully initialized and working object of the dependency. For testing, the factory can be instructed to return a test double instead of a real object. This can be achieved in two ways:[27]

1. The method of the factory class which is responsible of returning the dependency is made static. This allows the dependency to be requested from everywhere in the code without having to instantiate the factory class itself. To substitute the dependency with a test double, the factory class contains a static setter method which can be used to inject the test double into the factory class. An advantage of this approach is, that the SUT does not have to be changed at all. It always requests the dependency from the same method of the factory class, regardless of whether it is running in a production or test environment. On the other hand, using static methods and variables in the factory

[25][Osh14], p. 61 - 63.
[26][Osh14], p. 63.
[27][Osh14], p. 63 - 66.

class can cause complications when the static states are not properly reset after each test.

2. Instead of injecting the test double to the factory class via a static setter method, it is also possible to create a fake version of the factory for testing purposes. This fake factory creates and returns a test double instead of the original object. In this scenario, an instance of the factory class has to be passed to the SUT. As fake factories which return fake dependencies can make code confusing, such setups should be avoided if possible.

Dependency injection using a local factory method When using a local factory method, the factory method is part of the system or class under test and not contained in a separate factory class. Instead of privately creating or accessing a dependency, the SUT implements a public getter method which executes all the code needed to instantiate or retrieve a dependency. The SUT then uses its own getter method to access the dependency. To inject a test double, a subclass of the class under test has to be created. The getter method used to retrieve the dependency is then overridden in the subclass so that it returns the test double instead of a real object. Instead of using the real class under test, the test case has to be executed with the subclass of the class under test. This approach, which is also called "extract and override", is a great and easy way to simulate fake inputs to the SUT, but less suitable for verifying interactions between the SUT and its dependencies[28].

[28] [Osh14], p. 66 - 69.

3 Xcode testing environment

Xcode first provided functionalities for automated testing in 2005 when Xcode 2.1 included a framework called OCUnit. OCUnit was an open source testing framework developed by Swiss company Sen:te[1]. OCUnit itself was based on SUnit, a unit testing framework developed by Kent Beck for the programming language Smalltalk[2].
Starting with Xcode 5 in 2013, Apple discontinued the integration of OCUnit and introduced its own testing framework called XCTest[3]. XCTest has since been continually improved and enhanced and provides a large set of tools to easily write many different kinds of automated tests. This chapter will highlight the most important parts of XCTest and explain how the tools are correctly used and set up within a project.

3.1 Creating a test setup for Xcode

Adding support for automated testing in Xcode is very easy to do. All that has to be done is creating a dedicated test target. This can either be done by clicking the "Include Unit Tests" and "Include UI Tests" checkboxes when setting up a new project, or by navigating to the File → New → Target menu and choosing the unit testing bundle or UI testing bundle in an existing project[4]. Usually, two test targets are created, one for unit/integration testing and another one for UI testing. Once those targets are set up, the project is ready to execute test cases. Test classes containing test cases are preferably stored in separate test folders in Xcode's project navigator as can be seen

[1] [Sen05].
[2] [Bec99], p. 277-287.
[3] [Appr].
[4] [Mis17], p. 15-18.

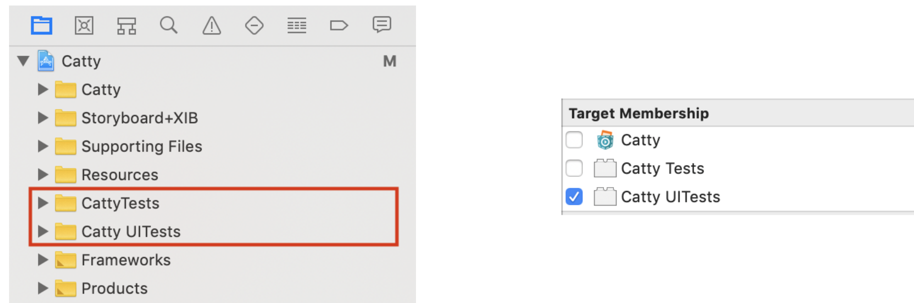

Figure 3.1: Unit and UI test folders in Xcode's project navigator

Figure 3.2: Assigning a test file to a test target

in figure 3.1. After a test case has been written, the file containing the test has to be assigned to one (or multiple) of the previously created test targets. This is done by selecting the file in the project navigator and then clicking on the according checkboxes in the inspector sidebar (see figure 3.2)

After that, test cases can be executed by opening the test file and clicking on the play button beneath the name of the test class or test method. Depending on whether the button beneath the class name or method name has been pressed, it will either start all test cases within that class or just a single test case. Alternatively, test cases can also be executed from the test navigator in the left sidebar. The test navigator hierarchically groups the test cases by class and target membership, so that it is possible to execute a single test case or all test cases of a class or a target. Figure 3.3 shows the test navigator.

3.2 Defining test cases

Writing test cases in Xcode is as simple as creating a test class and adding one or more test methods (test cases) to this class. Xcode then automatically identifies the newly created test cases and makes them executable as described in the previous section. For this automatic identification process to work, three conditions have to be met[5]:

[5][Appi].

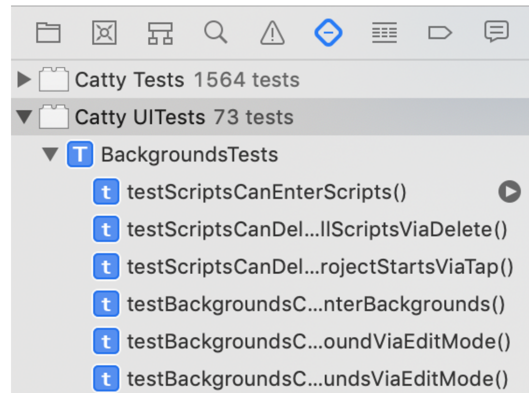

Figure 3.3: Test navigator

1. The test class must inherit from `XCTestCase`.
2. The names of the test methods must begin with the lower case word *test*.
3. The test file must be assigned to one of the previously created test targets.

`XCTestCase` is the base class for every test class written in Xcode and contains a lot of useful functionality and tools for writing automated tests. A test class that fulfills the above prerequisites is shown in listing 3.1. This listing additionally shows two other frequently used methods of `XCTestCase`: `setup()` and `tearDown()`. These methods are called before and after the execution of every test case and are generally used to bring the test class back into a desired state for the execution of the next test case.

3.3 Asserts

Assertion is the process of comparing the outcome of a test case with the expected outcome and failing the test case if the actual outcome does not match the expected one. Assertion is therefore a fundamental part of automated testing. It can be performed with various assertion methods that usually take the result of a test case or the state of the SUT and verify if

```
class CalculatorTests: XCTestCase {

    var calculator = Calculator()

    override func setUp( ) {
        super.setUp()
        calculator.reset()
    }

    override func tearDown() {
        calculator.clearResultHistory()
        super.tearDown()
    }

    func testTwoPlusTwo() {
        let result = calculator.plus(2, 2)
        XCTAssertTrue(result , 4)
    }
}
```

Listing 3.1: Basic structure of an Xcode test class

certain criteria are met. If some criteria are not met, the test case fails[6]. A test case can contain one or multiple assertion methods and only passes successfully, if none of the used assertion methods fail. In Xcode almost every test case contains at least one assertion method. There is one exception where this rule does not apply which will be discussed in subsection 3.4.1. XCTest provides 14 different assertion methods which are listed in table 3.1. All of those methods take an additional three optional parameters which are not specified in the table. The additional parameters are called `message`, `file` and `line` and are used to display the failure reason, the file name and the line number of the failed code in case the assertion fails.

[6] [Mis17], p. 7.

Assertion Method	*Description*
XCTAssert(expr, message)	Verifies the expression evaluates to true.
XCTAssertTrue(expr)	Same as XCTAssert
XCTAssertFalse(expr)	Verifies the expression evaluates to false.
XCTAssertNil(obj)	Verifies the object is nil.
XCTAssertNotNil(obj)	Verifies the object is not nil.
XCTEqual(obj1, obj2)	Verifies that object 1 is equal to object 2.
XCTNotEqual(obj1, obj2)	Verifies that object 1 is not equal to object 2.
XCTAssertGreaterThan(val1, val2)	Verifies that value 1 is greater than value 2.
XCTAssertGreaterThanOrEqual(val1, val2)	Verifies that value 1 is greater than or equal to value 2.
XCTAssertLessThan(val1, val2)	Verifies that value 1 is less than value 2.
XCTAssertLessThanOrEqual(val1, val2)	Verifies that value 1 is less than or equal to value 2. Fails the test case and displays an optional failure message otherwise.
XCTAssertThrowsError(expr)	Verifies that the expression (for example a function call) does not throw an error.
XCTAssertNoThrow(expr)	Verifies that the expression does throw an error.
XCTFail()	Immediately and unconditionally fails a test case.

Table 3.1: All available assertion methods in Xcode[7].

[7] [Appr].

3.4 Asynchronous testing

Usually, the majority of code a developer writes is synchronous. This means that all the commands run in a defined order where the next command or task only executes when the previous one has finished. Synchronous code is deterministic. Once the input parameters for a piece of synchronous code are known, the sequence of commands is predefined. Although synchronous code might be easy to write and understand, the simplicity also comes with some drawbacks. Due to its purely sequential nature, synchronous code can be slow. If a longer task is executing, it can even leave the program unresponsive for a while[8]. To overcome these issues, almost all modern software makes use of asynchronous programming patterns. When programming asynchronously, multiple tasks can run concurrently in different threads, on different processors or even on different machines[9] and notify one another of their results if necessary. Asynchronous code on the same system but on different threads is mostly used in cases where longer running operations shall not disrupt the computations on the main thread, so that the application stays responsive. A common example of asynchronous operations on different systems are network calls to communicate with certain APIs.
Testing synchronous code doesn't need any special considerations. The SUT is simply executed with the desired parameters and the result or state can then be asserted. Synchronous processes guarantee that all code has run and all the computations are finished once the outcome of a test is verified with an assertion method. Running the same test under the same conditions will therefore always lead to the same outcome (assuming the code under test has not been changed). Most test cases are synchronous and can be built with the basic testing tools discussed previously.
Asynchronous testing, on the other hand, is more complicated. Whether the asynchronous operations run on the same or an external system is not crucial. Difficulties and challenges remain the same in both cases. The following explanations are therefore applicable to all asynchronous types of operations, even if only one is mentioned. The reason asynchronous testing is harder than synchronous testing is the timing aspect. At some point during an asynchronous test case, computations in the main thread rely on

[8][BC13], p. 1.
[9][BC13], p. 2.

operations that are executed in a different thread. Such operations could be a callback, a notification to a delegate object or simply some longer running code that has been outsourced to a different thread[10]. When writing an asynchronous test, developers expect these asynchronous operations to be performed during test execution, but cannot control the exact time they start or finish executing. When the result of a test case relies on computations from other threads, it is therefore hard to tell when the test outcome has to be verified. If assertion is performed just a little too early, the asynchronous operations might not yet be finished and the assertion would fail. If no further precautions are taken, the success or failure of a test case would not only depend on the correctness of the SUT, but also on the timing of its asynchronous operations which would lead to unstable test results. To deal with this problem, Apple introduced asynchronous testing functionality to their XCTest framework starting with Xcode 6[11]. This functionality will be discussed in the following section.

3.4.1 Testing asynchronous code with expectations

The way asynchronous tests are handled in Xcode is by using expectations. Expectations are a tool to serialize asynchronous operations in a test case by telling it that a certain asynchronous operation is expected to be performed at a certain point within the test case[12]. Whenever a test case expects an asynchronous operation to occur, the test is halted until this operation has been completed, whereupon the test case continues executing normally. This leads to test cases being predictable again as they wait for asynchronous code to finish which removes the previously mentioned timing problem. When an expectation cannot be fulfilled, meaning that the asynchronous operation did not occur as expected, the test case fails. Using expectations is the only scenario where a test case might not contain an assert statement. This can be the case when a test validates if an asynchronous operation has been performed rather than validating the result of an operation. Working with expectations is a process that can be divided into three steps: Creating

[10][Appn].
[11][Appq].
[12][Appq].

the expectations, waiting for the expectations to be fulfilled and fulfilling the expectations.

3.4.1.1 Creating expectations

Creating an expectation is as simple as calling a method of *XCTestCase* with the same name[13]:

```
func expectation(description: String) -> XCTestExpectation
```

This method takes a description string describing the purpose of the expectation and returns the expectation object which is of class *XCTestExpectation*. The value of the description string is used to differentiate between multiple expectations and has no relevance during execution of test cases.

3.4.1.2 Waiting for expectations

To halt a test case and wait for expectations to be fulfilled, *XCTestCase* provides another method called *wait()*[14]:

```
func wait(for expectations: [XCTestExpectation],
          timeout seconds: TimeInterval)
```

This method pauses the execution of a test case and waits until all expectations passed to the *expectations* array are fulfilled, before resuming the test execution. The *seconds* parameter takes the maximum number of seconds the method waits for the expectations' fulfillment. If not all expectations are fulfilled after the time has elapsed, the test case fails. Limiting the amount of time the method waits for the fulfillment of expectations is important. Otherwise, a test case could get stuck in the *wait()* method if one of the passed expectations is never fulfilled. There are two additional variations of the *wait()* method that can also be used. The first one checks whether the specified expectations are fulfilled in the correct order by setting an additional boolean parameter to true.

[13] [Apps].
[14] [Apps].

```
func wait(for expectations: [XCTestExpectation],
          timeout seconds: TimeInterval,
          enforceOrder enforceOrderOfFulfillment: Bool)
```

The second variation does not need an array of expectations to be passed since it is automatically checking for the fulfillment of all existing expectations that were instantiated in the test case. The optional *handler* parameter takes a completion handler that gets executed once all expectations are fulfilled.

```
func waitForExpectations(timeout: TimeInterval,
                         handler: XCWaitCompletionHandler? = nil)
```

Choosing the right value for the timeout parameter is of crucial importance. If the chosen value is too big, test execution is unnecessarily slowed down if an expectation does not get fulfilled. If the value is chosen too small, the test might fail by mistake because not enough time was allowed to wait for the asynchronous operation to complete.

3.4.1.3 Fulfilling expectations

A normal expectation of class *XCTestExpectation* does not get fulfilled automatically. The fulfillment needs to be performed by the asynchronous operation itself once it has finished. To do so, expectations need to be passed to the objects that execute the asynchronous code. The techniques of passing the expectations to these objects vary from case to case and can also involve the creation of test doubles. As soon as the asynchronous operation is finished, it retrieves the reference of the expectation it was passed and fulfills it by calling its *fulfill()* method[15].

```
expectation.fulfill()
```

An example for this process will be discussed in the next section.

[15][Apps].

3.4.1.4 Asynchronous testing example

Listing 3.2 shows how all of the parts discussed above are brought together to create an asynchronous test case. This test case simply evaluates whether a specified document within the project bundle can be opened successfully. It is structured into four parts:

1. An expectation is created.
2. The document that should be opened is localized in the bundle. A `UIDocument` instance is created that represents the document.
3. The document is opened by calling the asynchronous `open()` method of the `UIDocument` object. Additionally, a completion handler is passed that gets called when the process of opening the document has finished. The completion handler provides a boolean value that indicates whether the document was opened successfully or not. Within the completion handler, two tasks are executed. First, an assert method checks if the document was successfully opened. Then, the expectation gets fulfilled to tell the test case that the process of opening the document has finished.
4. The `waitForExpectations()` method is called. This method suspends the execution of the test case until the expectation is fullfilled (i.e., the document has been opened) or the timeout is hit. The `waitForExpectations()` method also takes an optional completion handler which closes the document again in this case.

```
func testDocumentOpening() {
    // 1. Create an expectation object. This test
    //    only has one, but it's possible to wait for
    //    multiple expectations.
    let openExpectation = expectation(description: "open doc")

    // 2. Retrieve a document to open
    let URL = Bundle(for: type(of: self)).url(forResource: "myDoc",
                                        withExtension: "txt")
    let doc = UIDocument(fileURL: URL!)

    // 3. Open document, pass expectation to completion handler
    doc.open { openSuccess in
        XCTAssert(openSuccess);
        openExpectation.fulfill()
    }

    // 4. The test will pause here until the timeout is hit
    //    or all expectations are fulfilled.
    waitForExpectations(timeout: 2) { error in
        doc.close(completionHandler: nil)
    }
}
```

Listing 3.2: Example of an asynchronous test case[16]

3.4.1.5 Expectations with predicates

Calling the fulfill method of an expectation is a very easy and efficient way to inform the test case that an asynchronous operation has finished. The downside of this approach is, that the expectation has to be passed to

[16] [Appq].

the asynchronous code. It might be simple to insert an expectation into a callback, a completion handler or a method of a test double because the developer has full control over this code in test cases. However, sometimes developers do not have direct control over or access to the asynchronous code within a test case. They therefore cannot pass the expectation to the asynchronous operation nor can they fulfill it from there. Even though asynchronous code might be inaccessible from within a test case, it still affects the execution of the tested code, for example by modifying the state of certain objects under test in the background. To wait for the fulfillment of asynchronous code that developers are not fully in control of, Apple provides a different kind of expectation that fulfills itself. It does so as soon as a certain logical condition is met rather than by calling the *fulfill()* method. Such conditions are called predicates and either evaluate to true or false. They are represented by the class *NSPredicate* in Swift and Objective-C. Predicate-based expectations are created as follows:

```
func expectation(for predicate: NSPredicate,
                 evaluatedWith object: Any?,
                 handler: XCTNSPredicateExpectation.Handler? = nil)
```

The *predicate* parameter takes the predicate that has to be evaluated, the *object* parameter takes an additional object that might be needed to evaluate the predicate and the optional *handler* parameter takes a block of additional evaluation code that is performed once the predicate evaluates to true. A simple example of a predicate based expectation can be seen in listing 3.3. This test case aims to bill a customer for some kind of service. The bill is created in a asynchronous background task which is not accessible from within the test case. To check if the bill creation was successful, an *NSPredicate* gets passed to the expectation. The predicate evaluates the number of bills saved in the customer object. Once the number of bills rises from zero to one, the bill has been created and the predicate returns true. The *waitForExpectations()* method periodically evaluates the predicate and checks its return value until it evaluates to true (or the timeout is reached). As soon as the predicate evaluates to true, the associated expectation is automatically fulfilled.
As the predicate is only evaluated once every second, it is important to consider this delay when specifying the timeout parameter of the *wait()* method.

```
func testBilling() {
        createBillForCustomer(customer)
        let billSuccessPredicate = NSPredicate { _, _ in
                                    customer.getBills().count == 1 }
        expectation(for: billSuccessPredicate,
                    evaluatedWith: nil,
                    handler: nil)
        waitForExpectations(timeout: 5, handler: nil)
    }
```

Listing 3.3: Example of an predicate based asynchronous test case

3.5 Third party testing tools for Xcode

Since the introduction of Apple's XCTest framework, different open source projects have emerged that provide additional functionality to enhance and improve Xcode's testing capabilities. This section discusses some of the most popular frameworks.

3.5.1 Isolation frameworks

As described in subsection 2.2.3 isolation frameworks are popular tools to create test doubles in a fast and easy way. Unfortunately, there is no widespread isolation framework available for Swift. This is due to the fact, that Swift only allows (very limited) read but no write reflection. Without the possibility of using write reflection and with Swift's additional strict type safety, the dynamic creation of test doubles at runtime is not possible[17]. Nevertheless, developers have come up with solutions to circumvent this problem. Objective-C on the other hand does provide read-write reflection and therefore isolation frameworks exist.

[17][Oro15].

OCMock *OCMock*[18] is the most popular open source isolation framework for Objective-C. It has been first released in 2004 and provides all basic tools to dynamically create test doubles. It's syntax and functionality is similar to Mockito. As Objective-C is no longer the main programming language for Apple devices, further development work on the framework has drastically decreased.

Cuckoo Although Swift does not allow write reflection, with a few tricks it is still possible to create test doubles in the same style as OCMock or Mockito. One of the few frameworks that does so is *Cuckoo*[19]. Instead of creating the test doubles at runtime, Cuckoo uses a compile-time generator that creates a file with actual swift code for each defined test double. It does so by using inheritance and protocols. Once the files for the test doubles are created, they can be used within test cases. Cuckoo's runtime engine will then handle the execution of the test double's fake behaviour once the tests are executed.

3.5.2 Other frameworks

Two other very popular open source testing frameworks for Xcode are Quick and Nimble. Although the frameworks belong to the same project, they serve two different purposes.

3.5.2.1 Quick

Quick[20] is a framework for behaviour driven development. As the title suggests, behaviour driven development is a process that intends to describe the behaviour of a system from a customer's or user's perspective. In this process, everyone involved in the development process shares a common language to talk about the system. The common language is used to specify all scenarios of how the system or subsystem is supposed to work. These

[18] [Dör].
[19] [Bri].
[20] [Quib].

scenarios are then turned into behaviour driven tests which are also specified in the common language. This approach improves the readability of tests (for engineers as well as for other project members) and lets the tests serve as a living documentation[21]. Behaviour driven tests are not much different from normal tests. The most notable difference is, that behaviour driven tests substitute the name of test methods with a description of the scenario they represent. Quick does this by using the *Given-When-Then* pattern. With this pattern, tests are specified by their initial state (Given), the action performed on the initial state (When) and the expected outcome (Then)[22]. Rewriting the test case seen in listing 3.3 using Quick, could look like the code seen in listing 3.4. Note that instead of *given*, *when* and *then*, Quick uses the words *describe*, *context* and *it*.

```
describe("Customer has not been billed yet") {
        context("A bill is produced for the customer") {
                it("The bill should be produced successfully") {
                        createBillForCustomer(customer)
                        let billSuccessPredicate = NSPredicate { ... }
                        expectation(for: billSuccessPredicate, ... )
                        waitForExpectations(timeout: 5, handler: nil)
                }
        }
}
```

Listing 3.4: A behaviour driven test with Quick

3.5.2.2 Nimble

Nimble[23] is a matcher framework that extends the assertion functionality of XCTest. Although XCTest offers various assertion methods, not all scenarios can be tested in a clean and readable way. XCTest for example does not have an assertion method to check that a string contains a particular substring.

[21] [WH12], p. 4.
[22] [Mis17], p. 319.
[23] [Quia].

Nimble on the other hand provides this (and much more) functionality and helps writing assertions that are not available in XCTest. Additionally, Nimble improves the readability of tests by changing the assertion syntax. This includes an easier syntax for writing asynchronous, predicate based tests. Listing 3.5 compares the assertion process of XCTest and Nimble by using the substring example mentioned above.

```
// Asserting that a string contains a substring with XCTest
XCTAssert("Hello World".rangeOfString("Hello") != nil)

// Asserting that a string contains a substring with Nimble
expect("HelloWorld").to(contain("Hello"))
```

Listing 3.5: Asserting with Nimble and XCTest

4 Pocket Code

The following chapters describe Pocket Code's audio features and investigate approaches for their automated testing. To understand the concepts discussed in these chapters, it is necessary to have some knowledge of the basic ideas and the terminology of Pocket Code. A short introduction to Pocket Code is therefore given below.

4.1 What is Pocket Code?

Pocket Code is an open source IDE for smartphones and tablets that enables its users to create games, animations and interactive stories in the visual programming language Catrobat[1]. It is designed to provide a simple and intuitive user experience, to offer teenagers and young adults an easy entry into the world of programming[2]. Pocket Code is inspired by Scratch[3], another visual programming language developed by the Lifelong Kindergarten group of the Michigan Institute of Technology.
Working with a system of interconnectable building blocks, Pocket Code imitates the principle of Lego bricks to create an intuitive programming language. Programming commands are visualized by bricks, that can be stacked on top of each other or nested inside of other bricks (as shown in figure 4.1). These bricks form the programming code that will be executed from top to bottom once a program is started.

[1] [Int].
[2] [Koi16], p. 4.
[3] [Masc].

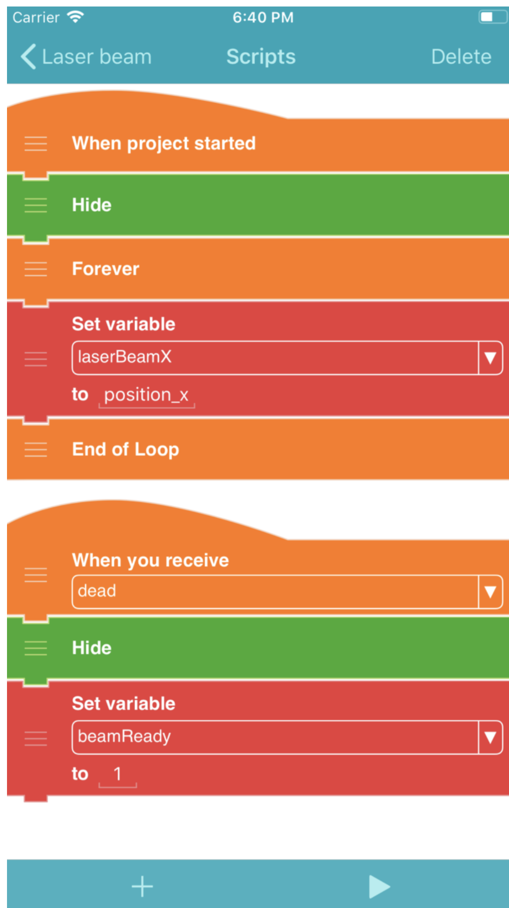

Figure 4.1: Pocket Code scripts and bricks

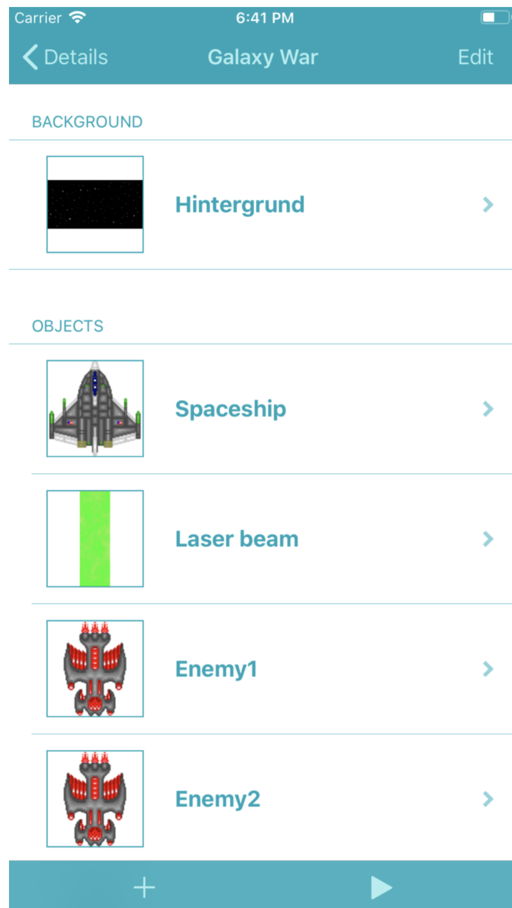

Figure 4.2: Objects inside of a project

4.2 Structure of a Pocket Code Project

Programs created with Pocket Code are organized in multiple structural levels that organize the code and its resources. A short overview of these levels is provided below.

4.2.1 Project level

A program created with Pocket Code is called a project. Almost every aspect of a project's lifecycle is handled within Pocket Code: It is programmed in Pocket Code, stored in Pocket Code as well as executed in Pocket Code. Additionally, a finished project can be uploaded to "Catrobat Share", a server used to share projects with other Pocket Code users.
A project serves as as a container that stores all resources needed for its execution. Inside of a project, the resources are organized into different groups called objects. These objects are the first things that can be seen when opening a project. Figure 4.2 shows a project called "Galaxy War". The screen lists all the objects which belong to this project.

4.2.2 Object level

Pocket Code objects represent visible elements of a project, for example a game character or an animated element that can be seen on screen. Each object contains executable code and additional resources (images and audio files) that it needs to fulfil its task in the context of a project. By touching objects on the screen in a running project, users can interact with them and start the execution of certain code stored within the object. Additionally, objects can exchange messages with each other which allows code execution in one object to be triggered by another object. A project can contain an arbitrary number of objects.
As visible in figure 4.2, there is one special object that exists in every project: the background object. Although not explicitly labelled as an object, its purpose is the same as a regular object. The only difference is, that it defines the appearance and behaviour of a project's background. Also, it often

contains code which is of a more general nature and not related to a specific object. An object itself consists of the three sections shown in figure 4.3: a scripts section, a looks section and a sounds section.

- The scripts section contains the executable code of an object.
- The looks section contains a list of images that define how the object appears on screen. An object can contain multiple looks, but only one can be active for every object at any given moment. The active look can be changed during project execution, for example to animate the movements of a game character or to change its outfit.
- The sounds section stores a number of sound files which can be played back by the object they reside in.

4.2.3 Script level

The executable code of an object is grouped into different scripts. Each script in turn consists of an arbitrary number of bricks that execute specific commands. A script always starts with a script brick, recognizable by the curved shape in figure 4.1. Script bricks are triggers that listen for certain events and start the execution of the script once this event occurs. Such events include the reception of a message from another object, a user touching an object on the screen or a user starting the execution of the project. When a script gets triggered, it executes all its associated bricks one after another.

4.2.4 Brick level

Executed bricks have a direct impact on the state of their associated object and the project itself. Every brick is assigned to one of five categories:

- Control bricks regulate the control flow of a project (if-else-statements, loops, etc.)
- Motion bricks influence the position and movement of objects on the screen.

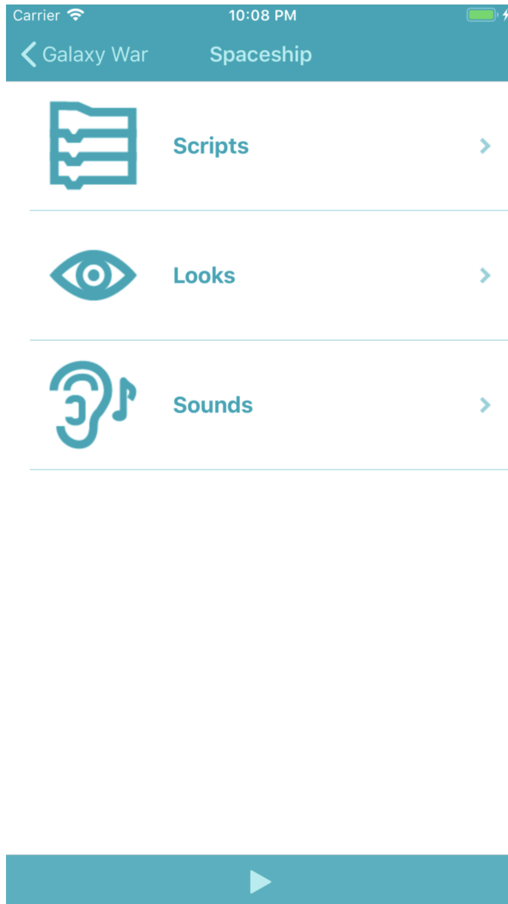

Figure 4.3: Scripts, looks, and sounds sections of an object.

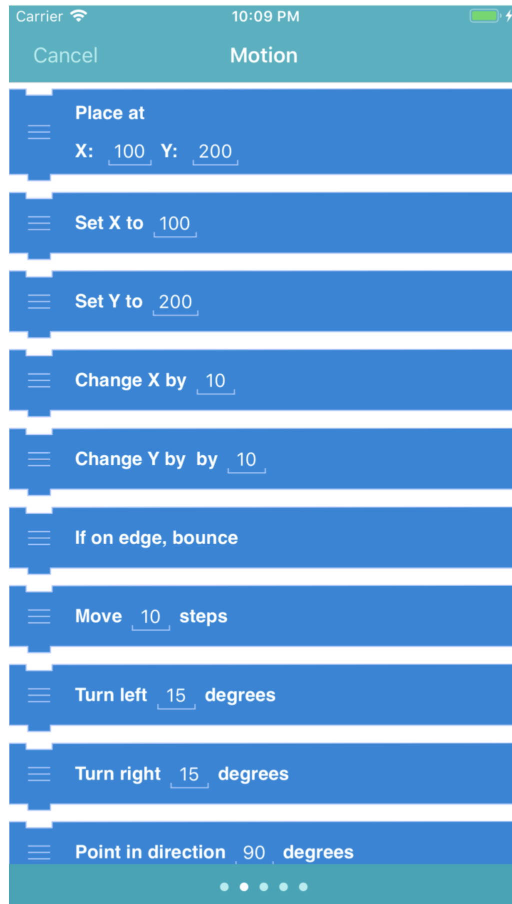

Figure 4.4: A brick selection menu listing all motion bricks

- Look bricks change the appearance of objects, by activating another look or changing the color or transparency of an object, etc.
- Sound bricks control the playback of the audio files mentioned in section 4.2.2 and perform other audio related tasks.
- Variable bricks maintain and modify variables and arrays that are used during project execution.

The behaviour of most bricks can be adjusted by a number of parameters. A motion brick for example might need two parameters to define the new vertical and horizontal positioning of an object while a variable brick might need a variable name and a new value to alter the value of the desired variable. Figure 4.4 shows a brick selection menu (or more specifically, the brick selection menu for motion bricks) that opens when a user wants to add a new brick to a script. Most bricks have at least one adjustable parameter, highlighted by an underscore.

As seen in this chapter, Pocket Code uses the terms *project*, *object*, *sound* and *look* to describe parts of its own functionality and structure. These terms are ambiguous and could also be used in other contexts. To avoid confusion, these terms are abbreviated in the remainder of this thesis whenever they relate to their meaning in the context of Pocket Code and not to an alternative meaning. The abbreviations are PCProject (Pocket Code Project), PCObject (Pocket Code Object), PCSound (Pocket Code Sound) and PCLook (Pocket Code Look). Although the terms *script* and *brick* are also ambiguous, they are only used in the context of Pocket Code in this thesis. Those terms are therefore not abbreviated.

5 Audio processing on macOS and iOS

The aim of this thesis is to extend, improve and test the audio features of Pocket Code. The following chapters will therefore describe this process by analysing Pocket Code's original audio engine, as well as the redesigned version. To understand the concepts of these discussions, a short introduction to the audio processing frameworks of macOS and iOS will be given in this chapter.

Apple's operating systems offer multiple audio processing frameworks and APIs which can be divided into three different groups based on their functionality and abstraction levels[1]. The following sections provide an overview of those three groups and introduce an additional open source audio processing framework.

5.1 Core Audio

Core Audio is a collection of various frameworks written in the programming language C which, in their entirety, represent the main audio infrastructure on iOS and macOS. It is responsible for all sounds played and recorded on these operating systems[2]. Core Audio's features are extensive and manifold and can be divided into two categories: frameworks that produce and process audio streams, and frameworks that facilitate the production or processing of said streams. Figure 5.1 shows the numerous services Core Audio provides. Included are services to read and write audio

[1] [App14b].
[2] [AA12], p. 13.

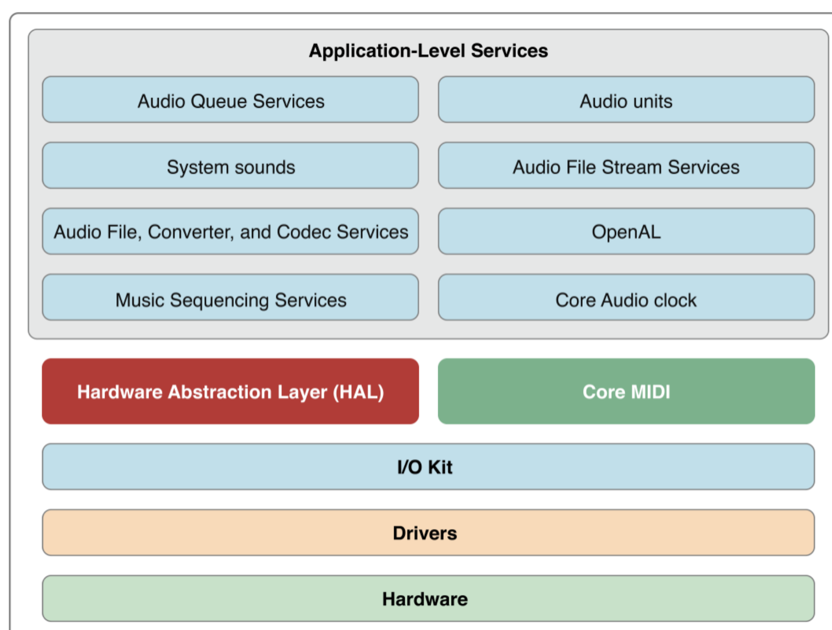

Figure 5.1: The architecture of Core Audio[3]

files, convert audio formats, play virtual instruments, spatialize sounds or interact with the audio hardware of a phone, tablet or computer.

Core Audio provides developers all tools necessary to create complex, low latency and real-time audio applications. Unfortunately, the Core Audio APIs are very low-level and do not hide many of the difficult processes involved in digital audio processing. Therefore, extensive knowledge of the subject is essential. Additionally, Core Audio's procedural C code adds another layer of difficulty. Using Core Audio directly for the development of audio features is only advisable if the higher level APIs discussed in the following sections do not meet the requirements.

[3] [Appp].

5.2 High-level audio classes

To simplify the development of audio features, Apple added several easy to use, high-level audio classes to their *AVFoundation* framework, a framework for handling audiovisual media. The most notable classes are `AVAudioPlayer`, `AVAudioRecorder` and `AVSpeechSynthesizer`. `AVAudioPlayer` and `AVAudioRecorder` do exactly what their names imply: playing audio data from a file or memory[4], and recording the microphone's audio input[5]. `AVSpeechSynthesizer` is a class that provides text to speech (TTS) functionality. A TTS system takes an arbitrary string as input and creates a synthesized speech output of the given text[6].
Although still relying on its functionality in the background, all these classes hide Core Audio's complex infrastructure from developers. While this makes it significantly easier to implement basic audio requirements for an application, the simplicity also has its drawbacks. `AVAudioPlayer`, `AVAudioRecorder` and `AVSpeechSynthesizer` perform their tasks isolated from other potential audio processing elements in an application and cannot be integrated into a bigger audio processing context. Further processing and manipulation of the audio data is therefore not possible.

5.3 AVAudioEngine

To overcome the gap between the simple but limited and the powerful but complex programming interfaces, Apple introduced a new way of audio processing in a presentation at their 2014 developers conference. The information given in this section is largely based on the contents of this presentation available at [App14a] and [App14b]. The core of the newly introduced features, which are also part of the AVFoundation framework, is a class called `AVAudioEngine`. Although `AVAudioEngine` is only one of many new classes released as part of a whole toolkit, the toolkit itself

---

[4] [Appd].
[5] [Appf].
[6] [Apph].

is commonly referred to as AVAudioEngine. The two meanings are distinguished by using the typewriter font (with which all class, method and variables names are written in this thesis) to refer to the `AVAudioEngine` class, and the normal font to refer to the AVAudioEngine toolkit.
Internally, AVAudioEngine makes use of the existing Core Audio APIs and was designed to be simple but still powerful enough to implement complex tasks[7]. The goal of AVAudioEngine was to build a modular audio processing system. Each module fulfills its specific task such as playing an audio file, recording audio data, or modifying audio data that is currently being played. The modules are then connected to a bigger structure that processes the audio data in the desired way. In the context of AVAudioEngine, modules are called nodes. The structure of connected nodes is represented by the `AVAudioEngine` class itself and is also referred to as a graph. [Appa] describes the `AVAudioEngine` class as "a group of connected audio node objects used to generate and process audio signals and perform audio input and output".
Every node is based on the `AVAudioNode` class and has a number of input buses supplying audio data to the node, and a number of output buses sending the generated or processed audio data to the next node in the graph. Each bus has an audio format assigned to it, which specifies the number of channels and the sampling rate of this bus (among other format parameters)[8]. Pocket Code does not need more than two channels per bus as it only processes mono or stereo audio data.
There are three different types of nodes that are relevant for building a graph with AVAudioEngine: source nodes, processing nodes and destination nodes.

- Source nodes provide new audio data to the audio engine. The most common source nodes are `AVAudioPlayerNode`, `AVAudioInputNode` and `AVAudioMIDIInstrument`. `AVAudioPlayerNode` feeds audio buffers from an audio file to the audio engine[9], while `AVAudioInputNode` connects to the system's audio input (usually the device's microphone) and obtains the audio data from there. `AVAudioUnitMIDIInstrument` is a superclass for software instruments

[7] [App14b].
[8] [Appc].
[9] [Appe].

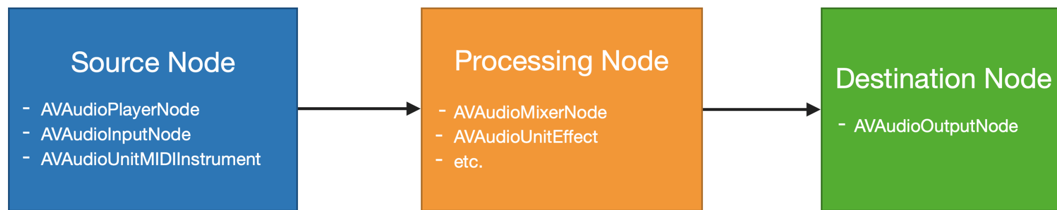

Figure 5.2: AVAudioEngine processing chain with source, processing and destination node.

that produce the audio data themselves, like a software synthesizer or sampler[10].

- Processing nodes take the audio data from the source nodes, and modify it. One of the most frequently used processor nodes is the `AVAudioMixerNode` which accepts multiple input buses and mixes them to a single output bus[11] (usually a stereo output bus). Other examples of processing nodes are audio effects like delay, reverb or distortion. The nodes responsible of applying these effects all inherit from `AVAudioUnitEffect`.
- Finally, a destination node takes the processed audio data and sends it to a hardware output (speaker). The only available destination node is `AVAudioOutputNode`, which sends the audio data to the system's default audio output. An `AVAudioEngine` instance can only have one single output node which is implicitly created when instantiating the engine. It can be accessed by the `outputNode` property of the audio engine.

Multiple nodes that are linked together are called a processing chain. An active, functional chain always starts with a source node and ends with a destination node. In between, there is an arbitrary number of processing nodes. A processing chain that does not meet this requirement, or a chain that is not fully connected, is inactive and will not be able to provide any audio data to the device's audio output. Figure 5.2 shows a fully connected chain from a source node to a destination node and lists some of the possible classes in each category. This chain already represents a full processing graph ready to be put into operation by the AVAudioEngine. For more complex audio applications, a processing graph usually consists of more

[10] [Appg].
[11] [Appc].

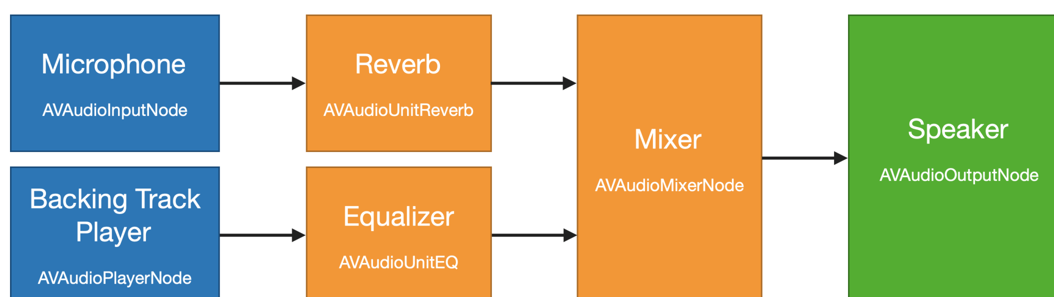

Figure 5.3: A processing graph consisting of two processing chains.

than just one processing chain. Figure 5.3 shows an example of a more complex graph. It represents a karaoke application that plays a backing track, captures the device's microphone input and sends both signals to the device's speakers[12]. The backing track can be modified by an equalizer, and the voice can be enhanced with a reverb effect. This graph consists of two processing chains, one going from the microphone to the speaker, the other one going from the backing track player to the speaker. The chains meet at the mixer node.

The process to create and run an audio engine is as follows[13]:

1. Create an instance of `AVAudioEngine`.
2. Instantiate as many nodes as needed.
3. Attach all nodes to the audio engine.
4. Connect the nodes as needed to build an audio processing graph.
5. Start the audio engine.

Once the audio engine has been started, an active render thread is created. In this thread, audio buffers are pushed from the source nodes into their respective processing chain, processed by all processing nodes in that chain, and then pulled by the destination node.
A great feature of AVAudioEngine is, that it can be dynamically reconfigured during runtime. Nodes can be added, removed, adapted or newly connected and the changes will be audible immediately[14]. This proves particularly

[12][App14a], p. 10.
[13][App14a], p. 43 - 50.
[14][App14b].

useful for applications where the audio requirements change in a non-deterministic fashion based on how users interact with the application. A multiplayer game for example might have to play movement noises (footsteps) for every player that is within earshot of one's own character. AVAudioEngine can simply do that by dynamically adding or removing audio players and adapting their volumes based on the number and distance of other players in the game. The dynamic reconfiguration of the audio engine is also useful for Pocket Code, where the audio output of a PCProject can heavily depend on user interaction.

5.4 AudioKit

In addition to the broad choice of audio frameworks and APIs maintained and developed by Apple, an external framework called *AudioKit*[15] has gained popularity in recent years. AudioKit is an open source audio framework for iOS, macOS and tvOS which supports many features such as audio processing, audio synthesis and audio analysis. It was built to be easy to learn and use while still offering maximum flexibility and powerful tools. At first glance, AudioKit looks like a copy of AVAudioEngine as it also builds audio processing graphs using a similar syntax. In fact, AudioKit builds on top of the existing Apple frameworks and uses AVAudioEngine in the background to create its own audio processing graphs. On closer inspection, it becomes clear that AudioKit provides far more features than AVAudioEngine. While AVAudioEngine includes about 20 different source or processing nodes, AudioKit includes more than 150. Many of the additional nodes are synthesizers and audio effects, using digital signal processing algorithms from other open source libraries like *Soundpipe*[16] or *Synthesis ToolKit*[17].
To facilitate the addition of this large range of new nodes, AudioKit has to interact with Core Audio and use several of its services. Luckily, the interaction with Core Audio is hidden from AudioKit's end users (unless they are implementing their own custom nodes) which makes the creation

[15] [Proa].
[16] [Bat].
[17] [CS].

of audio processing graphs with AudioKit more powerful but just as easy as with AVAudioEngine.
Apart from extending the functionality of AVAudioEngine, AudioKit offers many other features including UI components to visualize audio waveforms or audio analysis tools performing tasks like pitch analysis, loudness analysis or Fast Fourier Transform (FFT).

6 Original Pocket Code iOS audio engine

This chapter discusses the initial state of Pocket Code's audio engine before its redesign that has been carried out during the course of this thesis. In combination with the discussion of the redesigned audio engine in chapter 7 and the new testing strategies in chapter 9, detailed insight is given on how the audio engine and the testing of the audio engine have evolved and improved. The chapter is divided into four sections which describe the existing audio features, the audio engine's architecture, the problems and limitations of the implementation and the existing automated tests.

6.1 Overview of Pocket Codes's audio features

A lot of Pocket Code's functionality is based on the functional principles of Scratch. It is therefore expected, that Pocket Code matches Scratch's behaviour for features that both applications share. This also applies to Pocket Code's audio features of which many have been directly adopted from Scratch and therefore should also have the same behaviour whenever possible. For features that only exist in Pocket Code but not in Scratch, Pocket Code's iOS version should match the behaviour of its Android equivalent. Considering these aspects, the following description of the audio features in Pocket Code always specifies the desired behaviour. Deviations from the desired behaviour in the actual implementation are pointed out in section 6.3.

Before the redesign of the audio engine, Pocket Code's audio features

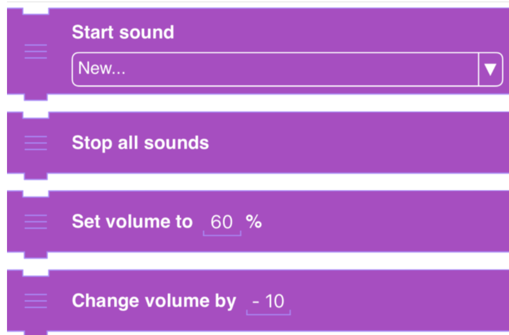

Figure 6.1: Sound bricks in Pocket Code.

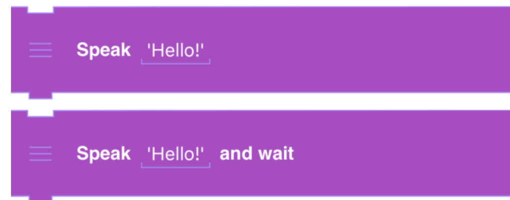

Figure 6.2: Speech bricks in Pocket Code.

mainly consisted of six different bricks. As shown in figure 6.1 and figure 6.2 these bricks can be divided into two groups: bricks that control the playback of PCSounds, and bricks that control the playback of speech. A short description of these bricks and their desired behaviour is given below. Note that the numbers in square brackets given in the brick names, are placeholders for parameters that can be chosen by users for each brick. More information about the parameters (the parameter name, value range and value type) is provided in a separate table for each brick.

6.1.1 Sound bricks

Start sound [1] The *Start sound* brick is used to play an audio file. This file can either be recorded in advance with the microphone of the mobile device or downloaded from an external Pocket Code media library. The execution of the script's subsequent bricks continues immediately after the playback of the PCSound has been initiated. If multiple *Start sound* bricks are executed at the same time, the triggered PCSounds are played simultaneously, with a few rules that apply:

- Different PCSounds can always be played simultaneously.
- A PCObject cannot play the same PCSound multiple times simultaneously. If a PCObject triggers a currently playing PCSound for a second time, the playback will stop and restart from the beginning.

- Simultaneous playback of the same sounds can only be achieved by adding multiple instances of the same audio file to a PCProject. To simultaneously play two instances of the same sound, the respective audio file has to be added to the PCProject twice. Although such duplicated audio files sound exactly the same, they are still considered to be two individual PCSounds.

Parameter	Range	Type
1 Sound	Arbitrary PCSound	PCSound selection menu

Table 6.1: Parameters of the *Start sound* brick

Stop all sounds This brick immediately stops the playback of every PC-Sound currently played by the PCProject.

Set volume to [1] This brick sets the sound volume to a desired value between 0 and 100 percent. The change is applied at the PCObject level, which means that the new volume is only valid for PCSounds played by the same PCObject that executed the *Set volume to* brick. The volume change takes immediate effect on currently playing PCSounds as well as PCSounds being played in the future.

Parameter	Range	Type
1 Volume	0 to 100	Decimal

Table 6.2: Parameters of the *Set volume to* brick

Change volume by [1] The *Change volume by* brick behaves the same as the *Set volume to* brick with only one difference: Instead of setting an absolute volume value, a relative volume change based on the previously defined volume is performed. In the same manner as for the *Set volume to* brick, the change is applied at the PCObject level.

Parameter	Range	Type
1 Volume change	-100 to 100	Decimal

Table 6.3: Parameters of the *Change volume by* brick

6.1.2 Speech bricks

Speak [1] The *Speak* brick provides TTS functionality and takes a string parameter specifying the text to be spoken. When executed, this text is sent to a speech synthesizer which then immediately starts speaking the text. The execution of the script's subsequent bricks continues directly after the text has been forwarded to the speech synthesizer. Scratch's reference implementation does not restrict the number of texts that can be spoken simultaneously by a PCObject and sets a fixed volume for the spoken text. The speech bricks are therefore not affected by *Set volume to* and *Change volume by* bricks.

Parameter	Range	Type
1 Text	Arbitrary text	String

Table 6.4: Parameters of the *Speak* brick

Speak [1] and wait This brick is a variation of the *Speak* brick. As opposed to the standard *Speak* brick, it waits until the speech synthesizer has finished speaking the text before subsequent bricks are executed. Scratch only has one brick that provides TTS functionality. It is called *Speak* but is equivalent to Pocket Code's *Speak and wait* brick.

Parameter	Range	Type
1 Text	Arbitrary text	String

Table 6.5: Parameters of the *Speak and wait* brick

6.1.3 Other audio features

A number of other small audio features exist within Pocket Code, the most notable being the loudness sensor. This sensor continuously captures the microphone input during execution of the program, and evaluates the loudness of the signal. This loudness value can then be used as a parameter for any brick that takes a numeric parameter. Other features include the media library from which a user can download new PCSounds or the Pocket Code recorder used to record custom PCSounds. All these additional features operate isolated from each other and are not directly related to the main audio engine. They are therefore not dealt with any further in the course of this thesis.

6.2 Audio engine architecture

Pocket Code's original audio engine consists of a number of different classes, components and participants which all work together to provide its overall functionality. This section intends to give insight into the audio engine from a developer's point of view, describing the individual components and explaining how they are orchestrated to perform the engine's desired behaviour. A visualization of the engine's architecture can be seen in figure 6.3.

6.2.1 Audio engine components

Audio player To play PCSounds triggered by *Start sound* bricks, Pocket Code uses objects of type `AVAudioPlayer`, which offer an easy and high-level interface for audio playback (as described in section 5.2). Among others, `AVAudioPlayer` provides easy to use methods to play, stop, pause and resume sounds and to set their volume[1].

[1] [Appd].

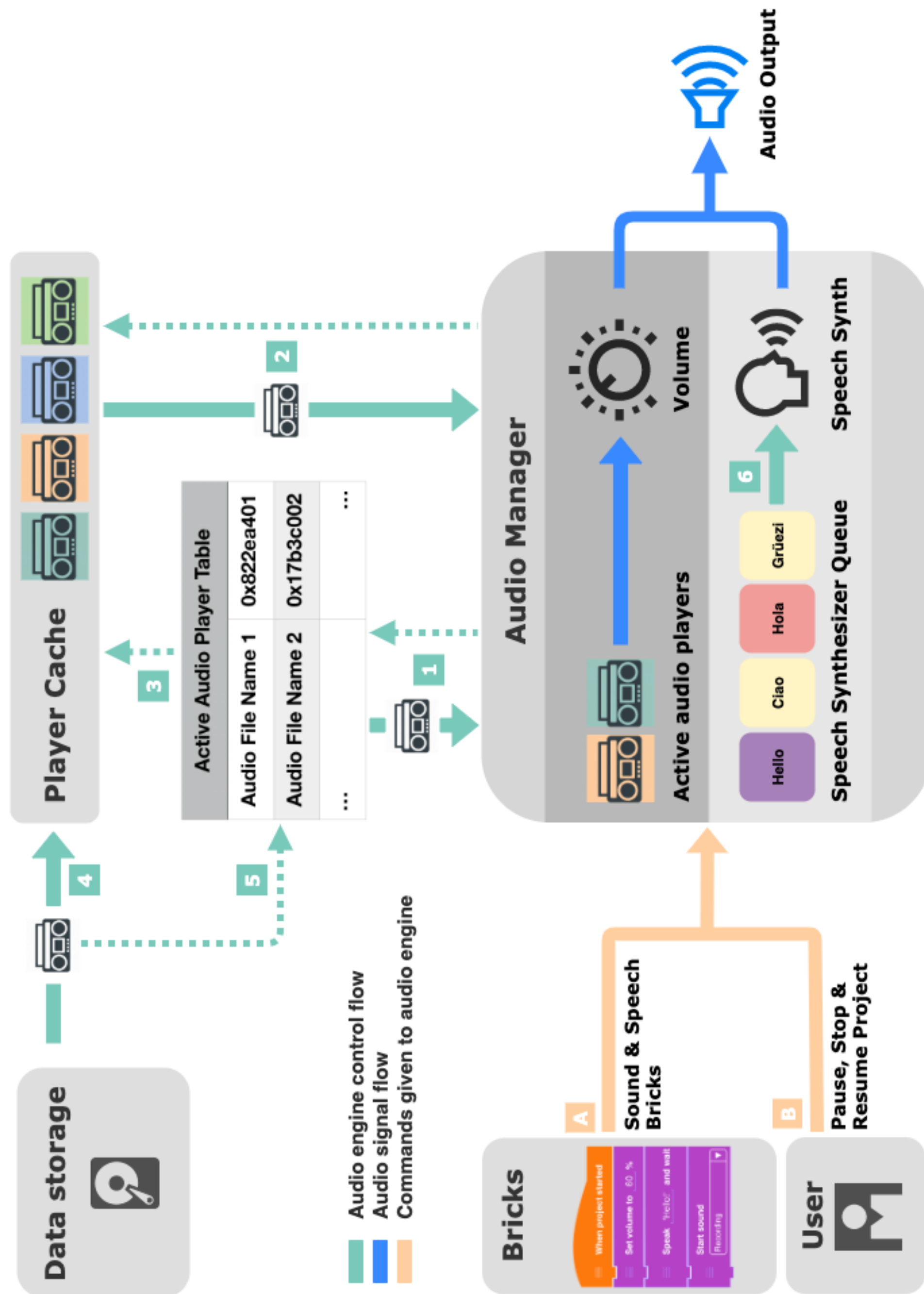

Figure 6.3: Visualization of the original audio engine

Audio player cache Audio data can use quite a lot of memory, which makes audio players expensive to create. It is therefore advantageous for the program's performance to store previously created audio players for later use instead of newly instantiating them every time a PCSound needs to be played. For this purpose, the audio engine maintains an audio player cache where every audio player is stored after its creation. The player cache is based on Apples `NSCache` class which is used to temporarily store key-value pairs. If the device runs out of memory, `NSCache` applies several auto-eviction policies which will delete some of the audio players from the cache. Therefore an `NSCache` should only contain objects that are not critical for the correct behaviour of an application[2]. Since an audio player can always be instantiated again, the use of `NSCache` is appropriate in this case. The keys used to store audio players are the names of the audio files the players have been initialized with.

Active audio player table The active audio player table is a table maintained by the audio engine that stores a reference of all audio players that are currently playing. Similar to the audio player cache, the table additionally stores the name of each player's audio file.

AVSpeechSynthesizer To provide the TTS functionality needed for the speech bricks, Pocket Code makes use of the `AVSpeechSynthesizer` class mentioned in section 5.2. In order for `AVSpeechSynthesizer` to speak a text, an object of type `AVSpeechUtterance` has to be instantiated first. This object contains a string with the text to be spoken together with various parameters that control the speech output (voice, pitch, speed, etc.)[3]. The `AVSpeechUtterance` object is then handed over to the `speak()` method of the speech synthesizer (arrow 6 in figure 6.3) which finally converts the text into an audio output. Although multiple speech synthesizers can be instantiated, `AVSpeechSynthesizer` does not support simultaneous playback of multiple utterances. If a speech synthesizer is already speaking, all attempts from other speech synthesizers to speak at the same time will

[2] [Appj].
[3] [Apph].

be ignored[4]. On the other hand, if the same speech synthesizer is prompted to speak another utterance while it is already speaking, the new utterance will be added to a queue. The utterances in the queue will then be spoken one after another. Like `AVAudioPlayer`, `AVSpeechSynthesizer` also provides easy to use methods to start, stop, pause and and resume the playback of spoken text.

6.2.2 Audio manager

The `AudioManager` class is the central class for sound or speech playback in Pocket Code's original audio engine. It manages the lifecycle of objects and components that are part of the audio engine, orchestrates their interactions, and is responsible to ensure the audio engine's correct behaviour according to the brick's specifications.
The audio manager's tasks can be divided into several categories that are discussed below. Commands to perform those tasks are either given by sound and speech bricks that are executed by PCObjects or users that are pausing, stopping or resuming a PCProject and consequently also the audio engine.

Playing PCSounds The audio manager's main task is the creation and maintenance of the structure used to play PCSounds. A PCSound's playback is always initiated by a PCObject executing a *Start sound* brick. Arrow A in figure 6.3 shows how the command to play a PCSound is given to the audio manager when executing a *Start sound* brick.
After a command to play a PCSound was given, the audio manager needs to retrieve or create an `AVAudioPlayer` that is responsible to play the desired PCSound. Audio players are created dynamically as soon as a PCSound needs to be played and are then stored for later use. They therefore do not exist immediately after starting a PCProject. The audio manager assigns a separate audio player to every PCSound that is used within a PCProject. This player then exclusively controls the playback of its assigned PCSound.

[4][Tho18].

At no time may a PCSound be assigned to multiple players. To create or retrieve an audio player, the audio manager uses the following process:

- Before instantiating a new audio player, the audio manager first checks if an audio player has already been created for the desired PCSound. It does so by querying the active audio player table (arrow 1 in figure 6.3). If the table contains an entry with the name of the desired PCSound, the audio manager reads the associated audio player reference and retrieves the player from its memory location.
- If no audio player with the desired PCSound could be found in the active audio player table, the audio manager repeats the search in the audio player cache (arrow 2). However, the active audio player table from the previous step also points to players stored in the player cache (arrow 3). This means, that carrying out step one is obsolete, since both step one and two directly or indirectly query the player cache to retrieve an audio player.
- If neither the reference table nor the player cache contain an audio player for the desired PCSound, a new audio player has to be instantiated. The audio file is loaded from the device storage into a new audio player which is then stored in the audio player cache (arrow 4) and referenced in the active audio player table (arrow 5).
- Finally, after retrieving or creating an audio player, the audio manager can carry out the command received by the *Start sound* brick by calling the `play()` method of the audio player.

Setting the audio players' volume The audio manager also takes care of adapting the audio players' volume. A change in volume initiated by a *Set volume to* or *Change volume by* brick (arrow A) has a global effect on all audio players of every PCObject (visualized by a single volume knob for all active audio players in figure 6.3). First, the audio manager adapts the volume of currently playing audio players by looping over the active audio player table and adjusting the volume property of every `AVAudioPlayer` object the table contains. The change in volume is audible immediately. To ensure that non-active and newly instantiated audio players are also assigned the correct volume for their next or first use, the volume property is also updated with every execution of a *Start sound* brick.

Speaking text As mentioned in section 6.2.1, *AVSpeechSynthesizer* does not allow multiple instances to speak at the same time. Due to this, the audio manager only uses a single speech synthesizer instance which is used by all PCObjects. The way the audio manager converts text to speech is straightforward. During the execution of a speech brick (arrow A), an utterance with the spoken text is given to the speech synthesizer's *speak()* method. The rest is taken care of directly by the speech synthesizer without any further involvement of the audio manager. The speech synthesizer immediately starts speaking the utterance or, if already speaking, places it in the utterance queue.

Pausing, resuming and stopping the audio engine When a user decides to pause, resume or stop a PCProject (arrow B), the same action has to be applied to the audio engine as well. Otherwise, the audio engine would continue playing sound although the PCProject has been paused or stopped. As both *AVAudioPlayer* and *AVSpeechSynthesizer* provide methods to pause, resume and stop their playback, Pocket Code simply calls these methods on the speech synthesizer as well as on every audio player in the active audio player table. When stopping a PCProject, the audio manager additionally removes all entries from the active audio player table.

6.3 Problems and limitations

While studying the original audio engine's behaviour and comparing it to the specifications, it became evident, that it was affected by many issues and flaws. Some of these issues were of a permanent, structural nature and occurred either because the reference behaviour of Scratch or Pocket Code Android was not studied carefully enough, or because the used iOS frameworks do not allow the desired behaviour. In other cases, the flaws were most likely introduced by recent code changes and remained undetected. A non-exhaustive list of encountered problems is given below.

Audio manager

- The audio manager is coordinating an accumulation of different individual audio processing components which cannot be interconnected to form a higher-level audio processing structure. The speech synthesizer and all audio players work independently from each other, directly sending their audio signals to the audio output of the operating system. The generated audio data cannot be used for further processing nor can it be accessed, analysed or mixed with other audio signals. This makes it impossible to extend the audio engine with features that rely on further processing of generated audio signals. Furthermore, it is not possible to implement testing strategies that verify the audio engine's behaviour based on its audio output if the audio signals are not accessible.
- When a running PCProject is paused or stopped by a user, the audio engine keeps playing. Although the audio manager should pause or stop the audio engine accordingly, the speech synthesizer and audio players can still be heard after ending or pausing the PCProject's execution.

Audio players and sound bricks

- When a PCObject triggers the playback of a PCSound that it is already playing, the PCSound should stop and restart playing from the beginning again. Instead, the command to start the playback from the beginning is simply ignored and the PCSound keeps on playing until it reaches the end.
- All audio players are assigned the same volume. This does not match the behaviour of Scratch and Pocket Code Android, where the volume is controlled on the PCObject level.
- The specification limits the volume of audio players to a range of 0% to 100%. If a user enters a value outside of this range in a *Set volume to* brick, or if a *Change volume by* brick modifies the volume so that it would fall outside of this range, the volume should be set to the upper or lower bound instead. However, the bounds are not checked when setting a new volume so that values over 100% or even negative values can occur. `AVAudioPlayer` seems to treat negative values as if they

were positive. Therefore, an audio player with a volume of -80% is just as loud as an audio player with a volume of 80%, which leads to unexpected and faulty audio output.

Speech synthesizer and speech bricks

- Only one speech synthesizer can speak at the same time on iOS. Therefore, all utterances are placed in a queue of a single speech synthesizer and are then spoken successively, although they might have been triggered at almost the same time. As highlighted in section 6.1.2, this is not in accordance with Scratch and Pocket Code's Android version, which allow multiple texts to be spoken simultaneously. The utterance queue leads to situations where the spoken text is delayed and out of sync with the current state of the executed PCProject.
- The *Speak and wait* brick is not working properly. All bricks that are placed after a *Speak and wait* brick in a script are completely ignored.

Audio player cache

- Importing the same audio file into different PCObjects creates multiple PCSounds that sound identical. According to the specifications, it should be possible to play multiple PCSounds simultaneously, even if they are identical. However, the audio player cache uses the audio file name as key. All PCSounds that are loaded from identically named audio files will therefore all wrongly share a single audio player in the audio player cache instead of creating an individual audio player for each PCSound. Therefore, when triggering such duplicated PCSounds around the same time, only the instance that was triggered by the first executed *Start sound* brick is audible. While this instance is playing, all *Start sound* bricks that trigger PCSounds stemming from identically named audio files will be ignored.
- Maintaining the active audio player list is unnecessary since all audio players are already stored in the audio player cache. Using both the audio player cache and the active audio player table to retrieve previously created audio players as described in section 6.2.2 leads to convoluted and error-prone code.

6.4 Automated testing

Despite the many specifications Pocket Code's audio engine should adhere to and the many problems and bugs that repeatedly occurred, only few audio related unit tests were available in Pocket Code before writing this thesis. Two of those tests assure that the audio manager successfully initializes new audio players when prompted to play a specific PCSound. By checking the boolean return value of *`AVAudioPlayer`*'s *`play()`* method, the tests also check if the audio player correctly initiates the playback of the PCSound.
A second batch of unit tests verifies the behaviour of the loudness sensor. The tests evaluate if the sensor is able to correctly retrieve and convert the values from a data source stub. As using real sensor data would significantly complicate the testing process, working with a test double is a legitimate and useful alternative.

The small number of implemented tests and the many bugs and weaknesses found in Pocket Code's original audio engine shows the need for several improvements:

- **Improving the unit test coverage:** Unit tests are a great tool to gain confidence in code during development and increase product quality. With an appropriate set of unit tests that validates the internal state and logic of the audio manager and other participating classes, many of the encountered bugs could have been detected and fixed immediately. Instead, many of them remained in Pocket Code over months or even years.
- **Building a living documentation:** Some of the audio engine's bugs have been introduced because the audio engine's specifications were not studied carefully enough during implemention or refactoring. This shows the importance of building a living documentation in the form of automated tests. When refactoring some unfamiliar code, it can be much easier to study its intended behaviour from easily readable tests than from complex production code. By using behaviour driven patterns (for example by using the Quick and Nimble frameworks discussed in section 3.5.2) the readability and information value of test cases can even be further improved.

- **Including integration tests:** The few available automated tests are simple unit tests and only validate isolated method calls. While unit testing is an important part of automated testing, some of the discussed bugs are caused by the way the audio engine's components work together and can only be observed when testing bigger parts of the audio engine's structure as a whole. As an example, validating if two PCSounds which were created from the same audio file can be played simultaneously, is only possible when testing multiple audio players together in the context of the audio manager. Other bugs only appear when whole scripts are executed. Test cases validating such behaviour therefore need additional, non-audio related components to be included. Testing whether a *Speak and wait* brick correctly halts the execution of a script until the text finished speaking for example is only possible by setting up a real script and incorporating the brick scheduling logic as well as the audio manager into the test case. Writing useful integration tests for the audio engine is therefore of vital importance.

- **Direct validation of audio output:** Unit and integration tests always validate a result, inner state or logic of a process or operation. In case of the previously mentioned unit test that checks if an audio player correctly initiates the playback of a PCSound, the validated end result is a boolean value returned by the `AVAudioPlayer`'s `play()` method. This boolean value can be seen as an indicator, telling whether the audio player is playing a PCSound correctly or not. In many cases, especially when dealing with unit tests, validating such abstract indicator data is a quick and easy way to presume correct behaviour of the engine. However, the indicator data cannot guarantee with absolute certainty that the SUT is working properly, as the actual result that should be verified is the audio output of the audio engine.
  Some of the audio engine's specifications describe how different PCObjects, audio bricks or audio players interact with and influence each other. Testing such scenarios with the aid of indicator data increases the complexity, as the result that has to be verified is a sequence of events. This means that timing is introduced as an additional component that has to be considered during validation. An integration test for a *Speak and wait* brick for example, would include at least the

following checks:

- Checking if the speech synthesizer immediately starts speaking after execution of the speech brick.
- Checking if the script pauses for the right amount of time until the text has been spoken completely.
- Checking if the subsequent brick (for example a *Start sound* brick) is executed once the speech synthesizer has finished speaking.

Verifying all this behaviour by checking internal states or intermediate results of the audio engine can be very difficult, as the data that has to be verified is located in different components of the SUT and needs to be evaluated at specific times during test execution. To simplify the test procedure for such scenarios, it would be interesting to evaluate possible techniques to directly record and validate the engine's audio output instead of solely relying on indicator data. Directly validating the audio output can also be useful when indicator data can be easily verified but not fully trusted. This might be the case when the underlying audio frameworks have known problems or anomalies that have not yet been fixed.

7 New Pocket Code audio engine

The new Pocket Code audio engine must be built to meet a variety of requirements:

- Existing weaknesses described in section 6.3 have to be remedied.
- The audio engine has to incorporate a set of new features that already exist in Scratch.
- Future additions of new audio features to Pocket Code should not require a complete refactoring of the audio engine. Instead, the audio engine should be modular and easy to expand.
- The audio engine has to be testable in order to immediately detect problems as soon as they emerge. For the reasons mentioned in section 6.4, test cases should also have access to the engine's audio output. This ensures that, if necessary, the generated audio data can directly be validated without solely relying on the audio engine's inner states and intermediate results during test execution.

With these requirements in mind, all audio frameworks discussed in chapter 5 need to be evaluated for their suitability. However, to make an informed decision, the specifications of the audio engine's new features have to be analysed first.

7.1 New audio features

The new audio engine was designed to incorporate a total of ten new bricks, which can be divided into three different categories: sound bricks, effect bricks and music bricks.

7.1.1 Sound bricks

Start sound [1] and wait The *Start sound and wait* brick is very similar to the *Start sound* brick. All behaviour discussed in section 6.1.1 also applies to the *Start sound and wait* brick. However, when a script executes a *Start sound and wait* brick, the script pauses until the playback of the PCSound has finished. When interrupting and restarting the playback of a PCSound by playing the same PCSound again, the script which was playing the first instance of the PCSound immediately resumes its execution.

Parameter	Range	Type
1 Sound	Arbitrary PCSound	PCSound selection menu

Table 7.1: Parameters of the *Start sound and wait* brick

7.1.2 Effect bricks

The effect bricks are a set of three bricks that directly affect PCSounds played by *Start sound* and *Start sound and wait* bricks. An audio effect is a digital signal processing algorithm that takes the input samples of audio signals and transforms them into a new series of output samples to modify their original sound in the desired way.

Set [1] effect to [2] This brick can be adjusted by two different values. The first one selects the effect that is applied to the PCSounds while the second one controls an effect specific parameter that usually defines how strongly the effect modifies the original signal. The value range of this parameter is effect specific and cannot be generalised. Effects are applied on the PCObject level, meaning that an effect modifies all PCSounds played by the PCObject that executed the effect brick. Each PCObject can activate one instance of every available effect.

Parameter	Values/Range	Type
1 Effect	[Pan, Pitch]	Effect selection menu
2 Effect intensity	Pan: -100 to 100 Pitch: -360 to 360	Decimal

Table 7.2: Parameters of the *Set effect to* brick

Change [1] effect by [2] Instead of setting an absolute value, this brick changes the effect intensity value by a specified amount. Otherwise, there are no differences to the *Set effect to* brick.

Parameter	Values/Range	Type
1 Effect	[Pan, Pitch]	Effect selection menu
2 Effect intensity change	Pan: -200 to 200 Pitch: -720 to 720	Decimal

Table 7.3: Parameters of the *Change effect by* brick

Clear sound effects When this brick is executed, all active effects of the executing PCObject are cleared. The effects of other PCObjects are not affected.

Pocket Code's redesigned audio engine supports two different effects which are also available in Scratch: Pan left/right and pitch. The creators of Scratch originally planned to implement more effects but removed them before the release of Scratch 3.0[1]. It is possible that Scratch and Pocket Code will support additional effects in the future.

Pan left/right A panorama or pan control is used to modify the apparent position of an audio signal in a multichannel audio processing environment. This is done by adjusting the relative amplitudes of the signal's channels[2].

[1][Masb].
[2][Zöl02], p. 138.

Pocket Code's audio engine works with a stereo setup. To shift an incoming signal to the right, the pan control increases the amplitude of the right channel and attenuates the amplitude of the left channel. The position shifts further to the right until the amplitude of the left channel is zero. The pan effect takes values between -100 and 100. A value of 100 shifts the signal to the very right, -100 to the very left and 0 to the center.

Pitch The pitch effect adapts the pitch of an incoming sound. This is done by using a method called "variable speed replay", which increases or decreases the playback speed of audio data[3]. Doubling the playback speed will increase the pitch by one octave, halving it will lower the pitch by one octave. This phenomenon can also be observed when fast forwarding a cassette in a cassette player or adapting the rotation speed of a turntable. Changing the pitch effect value by 10 will lead to a pitch increase/decrease of a semitone. To go up a whole octave, the value therefore has to be increased by 120. The value has to be between -360 and 360 which is equivalent to a decrease or increase of three octaves[4].

7.1.3 Music bricks

Music bricks are a set of six bricks that give Pocket Code users the opportunity to arrange and compose their own music with 21 different musical software instruments and a drum kit. The following music bricks are available:

Play note [1] for [2] beats When executed, this brick plays a note with the currently active instrument. Parameter one sets the note that has to be played. A value of 69 represents the concert pitch A4 (440 Hz). A change by one integer value is equivalent to a change of one semitone. Parameter two determines the duration of the played note in beats, which is a relative measure. The actual length of a beat can be set with the *Set tempo to* brick. The subsequent brick gets executed exactly after the defined duration of

[3][Zölo2], p. 202.
[4][Masb].

the *Play note* brick. A script can only play one note at a time. To let a PCObject play multiple notes simultaneously, notes must be played from within different scripts. The volume that is set with the *Set volume to* or *Change volume by* bricks for all audio players of a PCObject is also valid for software instruments of the same PCObject.

Parameter	Range	Type
1 Note	0 - 130	Integer
2 Duration	0 - ∞	Decimal

Table 7.4: Parameters of the *Play note* brick

Set instrument to [1] By default, all notes are played by a piano. By using this brick, the active instrument can be set to any of the available 21 instruments. Once this brick is executed, notes will be played with the new instrument. This brick works on the PCObject level, meaning that a PCObject can only play one instrument at a time. Once a PCObject executes the *Set instrument* brick, the instrument will change for all notes played by this PCObject. Notes that started playing before the instrument was changed will finish playing with the old instrument. The available instruments are: piano, electric piano, organ, guitar, electric guitar, bass, pizzicato violin, cello, trombone, clarinet, saxophone, flute, wooden flute, bassoon, choir, vibraphone, music box, steel drum, marimba, synth lead and synth pad.

Parameter	Range	Type
1 Instrument	1 - 21	Instrument selection menu

Table 7.5: Parameters of the *Set instrument* brick

Play drum for [1] beats The *Play drum* brick is very similar to the *Play note* brick. It will always play one element (hi-hat, snare, tom etc.) of a drum kit and is therefore not affected by the *Set instrument* brick. Parameter one chooses the drum element being played and parameter two sets the duration in the same way as the *Play note* brick. The drum sounds are always played

until the end, even if the chosen duration is smaller than the length of the sound. Nevertheless, the subsequent brick will be executed exactly after the chosen duration. PCObjects can play the drums simultaneously with a normal instrument and in the same manner as for normal instruments, the *Set volume to* and *Change volume by* bricks also affect the volume of drums.

Parameter	Range	Type
1 Drum sound	1 - 18	Drum selection menu

Table 7.6: Parameters of the *Play drum* brick

Rest for [1] beats This brick pauses a script for a specified amount of beats and is primarily intended for adding a break between two notes.

Parameter	Range	Type
1 Duration	0 - ∞	Decimal

Table 7.7: Parameters of the *Rest* brick

Set tempo to [1] Sets the tempo in beats per minute (BPM). The new tempo is valid globally for notes and pauses played by all PCObjects.

Parameter	Range	Type
1 Tempo	20 - 500	Decimal

Table 7.8: Parameters of the *Set tempo to* brick

Change tempo by [1] Changes the tempo relative to the previously set BPM. The new tempo is valid globally for notes and pauses played by all PCObjects.

Parameter	Range	Type
1 Tempo	-480 - 480	Decimal

Table 7.9: Parameters of the *Change tempo by* brick

7.2 Choosing the right audio framework

After a precise analysis of the new audio features and the original audio engine's weaknesses, the available audio frameworks were evaluated for their suitability. Reasons for or against the individual frameworks are listed below.

High-level audio classes The high-level audio classes used to build the original audio engine do not allow further processing of generated audio data as it is required for the effect bricks. Additionally, a high-level class capable of playing musical instruments does not exist. In combination with the missing capability to access and analyse the generated audio data, a requirement to directly test the engine's audio output, these drawbacks make the high-level audio classes unsuitable for the redesign of Pocket Code's audio engine.

Core Audio As Core Audio builds the foundation for all audio processing on Apple's operating systems, providing all necessary tools to build any kind of audio application, it would fulfil all technical requirements for the redesign of Pocket Code's audio engine. However, its complexity and the additional need for a lot of domain specific knowledge make it unsuitable for a small, community driven open source project like Pocket Code.

AVAudioEngine As discussed in section 5.3, AVAudioEngine provides a set of nodes which can be divided into source nodes, processing nodes and destination nodes. Two of the available processing nodes, `AVAudioUnitMixer` and `AVAudioUnitTimePitch` could be used to implement Pocket

Code's new panorama and pitch effects. Furthermore, AVAudioEngine offers the capability to create software instruments with its source node *AVAudioUnitSampler* which would allow for the realization of all new music bricks. With AVAudioEngine's graph structure, Pocket Code's audio engine could be easily extended if new features have to be implemented in the future. Additionally, AVAudioEngine also provides the possibility to install so called "audio taps" within graphs. Audio taps can be installed on arbitrary buses between nodes to record the audio data that is sent from one node to another. This allows the audio data to be analysed at any location in a graph, making it possible to verify real audio output in test cases. Another feature allows audio data to be rendered offline with AVAudioEngine, which also provides the possibility to test real audio data. Both of those features are discussed in more detail in section 9.2 and section 9.3. With its simple interface AVAudioEngine would be an adequate framework for the audio engine's redesign.

AudioKit AudioKit is largely based on AVAudioEngine. With that in mind, everything that has been said about AVAudioEngine is equally valid for AudioKit. AudioKit is simple enough to be maintained by developers that are not familiar with the theory of digital audio processing but still powerful enough to fulfill all important requirements of Pocket Code's new audio engine. However, there are a few points which make AudioKit a more suitable choice for Pocket Code than AVAudioEngine. First of all, AudioKit is a widely used, open source project with an active community, which is continually improving and extending its functionality. Furthermore, thanks to the integration of other audio processing libraries, AudioKit provides many additional nodes and features that AVAudioEngine does not offer. The additional functionality, which includes audio analysis nodes, UI elements for audio visualization and a wider range of effect nodes, offers more flexibility for future extensions and improvements of Pocket Code's own functionality. Due to these advantages, AudioKit was ultimately chosen as the preferred tool to build Pocket Code's new audio engine.

7.3 Building Pocket Code's audio graph

This section discusses how AudioKit was used to implement Pocket Code's audio requirements, describes the classes and tools used in the redesign process and finally details the full structure of Pocket Code's redesigned audio engine. The following list gives a brief summary about the basic tools needed to implement the new audio engine with a graph structure:

- Audio players, that play PCSounds when a *Start sound* or *Start sound and wait* brick is executed.
- Speech synthesizers to provide TTS functionality needed by the *Speak* and *Speak and wait* bricks.
- A component for volume control to adapt the volume of all PCSounds, instruments and drums on the PCObject level.
- Mixer components to mix the audio data of all source nodes (audio players, software instruments and speech synthesizers) into a single audio output.
- A pan effect to modify the stereo panorama of PCSounds played by *Start sound* and *Start sound and wait* bricks.
- A pitch effect to modify the pitch of PCSounds played by *Start sound* and *Start sound and wait* bricks.
- A software instrument component that can play a variety of instruments and drums.

The AudioKit classes used to implement these components are explained below. As AudioKit makes heavy use of AVAudioEngine, some of the used AudioKit classes rely directly on equivalent AVAudioEngine classes. In those cases, the description will focus more on the underlying AVAudioEngine functionality than on the actual AudioKit class.

7.3.1 Audio player

To play audio files in the context of an AudioKit graph, AudioKit offers a source node called `AKPlayer`. Internally, `AKPlayer` makes use of AVAudio-Engine's source node `AVAudioPlayerNode` and extends its functionallity by offering additional features like fading and looping. As none of the

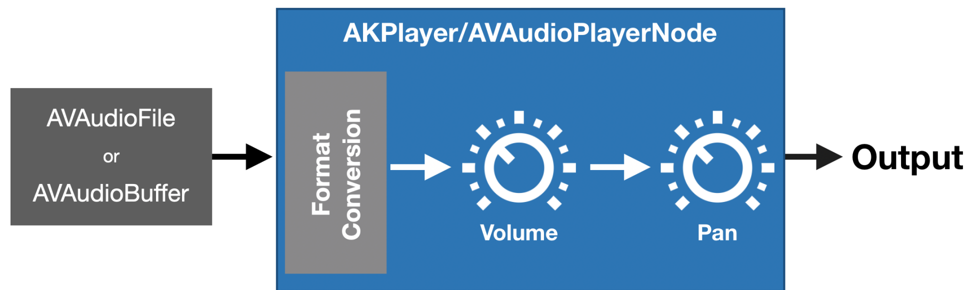

Figure 7.1: Visualization of *AKPlayer*'s/*AVAudioPlayerNode*'s functionality

additional features are currently relevant for Pocket Code's audio engine, this section only describes the functionality of the underlying *AVAudio-PlayerNode*.
AVAudioPlayerNode, depicted in figure 7.1, is a node used for audio playback and supports various audio formats. Audio data can either originate from audio files or from an external source which provides audio buffers for playback. In Pocket Code, the audio data always originates from audio files directly. Playback of audio data can either be started immediately or scheduled at a later time.
As discussed in section 5.3, all nodes have a number of input and output buses with configurable formats which are characterized by parameters like channel count and sample rate. Source nodes like *AVAudioPlayerNode* are special cases as they are always placed at the beginning of a processing chain and therefore only have output, but no input buses. An *AVAudio-PlayerNode*'s output format can be configured differently to the format of the audio files it is playing. In such cases, the *AVAudioPlayerNode* will automatically convert the audio data's channel count and sample rate to match the player's output format. However, it is recommended that the player's output format always matches the format of the played audio files. If format conversions are necessary, Apple recommends doing so with an *AVAudioMixerNode* (discussed in section 7.3.3). Last but not least, *AV-AudioPlayerNode* also offers the possibility to modify the volume and panorama of its audio output.

7.3.2 Speech synthesizer

AVAudioEngine does not provide a class for speech synthesis which could be integrated into an audio processing graph. AudioKit circumvents this lack of TTS functionality by creating a speech synthesizer node with speech synthesis tools from Core Audio. Unfortunately, the resulting `AKSpeech-Synthesizer` node is only compatible with macOS and not iOS[5]. As the original audio engine's speech bricks suffer from weaknesses caused by limitations of the high-level `AVSpeechSynthesizer` class, several open source TTS frameworks were evaluated as a replacement, but non of them offered the necessary functional scope and a satisfactory speech quality.
Due to this, the redesigned audio engine still uses `AVSpeechSynthesizer` for its speech bricks. The TTS functionality is therefore decoupled from the rest of the audio engine, remains very difficult to test and cannot be integrated into the new graph structure. To prevent the problem of delayed speech output caused by placing utterances in a queue, the redesigned audio engine no longer places new utterances in a queue when the speech synthesizer is already speaking. Instead, if a new utterance is handed to the speech synthesizer, the currently speaking utterance is stopped and the new one starts speaking immediately. If the stopped utterance was executed by a *Speak and wait* brick, the associated script resumes immediately.
As Apple has been slowly extending AVAudioEngine's functionality over the past few years, it is possible that a speech synthesis node will be added in the future. This node could then serve as the basis for AudioKit's own iOS compatible speech synthesis node.

7.3.3 Mixer

AudioKit handles mixing with a class called `AKMixer` which is a wrapper around AVAudioEngine's `AVAudioMixerNode` and adds a few convenience methods. As visualized in figure 7.2, `AVAudioMixerNode` is used to mix any number of input buses into a single output bus. The inputs can be of different sample rates and will be automatically converted to the sample rate of the output bus' audio format. Additionally `AVAudioMixerNode`

[5][Appm].

will also upmix or downmix the channels of the input buses to the channel count of the output bus[6]. In the same way as for the *AVAudioPlayerNode*, the volume and the panorama of the mixer output are adjustable.

Figure 7.2: Visualization of *AKMixer*'s/*AVAudioMixerNode*'s functionality

7.3.4 Software instrument

The available musical instruments and drums in Pocket Code should all emulate their real counterparts. There are two basic ways this can be accomplished: with a synthesizer or a sampler. A synthesizer produces different digital waveforms which can be layered and further processed to create the desired sound. Unfortunately, emulating a real instrument with a synthesizer will usually lead to artificial sounds because the complex waveforms of real instruments cannot be recreated with sufficient precision. A sampler on the other hand plays pre-recorded audio files. To emulate an instrument with a sampler, one would simply record sounds played by a real instrument, and play them with the sampler. This assures, that the software instrument really sounds as close to the original as possible. AudioKit provides a node called *AKSampler* for this purpose.

How does a sampler work? Since a sampler simply plays pre-recorded audio files, the question arises of how a sampler is different from the previously discussed audio player node. Indeed, a sampler does have similarities to an audio player and could be described as an audio player with extended functionality. [Zölo2] defines a sampler as a digital system for recording

[6][Appb].

and playing back short musical sounds in real-time which is controlled by a MIDI keyboard or controller[7]. The real-time and MIDI aspects are important information in this definition.
MIDI (short for Musical Instrument Digital Interface) is a communication protocol used to transmit information about musical events. In Pocket Code, there are two different MIDI events that are significant: the *Note on* and *Note off* events. A *Note on* event is sent to a software instrument (the sampler) whenever the controlling instance (a person playing a keyboard or in this case, Pocket Code executing a *Play note* brick) orders the software instrument to play another note. The *Note off* event is sent to the software instrument when a note, which has previously been triggered by a *Note on* event, should stop playing. *Note on* and *Note off* events always contain information about the pitch and velocity of the note that should start or stop playing[8]. Pocket Code works with a constant velocity for all played notes, which means that only the pitch information is of interest.
Ideally, a sampler would contain a recoded audio file (called sample) of every note that can be played on a particular instrument. Whenever a *Note on* event reaches the sampler, it would retrieve the sample that corresponds to the requested pitch and play back this file until a *Note off* event is registered or the file has finished playing. Unfortunately, storing a sample of every possible note for every instrument would take up a lot of memory. Instead, sample instruments often only contain samples of a few notes, distributed over the whole pitch range of the instrument. Whenever the sample instrument does not have a sample of the requested note, the sample of the closest available note is used for playback. To produce the correct pitch from this sample, the sampler uses the variable speed approach described in section 7.1.2. With this approach, the playback of a sample is either sped up or slowed down in order to generate the requested pitch. Figure 7.3 visualizes this process on a keyboard. The red keys represent the notes for which a recorded sample is available. Those notes are called root notes and are evenly distributed over the whole pitch range. If a *Note on* event for one of the root notes is received, the corresponding sample is played without any further modification. If a *Note on* event for a note in the green, blue or yellow area is fired, the sample of the root note which lies within the

[7][Zöl02], p. 522.
[8][MID].

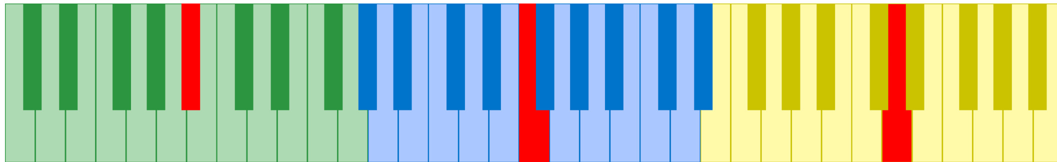

Figure 7.3: Keyboard layout of a sample instrument

respective area is played, but with a modified speed.

AKSampler AudioKit offers two different sampler classes. The first one is based on AVAudioEngine's sampler called `AVAudioUnitSampler` whereas the second one, called `AKSampler`, was developed by the AudioKit team from scratch. Although `AKSampler` is nowhere near as powerful as `AVAudioUnitSampler`, it meets all requirements of Pocket Code's music bricks and is very easy to handle and well documented[9]. For these reasons, `AKSampler` was finally chosen for the implementation of Pocket Code's music bricks. `AKSampler` offers different mechanisms to load samples. As depicted in figure 7.4, they can either be loaded from audio data that is already in the application's memory, from individual audio files, or from SFZ (Sforzando) soundfont files. In Pocket Code, samples are loaded from SFZ files by calling `AKSampler`'s method `loadSFZ(path: fileName:)`. SFZ files are essentially structured text files, describing how a set of samples is arranged to build a software instrument[10]. As such, they tell `AKSampler` which samples are used for a specific instrument, where the sample files are located, which root note is associated to each sample, and the range of notes that is played by each sample (the differently coloured areas in figure 7.3). Pocket Code uses the same samples as Scratch[11] to create its own software instruments and additionally compresses the samples with WavPack[12] to minimize their memory footprint. Once an SFZ file has been loaded by the sampler, *Note on* and *Note off* events can be fired by calling `AKSampler`'s methods `play(noteNumber: velocity:)` and `stop(noteNumber:)`. It is also worth mentioning, that `AKSampler` is a polyphonic sampler, mean-

[9][Aud].
[10][SFZ].
[11][Masa].
[12][Bry].

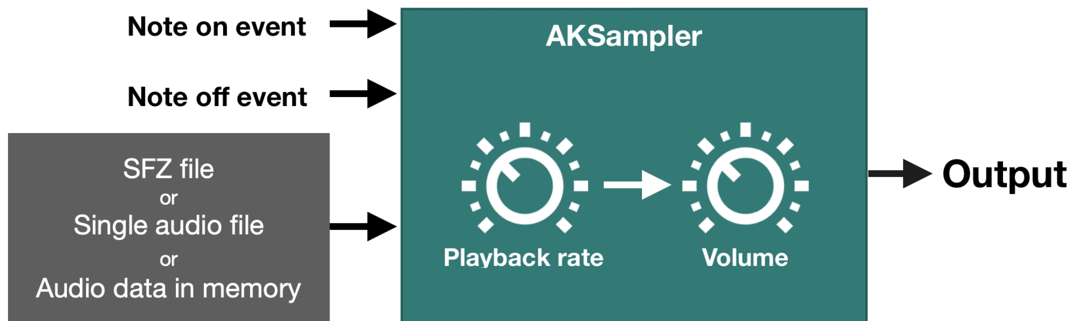

Figure 7.4: Visualization of `AKSampler`'s functionality.

ing that it can play multiple notes simultaneously. However, the same note can only be played once at a time.

7.3.5 Volume

To adapt the volume when executing *Set volume to* or *Change volume by* bricks, Pocket Code does not need separate processing nodes in its audio graph. As shown above, `AKPlayer`, `AKMixer` and `AKSampler` are already capable of controlling the volume of their output signals. Since the volume in Pocket Code is always changed on the PCObject level and not for individual audio players or software instruments, the audio engine always changes the volume with `AKMixer`'s volume control.

7.3.6 Pitch effect

As mentioned in section 7.1.2, the pitch effect uses the variable speed replay technique to shift the pitch of incoming audio data. AudioKit offers a class called `AKVariSpeed` which is based on Apple's `AVAudioUnitVarispeed` node. The only parameter that can be controlled with `AKVariSpeed` is the audio playback rate.

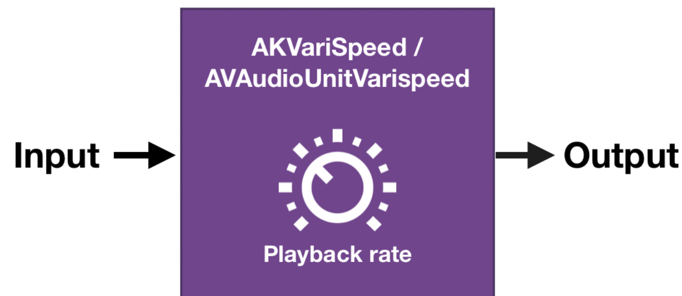

Figure 7.5: Visualization of *AKVariSpeed*'s/*AVAudioUnitVarispeed*'s functionality

7.3.7 Pan effect

Although the pan effect could be implemented with `AKMixer`'s panorama control, the redesigned audio engine uses a separate node for this purpose. This decision was made to provide a uniform interface for all effects in Pocket Code. As the pitch effect uses a separate node and possible future effects most likely require separate nodes as well, it would add unnecessary complexity to the audio engine's code if the pan effect would be an exception to this rule.
The pan effect is realized with AudioKit's `AKPanner` class, visualized in figure 7.6. To attenuate or boost the left or right channels, `AKPanner` multiplies the audio data of both channels with a gain between zero and one. The gains for the left and right channels are calculated as follows[13]:

$$\begin{aligned} gain_l &= \sin(0.5 * \pi * pan) \\ gain_r &= \cos(0.5 * \pi * pan) \end{aligned} \tag{7.1}$$

In this formula, the panorama (pan) has to be a value between 0 and 1 where 0 is hard left, 0.5 center position and 1 hard right. This gain calculation formula ensures that the power of the output signal stays roughly the same when it is shifted to another position in the stereo panorama.

[13][Cre17], p. 125.

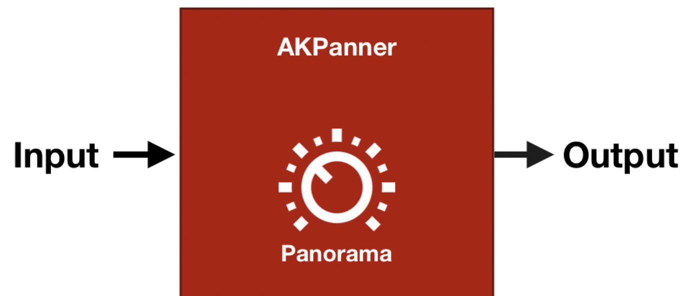

Figure 7.6: Visualization of `AKPanner`'s functionality

7.3.8 Combining nodes to a PCObject subgraph

Now that all required nodes for the redesigned audio engine are known, they can be connected to a graph. To make this process easy to understand, a subgraph of a single PCObject will be derived first. Whenever the term subgraph is used in the following chapters, it refers to a PCObject subgraph. The subgraphs of all PCObjects will later be connected to a full graph.
Figure 7.7 shows a functional diagram of the audio flow within a single subgraph. A PCObject can have an arbitrary number of audio players, but only one for each distinct PCSound (visualized by the differently coloured audio players). The figure shows three audio players. Their signals (1, 2 and 3) can be processed with a pan and pitch effect. Since these effects have to affect all audio players of a PCObject equally, the signals are first mixed together (4) before the effects are applied (5 and 6). The figure also shows that there can be exactly one melodic instrument and one drum kit in each subgraph. Their signals (7 and 8) and signal 6 are mixed together (9), before the overall volume of all audio players and software instruments of a PCObject is adjusted (10).
Figure 7.8 shows the same diagram again, but this time not only displaying the functional features of a subgraph, but also its actual node structure.

To keep the memory footprint as low as possible, the redesigned audio engine uses lazy instantiation to create its subgraphs. This means, that subgraphs will not exist until a PCObject executes its first sound, effect or music brick. The first time such a brick is executed by a PCObject, the audio engine initializes the subgraph by creating and connecting the two `AKMixer`

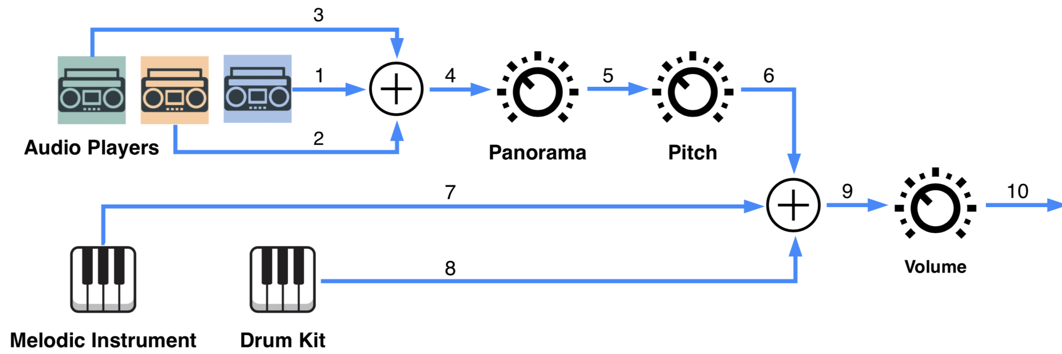

Figure 7.7: Functional diagram of a subgraph

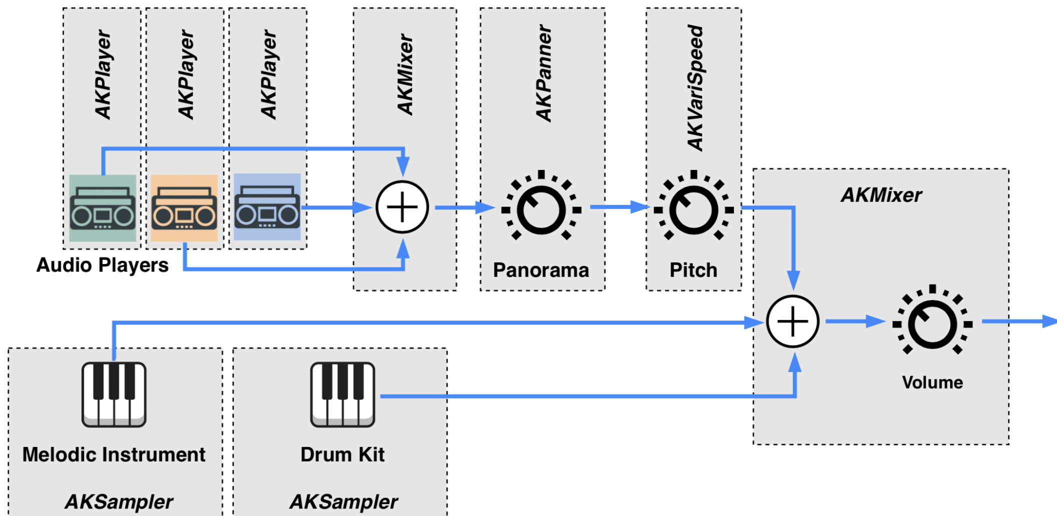

Figure 7.8: Connected nodes forming a subgraph

nodes, the `AKPanner` node and the `AKVariSpeed` node and connecting the subgraph's output to the audio graph's main mixer node. Depending on the executed brick, an `AKPlayer` or `AKSampler` will also be connected to the subgraph. For every successive execution of a sound, effect or music brick by the same PCObject, the subgraph will be supplemented by a new `AKPlayer` or `AKSampler` if necessary, but will not change its structure any further.

7.3.9 Combining subgraphs to a full graph

Now that the structure of a subgraph is know, combining them to build a full graph is very easy. As illustrated in figure 7.9, the subgraph's audio outputs are connected with another `AKMixer` node which is the graph's main mixer node, responsible to combine all subgraph signals to a single audio signal that can be sent to the device's audio output. Note that neither the subgraphs nor the full graph contain any information about speech synthesizers, as Pocket Code's TTS functionality is still handled separately.

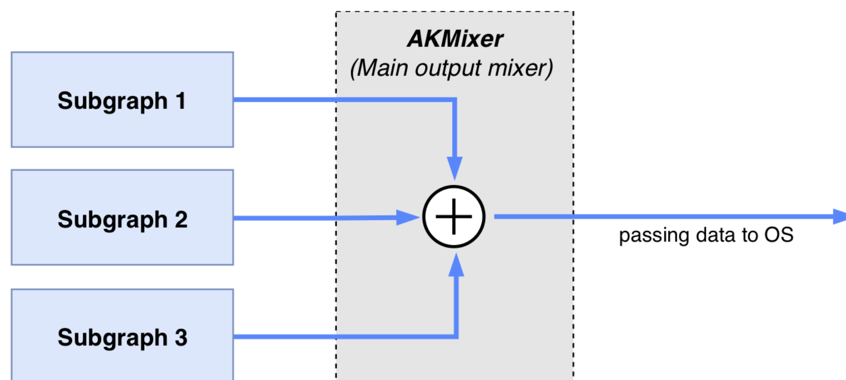

Figure 7.9: Connecting subgraphs to a full graph

7.4 Audio engine architecture

After defining and analysing the structure of the audio graph, the most vital part of Pocket Code's redesigned audio processing capabilities, it has to be

embedded into an architecture that maintains the graph and controls its behaviour to form what will be the finished audio engine. This section gives a technical overview of the audio engine's architecture and highlights some of the used classes and concepts. A visualization of the architecture can be seen in figure 7.10.

7.4.1 CBAudioEngine

`CBAudioEngine` (CB stands for Catrobat) is the central class of Pocket Code's audio engine, containing the complete audio graph and acting as the main gateway for every request or command sent to the engine. The simplest way of explaining `CBAudioEngine`'s functionality is by looking at it from three different angles: defining the public interface which is used to interact with the class, inspecting the data it encapsulates, and analysing the behaviour or logic that is initiated by calls to the interface and performed on the data.

Interface `CBAudioEngine`'s interface consists of a set of method's that can be divided into three groups.

- The first group of interface methods is called by executed audio bricks (arrow A in figure 7.10) and consists of a total of thirteen methods: `playSound()`, `stopAllAudioPlayers()`, `setVolumeTo()`, `changeVolumeBy()`, `speak()`, `setEffectTo()`, `changeEffectBy()`, `clearSoundEffects()`, `playNote()`, `playDrum()`, `setTempoTo()`, `changeTempoBy()` and `setInstrumentTo()`. When executing one of these bricks, the interface method which matches the executed brick's name is called, signalling the audio engine to perform the desired operation. Although not specified for simplicity, the methods listed above each take a number of parameters: the actual parameter values of the executed brick (for example the volume, the effect value or the sound file) and the name of the PCObject that executed the brick.

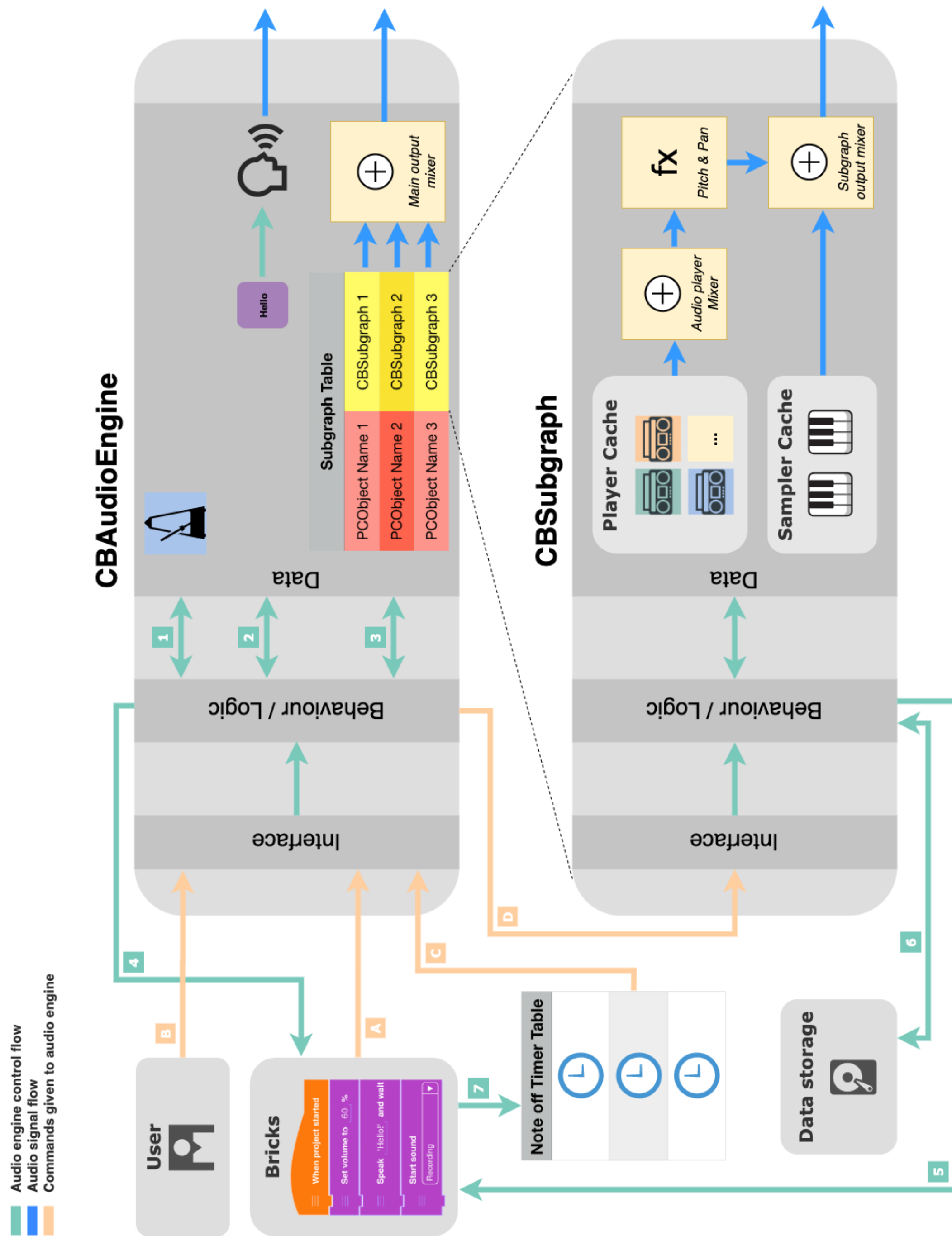

Figure 7.10: Visualization of the redesigned audio engine

- The second group of interface methods is called by users changing the operating state of a PCProject (arrow B), namely starting, stopping, pausing or resuming it. A change in the operating state of a PCProject must also be reflected in the operating state of the audio engine. The corresponding interface method names are *start()*, *stop()*, *pause()* and, *resume()*.
- The third group of interface methods, consisting of *stopNote()* and *stopDrum()* is called by a different part of Pocket Code. The purpose of the two methods is discussed in section 7.4.3.

Data *CBAudioEngine* only encapsulates a small amount of data. The first object it manages is the graph's main output mixer node depicted in figure 7.9. It also maintains a subgraph table, which is a table that contains all subgraphs connected to the the graph's output mixer node. The stored objects are of type *CBSubgraph* (see section 7.4.2) and are stored in the table as key-value pairs with the key being the name of the PCObject associated to the subgraph. As the speech synthesizer is still handled separately from the AudioKit's graph structure, *CBAudioEngine* also holds an *AVSpeechSynthesizer* object for Pocket Code's TTS functionality. The last important piece of data is the audio engine's metronome, a numerical value representing the currently set BPM valid for all music bricks.

Behaviour As *CBAudioEngine* only encapsulates a small amount of data, the number of operations that are performed on that data within *CBAudioEngine* are also limited. There are four basic behavioural areas *CBAudioEngine* covers:

- **Setup and Teardown:** *CBAudioEngine* is responsible to do some basic setup and teardown work. First of all, this includes initializing its own internal data (output mixer node, subgraph table, speech synthesizer and metronome), but more importantly also managing the state of AudioKit. *CBAudioEngine* has to inform AudioKit of the audio graph's main output node, so that AudioKit can forward the audio data to the device's audio hardware. *CBAudioEngine* also has to start the audio processing graph once a PCProject is started. On

the other hand, when a PCProject is stopped, AudioKit has to be shut down and disassembled properly.

- **Translating interface calls:** `CBAudioEngine`'s main task is translating and forwarding the interface calls to the corresponding `CBSubgraph` objects (arrow D), as most operations are performed on the PCObject level and therefore handled within `CBSubgraph`. To do that, `CBAudioEngine` searches the subgraph table for the corresponding subgraph on which the operation has to be performed. It does so by checking if the name of the PCObject that called the interface method is available as a key in the subgraph table. If it is, `CBAudioEngine` retrieves the `CBSubgraph` object and forwards the call to `CBSubgraph`'s interface. If not, `CBAudioEngine` instantiates a new `CBSubgraph` object, stores it in the table and forwards the call to the newly created object (arrow 3).
- **Operating global audio engine features:** Although most commands given to the audio engine are only relevant for a specific subgraph as described above, some apply to the whole audio engine and are therefore directly processed within `CBAudioEngine`. Two of those commands are the calls to the `setTempoTo()` and `changeTempoBy()` interface methods. The tempo of music bricks is a global state valid for all samplers of the audio engine, which is why said interface methods directly adapt the tempo without forwarding the calls to a `CBSubgraph` object (arrow 1).
  As the speech synthesizer resides within `CBAudioEngine` and not within `CBSubgraph`, `CBAudioEngine` additionally also has to take care of the correct operation of Pocket Code's TTS functionality. `CBAudioEngine` therefore has to ensure that the speech synthesizer starts speaking when the `speak()` interface method is called. It also has to stop the currently speaking utterance before a new utterance can start speaking (arrow 2). Finally, the `CBAudioEngine` has to make sure that scripts which were halted due to a *Speak and wait* brick are resumed at the right time. The right time to resume such a script is either when the current utterance is stopped by a new utterance or when the utterance finishes speaking and subsequently triggers a `utteranceDidFinish()` callback (arrow 4).
- **Ensuring thread safety:** As `CBAudioEngine` is accessed by many different scripts from different PCObjects and threads, it has to make sure

that thread safety is guaranteed. Without paying attention to thread safety issues, *CBAudioEngine* could otherwise show unexpected and unintended behaviour. Specifically, *CBAudioEngine* ensures that *CBSubgraph* objects can only be created one at a time, which prevents the creation of multiple subgraphs for the same PCObject. Also, *CBAudioEngine* serializes much of the speech synthesizer's behaviour. This ensures that different threads do not interfere with each other and actions like starting and stopping the speech synthesizer as well as resuming paused scripts happen in the correct order.

7.4.2 CBSubgraph

As already mentioned, *CBSubgraph* is a class that encapsulates the data and functionality of a whole audio subgraph. Its interface is not publicly available and is solely accessed by *CBAudioEngine*. In the same way as for *CBAudioEngine*, this section will explore the interface, data and behaviour of *CBSubgraph*.

Interface *CBSubgraphs*'s interface is almost identical to the interface of *CBAudioEngine*. This is due to the fact, that *CBAudioEngine*'s main task is to maintain the subgraph table and to forward the incoming commands directly to the correct subgraph. Interface method calls in *CBAudioEngine* are therefore delegated to the method of the same name of the according *CBSubgraph* object (arrow D). Almost all interface methods found in *CBAudioEngine* are also available in *CBSubgraph* with a few differences:

- The *speak()* method, does not exist in *CBSubgraph* as the TTS functionality is directly handled in *CBAudioEngine*.
- *CBSubgraph* provides an additional *connectSubgraphTo()* method. This method is called by *CBAudioEngine* to connect a newly created subgraph to the audio engine's main output mixer node.

Data *CBSubgraph* holds all the nodes which form a PCObject subgraph. This includes two mixer nodes, two sampler nodes, an *AKPanner* and

an *AKVarispeed* node, as well as an indefinite amount of *AKPlayer* nodes (see figure 7.8). The player and sampler nodes are not directly held within *CBSubgraph* but rather contained inside player and sampler caches which are maintained by *CBSubgraph*. The caches are implemented with *NSCache* which has been described in section 6.2.1.

Behaviour Similar to *CBAudioEngine*, *CBSubgraph* also performs a variety of operations with or on its data. As the data contained in *CBSubgraph* differs heavily from the data in *CBAudioEngine*, the operational code of those two classes pursues different objectives. Nevertheless, some of *CBSubgraph*'s behaviour is quiet similar to *CBAudioEngine* as shown below.

- **Setup:** Similar to *CBAudioEngine*, *CBSubgraph* also performs some basic but important setup tasks. This setup work mainly consists of initializing the basic (sub)graph structure. This includes the initialization of the nodes *CBSubgraph* holds, connecting those nodes to form a working subgraph and connecting the subgraph to the main output mixer node (located inside *CBAudioEngine*). An initialized subgraph at least holds two mixer nodes (the subgraph output mixer node and the player mixer node), as well as an *AKPanner* and *AKVariSpeed* node. All other subgraph nodes are created and set up at a later point as needed.
- **Translating interface calls:** In contrast to *CBAudioEngine* which translates incoming calls to the correct *CBSubgraph* object, *CBSubgraph* translates and forwards its interface calls to the correct nodes. This process includes the retrieval of the correct player or sampler nodes from the player and sampler caches on *playSound()* or *playNote()* calls or the creation of such nodes if they do not exist yet at that time. Another example would be the retrieval of the correct effect node when the *setEffectTo()* interface method is called.
- **Operating the subgraph nodes:** *CBSubgraph* is also responsible for performing the desired actions on the nodes retrieved in the previous *translating interface calls* step. This means that the commands given by executed bricks or user interactions, which subsequently directly or indirectly invoke interface calls in *CBAudioEngine* and *CBSubgraph*,

are finally executed on their destined subgraph nodes. In most cases, the most vital part of this process is as simple as calling a method or setting a variable on the destined nodes. The `setVolumeTo()` or `stopAllSounds()` interface calls for example primarily require the `volume` variable to be set on the mixer node or the `stop()` method to be called on all player nodes respectively. However, as Pocket Code's audio engine has to meet the many requirements and specifications outlined in previous chapters, the correct operation of all nodes involves additional logic and processes. This includes processes such as the management of the player and sampler caches, ensuring that parameters stay within the specified bounds, enabling or disabling effects based on their usage, pausing and resuming scripts when a *Start sound and wait* brick starts or finishes playing (arrow 5) as well as keeping track of the currently active notes of the samplers.

- **Accessing the device storage:** In order to fulfill their tasks, audio players and samplers need some additional external resources when they are initialized. In case of an audio player this is a simple audio file whereas a sampler requires an SFZ file (see section 7.3.4). `CBSubgraph` interacts between the device storage and the player and sampler nodes, accessing the resources on the storage and providing them to the nodes (arrow 6).
- **Ensuring thread safety:** Just like `CBAudioEngine`, `CBSubgraph` objects can also be accessed from many different threads at the same time and therefore also have to ensure thread safety for some of their processes. To avoid the creation of duplicate audio players and samplers, `CBSubgraph` serializes their instantiation process. It also serializes the execution of commands to start and stop the playback of audio players and samplers to avoid unexpected behaviour.

7.4.3 Other architectural elements

As briefly mentioned at the beginning of this section, there is a third group of interface methods that is not called by executed bricks or users changing the operating state of a PCProject. These methods, `stopNote()` and `stopDrum()`, are called by "note off timers" which are simple timer objects

(arrow C). Note off timers are used to determine when a note, played by a sampler, should stop playing. The timers are initialized with the length of the note they represent and start a countdown as soon as the associated note starts playing on the sampler. When the countdown is finished, the timer calls the `stopNote()` or `stopDrum()` interface method of `CBAudioEngine` to make sure that the note stops playing at the scheduled time. The creation of the timers happens right after the execution of a *Play note* or *Play drum* brick (arrow 7), but before the `playNote()` or `playDrum()` interface methods of `CBAudioEngine` are called. As illustrated in figure 7.10, note off timers are stored outside of `CBAudioEngine` and `CBSubgraph`. The reason for this is simple: Because other parts of the system also use timers to schedule certain actions within a running PCProject, it makes sense to store and handle all those timers in a central place.

8 Automated audio testing

As discussed in previous chapters, software development can greatly benefit from automated testing. It leads to more reliable code, speeds up the development process and increases trust in the project's code base. Most literature on the subject advises to cover as much code as possible with automated tests. The software development methodology "Extreme Programming" for example recommends to write automated tests for "everything that could possibly break"[1]. Although software audio features also fall into the category of code that could break, they often seem to be left out from testing or are tested in a very rudimentary way. While resources on the theory of automated testing and it's general approaches are widely available today, only few publications exist on specific testing approaches of audio functionality in software or hardware projects. The most likely reason for this is, that audio testing is a much harder task than automated testing of conventional code and many developers do not want to deal with the challenges that arise.
This chapter explains the difficulties of audio testing and gives an overview of the few audio testing approaches that are discussed in literature or implemented in open source projects.

8.1 The difficulties of audio testing

Every unit or integration test can have exactly two different outcomes: it either passes or fails. This means, that at the end of every test case, a decision has to be made: Is the result of this test case consistent with the expected result? For typical tests, this decision is trivial. The return value of

[1] [JAH01], p. 234.

a specific function is either equal to the expected value or not. An entry in a database was either set correctly or not. There is no grey area and no room for interpretation.
When it comes to testing audio functionality however, things behave a little differently. Subject of such tests is usually the audio output at any stage of the audio processing facilities of the SUT. The test case, therefore, has to decide if the audio data produced during test execution matches the expected reference audio output. Unfortunately, tiny irregularities in the setup and operation of tests can have a big impact on the resulting audio data, leading to a different audio output than expected. These differences can rarely be perceived by the human ear, but pose a big problem to the verification of test results.
Depending on the test setup, there are different reasons that can lead to irregularities in the captured audio output:

- **Noise:** Being primarily a problem of test setups containing analog equipment, this could either be ambient noise that is recorded together with the relevant signal or any noise added on the analog signal processing path.
- **Time delays:** Playback and recording of the test signal have to be perfectly in sync to match the desired reference signal. This is not only an issue for analog but also for fully digital setups, where the smallest timing differences in code execution can lead to time delays in the audio output.
- **Floating point arithmetic:** Floating point arithmetic has to deal with inaccuracies by design. Mapping any real number to a word length of 32- or 64-bit cannot be done with infinite precision, so rounding errors naturally occur. Unfortunately, these inaccuracies are not always consistent across machines and can vary depending on the compiler or processor architecture used[2]. A test signal created on one machine could therefore have a slight mismatch to the reference signal created on another machine due to small differences in floating point arithmetic.
- **Audio codecs:** Audio data can be encoded into different formats. Many codecs like MP3 or AAC use lossy compression algorithms to minimize

[2][Bol+15], p. 2.

the size of the encoded audio data. This is done by removing parts of the audio data that cannot be perceived by the human auditory system. Although the differences might not be audible for most persons, the waveform of an MP3 file will be different to the waveform of an AAC file of the same song which makes a direct comparison of the two very hard[3]. Developers need to be aware of this if they are not in full control of the testing data's audio format.

- **Time variant processing elements:** A system is time invariant if the output of a time-shifted input sequence is equal to the time-shifted output of the original input sequence[4]. In other words: A time-shift of the input sequence leads to the same time-shift of the output sequence without further affecting it. If a system is time variant, an input signal can lead to different output signals when fed to the system at different points in time. If a time variant signal processing element is used, the recorded audio signal in a test case might be different every time the test is executed and will therefore not match the reference signal. A good example of such a processing element is a tremolo effect which modulates a signal's volume with a low frequency oscillator[5]. Figure 8.1 shows an audio file processed with the same tremolo effect at two different points in time. The differences in the waveforms are clearly visible.

In some cases, the above irregularities can be mitigated or completely removed with a careful test setup, but often the efforts to do so would be too big or the complete removal of all the disruptive factors is not possible. In this case, direct comparisons between test signals and reference signals are no longer possible. The question asked in such test cases can no longer be "Is the test result (the obtained audio signal during the test) equal to the expected result (the reference signal)?" but rather "Is the test result similar enough to the expected result?" or "Does the test result share common properties with the expected result?".

[3] [HK02], p. 4.
[4] [OSB99], p. 2.
[5] [JM14], p. 127.

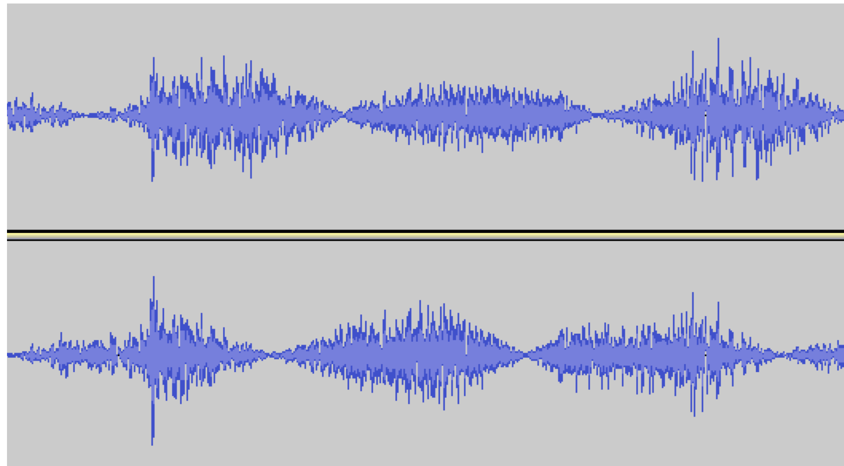

Figure 8.1: A audio signal processed with the same tremolo effect at two different points in time

8.2 Audio testing strategies in literature

Earliest references of automated audio test systems go back to the 1960s[6]. Digital audio equipment was not available to a wide audience at that time, so those early test systems targeted analog audio equipment like loudspeakers or amplifiers. In his 1968 paper "High Speed Automated Test Set"[7], Roberts described an automated test system for a variety of analog audio products, that could be operated by untrained personnel, responded with a simple "pass" or "fail" and drastically reduced testing times compared to manual testing. The system measured the audio output of the device under test and extracted several parameters like frequency response, distortion or maximum gain. The parameters were then verified to be within the allowed limits.

A similar approach was chosen by Richard Cabot, who developed automated systems to test loudspeakers[8] as well as analog expanders and limiters[9] (signal processing devices that modify the gain of their input signals). To

[6][Tak05], p. 29.
[7][Rob68].
[8][Cab86].
[9][Cab87].

facilitate and speed up the previously manual testing procedures, automated test setups were created that produced an input signal for the device under test and analyzed their audio output. From the audio output, relevant parameters were extracted and then verified for correctness.

A more recent publication deals with the "Entertainment Audio Platform", a "system master mixer" developed for Nokia's operating systems in 2005[10]. Studying the many possibilities of next generation smartphones at that time, Nokia recognized the need for a powerful audio platform, capable of managing and mixing audio streams of multiple apps. The Entertainment Audio Platform was capable of performing sample rate conversions, controlling the dynamic range as well as adjusting the volume and stereo panorama of multiple audio channels. Additionally, various audio effects were also supported. [Tak05] describes the development of methods required to test all of these functionalities.

Audio testing is also an issue in the computer music community. Computer musicians use computer technology to create compositions or audio installations. They often make use of visual programming languages and real-time media processing environments like Max[11] or Pure Data[12] to write their programs and compositions (called patches). The 2012 publication "An Automated Testing Suite for Computer Music Environments"[13] highlights the problems arising if such programs are not properly tested. Changes to complex compositions, the target environment or the operating system can lead to unforeseen problems which makes users of such environments hesitant to change their performances or setups. With this in mind, the publication presents an approach to audio testing in computer music environments.

[Tak05] and [PLP12] share significant similarities in their approaches to audio testing, both mentioning two possible approaches to audio testing: Bit-exact testing and parametric testing. The parametric approach can also be found in [Rob68], [Cab86] and [Cab87].

A further approach of automated audio testing can be found in [SSR08] where an existing test system, developed to test Nokia devices, was en-

[10][Tak05].
[11][Cyc].
[12][Puc].
[13][PLP12].

hanced with the method of audio fingerprinting. Audio fingerprinting was implemented to identify the audio output of the device under test, opening up new opportunities to automate the testing of voice control and multimedia functionalities. The tool was developed to clearly identify and distinguish between different audio outputs. To achieve this, unique identifiers called fingerprints were generated for various reference audio files based on their parameters. They were then compared to the fingerprint of the recorded test signal. If the similarity between the test signal's fingerprint and one of the reference fingerprints was large enough, the test signal was identified as the signal of the matching reference fingerprint.

The methods of bit-exact and parametric testing as well as audio fingerprinting, will be described in more detail on the following pages.

8.3 Bit-exact testing

In bit-exact testing, audio data recorded during execution of a test case is compared with a reference audio file that was created earlier. The comparison between the two signals is done bit by bit and will only yield a positive result (a passing test case) if both signals are exactly the same. The reference signal must therefore be created with a system that is considered to be correct (a reference system or reference implementation)[14].
Due to the difficulties described in section 8.1, this approach is often not feasible. In certain cases, when all the factors like timing, noise, time-variance or floating point calculations are guaranteed not to impact or influence the test results, bit-exact testing can still be a useful tool for audio testing.
Bit-exact testing can be found in some open source audio libraries like AudioKit[15] and Soundpipe[16].

[14] [Tako5], p. 26.
[15] [Proa].
[16] [Bat].

8.3.1 Bit-exact testing with hashes

To simplify bit-exact testing, hash functions can be applied to the audio data. This practise can also be found in [Proa] and [Bat]. A hash function takes input data of arbitrary length, processes the data with one of many available hashing algorithms, and returns a fixed-size bit-string called a hash. Hash values can be seen as fingerprints or unique identifiers of their input data and are usually much shorter than their corresponding inputs. They are widely used in modern cryptography, for example for digital signature schemes or the secure storage of passwords[17]. An ideal hash function has the following properties[18]:

- It computes the hash value quickly with any type of input data.
- It is deterministic. The same input data always leads to the same hash value.
- It is pseudo random. A small change in the input data leads to big changes in the output data, so that the new output data seems uncorrelated to the output data of the unchanged input.
- It should be collision resistant. This means, that it's computationally infeasible to find two values with the same hash value.
- It should be computationally infeasible to calculate the corresponding input message from its hash.

In audio testing, hashing is primarily used as a convenient way to compare test signals with reference signals. The procedure can be described in six steps.

1. A reference audio signal is recorded from a reference system.
2. A hash value is calculated from the reference signal. Depending on the hashing algorithm and the security needs of the application, the hash value usually is between 128 and 512 bits in size. Since audio testing normally is not used in security critical contexts, a hash size of 128 should be enough in most cases. As mentioned before, this is much smaller than the size of the original audio data.
3. The hash value of the reference signal is stored as a reference hash in the associated test case.

[17] [PP10], p. 293.
[18] [Dre17], p. 72.

4. The test case gets executed and records a test audio signal from the SUT.
5. The test signal is fed to the hashing algorith to calulate it's hash value.
6. The hash values of the test signal and the reference signal are compared. If they are equal, the test and the reference signal are identical as well and the test passes. If the two hashes are different, the test signal differs from the reference signal and the test fails.

The fact that only the hashes of audio signals and not the signals themselves are compared makes it unnecessary to store the original reference audio files. Instead, small reference fingerprints can be stored directly in the source code of the test cases. When working with a big number of test cases, this can have a big impact on the amount of memory used to store the reference data and also simplifies the handling and structure of the test cases. However, it can be harder to find the cause of a failed test case when the original reference audio file is not available.
For audio testing, only the determinism and speed properties of an ideal hash function are truly important. The other three properties can be neglected to a certain extent, but should not be completely ignored. This can also be seen in [Proa] and [Bat] where the MD5 hashing algorithm is used which is quick and deterministic, but is considered to be cryptographically broken and therefore should not be used in security related applications.

8.4 Parametric testing

When bit-exact testing is not possible, parametric audio testing has to be performed. As previously mentioned, parametric testing has been successfully introduced for analog setups in [Rob68] and [Cab87] many years ago and was later adopted for digital setups by [Tak05] and [PLP12] amongst others. In contrast to bit-exact testing, parametric testing does not compare all samples of a test signal to a reference signal. It only extracts certain features of the test signal which characterize the tested audio processing facilities and compares the extracted parameters with known reference parameters or the parameters from a reference audio signal. To successfully pass a test, the extracted parameters have to be equal to the reference parameters within

a certain tolerance range[19]. Parametric automated audio testing is usually performed in three steps: parameter selection, parameter extraction, and parameter verification[20].

1. **Parameter selection:** Parameter selection is the process of correctly setting up the SUT such that its audio output shows the behaviour which is subject of the test. This process can vary from test to test. In the simplest case, this just involves setting a parameter of an audio processing component to a specific value. When testing a panorama effect, this step would simply set the panorama control of a panorama node to the desired position. In more complex test scenarios this could involve setting multiple parameters of different audio processing components, which might change their values during the course of a test case. This step can also include parameter variation between different test runs to ensure that the audio processing components are tested with a broad range of possible parameter values.
2. **Parameter extraction:** Parameter extraction is performed once a test case has been executed and the recorded audio output is available. In this step, the samples of the test signal are analysed with suitable audio signal analysis algorithms to calculate the parameters which are later verified for correctness. The type of analysis that can be performed on audio signals is extremely wide-ranging. The ultimate choice of used analysis algorithms heavily depends on the SUT and the parameters that have to be extracted and cannot be generalised.
3. **Parameter verification:** Once the extracted parameters from the previous step are available, they can be verified for correctness by comparing them to the reference parameters. As previously mentioned this verification is often performed with a certain tolerance range to account for slight inconsistencies between test runs. More complex test cases can also include the verification of parameters that change over time or the verification of specific observed parameter sequences. In such cases, the test case also has to verify that the extracted parameters have the right values at the right time.

The type of parameters that can be extracted from a test signal are as numer-

[19][PLP12], p. 3.
[20][Tak05], p. 30.

ous as the available extraction algorithms. [PLP12] for example describes approaches to extract the pitch and panorama parameters from test signals. [Tak05] on the other hand shows how parameters like amplitude, signal energy, frequency spectrum or distortion are calculated.

It should be noted, that parametric tests cannot always be performed with arbitrary input signals but often need specific input signals for correct operation. Extracting the frequency response of an audio filter for example requires an input signal which contains the frequencies affected by the filter and verifying the correct behaviour of a pitch shifter is easier with a sinusoidal input signal than with white noise.

8.5 Audio fingerprinting

The concept of fingerprinting is very old and can be traced back thousands of years. Fingerprints can be found on many historical items, proving that their individuality was no secret to people throughout time. From the late 16th century onward, fingerprinting increasingly became a subject of scientific research. In 1880, Scottish surgeon Henry Fauld was the first to scientifically demonstrate the individuality of fingerprints based on empirical observations. He therefore suggested to use them for identification purposes. Continued interest in the topic and further scientific advancements ultimately lead to fingerprint recognition being formally accepted as a valid personal identification method. Over all these years, fingerprinting has also become an essential tool in forensics and criminal investigations. Many fingerprint identification agencies have been set up and criminal offenders are registered in fingerprint databases all over the world[21].
Conceptually, a fingerprint is nothing else than a drastically reduced set of a person's properties or parameters that still uniquely describes and identifies that person[22].

Fingerprinting of course cannot only be used for the identification of persons. Parameter extraction for unique identification can also be applied to other

[21] [Mal+09], p. 31-33.
[22] [HK02], p. 1.

areas. Around the turn of the millennium, when digital audio players and formats became popular, researchers began to experiment with audio fingerprinting, a transfer of the fingerprinting concept to the field of digital audio. The areas for application of this research are manifold:

- **Name that tune:** One of the most popular uses are "name that tune"-services. If an unknown song is playing on the radio, users can record a few seconds of it with their mobile phone and send the data to the service. The service then calculates and matches the fingerprint and finally returns the name of the song[23].
- **Broadcast monitoring:** Radio stations have to pay royalties to artists and labels for every song they play. Audio fingerprinting can help to automatically keep track of all the played songs, which may greatly reduce the administrative work.
- **Detecting copyright violations:** Audio fingerprinting algorithms can help detecting music that is used illegally on file- and video-sharing platforms.
- **Music library organisation:** With audio fingerprinting, duplicates can be efficiently removed from music libraries.
- **Automated testing:** Although not many references exist, audio fingerprinting is also used for automated testing of audio software functionalities according to [SSR08].

One of the first and best known companies to develop and operate an audio fingerprinting service is Shazam[24]. Their audio search engine that identifies songs for its users is still widely popular after many years of existence.
There are several parameters that influence the quality of an audio fingerprinting algorithm[25].

- **Robustness:** The most important parameter is robustness. Due to the reasons explained in section 8.1, a song that is recorded for identification will most likely be different from the original. Loud voices might be audible when trying to identify a song with Shazam in a pub, and a broadband monitoring algorithm might be confronted with the same

[23][BKe+13], p. 2.
[24][Sha].
[25][HK02], p. 2.

song in many different file formats. A good fingerprinting algorithm should still be able to clearly identify the corresponding song.

- **Speed:** An efficient search strategy is also of great importance. Popular audio fingerprinting services maintain databases with millions of fingerprints. Nevertheless, finding a match in the database for a queried song should be fast to guarantee the best possible user experience.
- **Granularity:** A user looking to identify a song will most certainly not be able to record it completely the moment he hears it playing on the radio. Also, a radio station might decide to only play twenty seconds of a song. An audio fingerprinting algorithm should still be able to identify the song in order to provide an answer to its users or to calculate the royalties that have to be paid. Granularity describes how much audio data is necessary to successfully perform a query on the database.
- **Reliabillity:** Reliability describes how often a song is mistaken for another song in the database.

8.5.1 General audio fingerprinting framework

Since the development and release of the first audio fingerprinting services, many others have followed. Although they use different techniques to create and match fingerprints, their general framework and processes are very similar.

Figure 8.2 shows the general audio fingerprinting workflow. Audio files that should be identifiable (the reference files) are processed by the audio fingerprinting algorithm. The resulting fingerprints are then stored in a database together with additional meta information about the audio file (song title, artist, etc.). To identify an unlabelled, unknown audio file, the fingerprinting algorithm calculates the fingerprint of the unlabelled material and then queries the database to match the unknown fingerprint with the fingerprints in the database. If a fingerprint in the database and the unlabelled fingerprint fulfill a certain similarity criteria, the unlabelled file is identified as the audio file associated with the found database entry. The meta data of this entry is then returned as the result.

In summary, there are two relevant processes to every audio fingerprinting

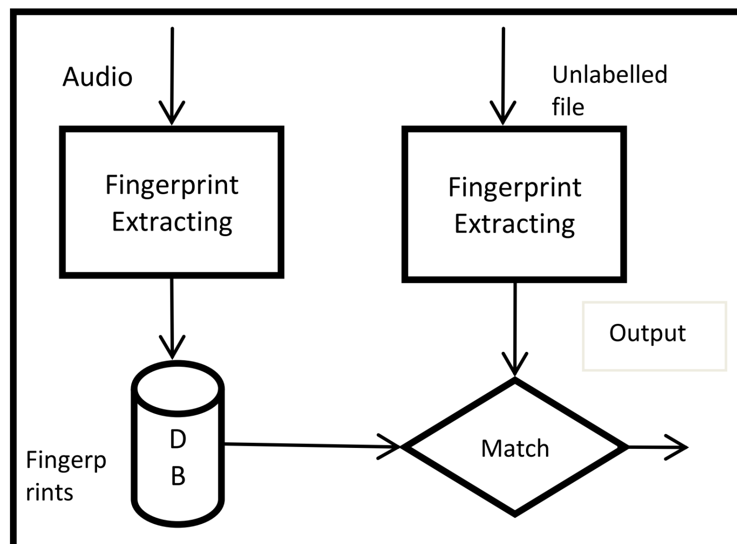

Figure 8.2: General audio fingerprinting processes[26]

service: the fingerprint extraction and the fingerprint matching. Those two processes can be further divided into sub-processes.

8.5.2 Audio fingerprinting framework for automated testing

When using audio fingerprinting for automated testing, some changes to the general framework are necessary. In the previous, general scenario, an unlabelled song has to be compared with potentially millions of fingerprints in the fingerprint database to identify the correct song. For the purpose of automated testing, the task is not to identify an unlabelled input file, but to compare an input file with exactly one reference file to determine whether they are the same or not. Therefore the fingerprinting process can be changed according to figure 8.3. First, a reference file is stored, that represents the correct, expected output of a test case. Every time the test case is executed, a test file is recorded. Then, fingerprints of both test and reference files are calculated. Those fingerprints are then compared with a similarity measure. If the similarity measure exceeds a previously defined

[26][BKe+13], p. 3.

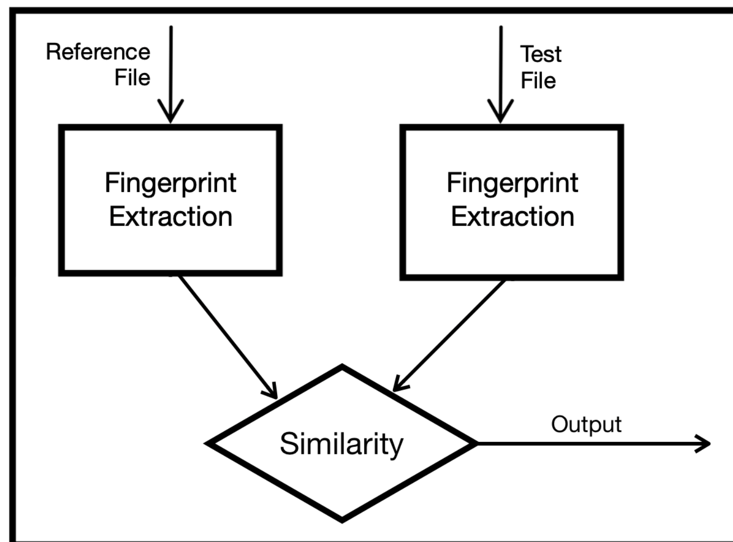

Figure 8.3: Audio fingerprinting process for automated testing

threshold, the files are considered to be the same. If not, the test case fails. The following sections describe the sub-processes of the fingerprint extraction and similarity measures that can be applied to the fingerprints. Since a fingerprint database and matching algorithm are not needed for automated audio testing, these components will not be discussed any further.

In the context of automated testing, audio fingerprinting can be seen as a form of parametric testing. As audio fingerprinting is a very distinct form of parametric analysis used to identify arbitrary audio files and because audio fingerprints consist of a very large number of different parameters, it is reasonable to discuss this topic separate from the parametric test approach discussed before.

8.5.3 Fingerprint extraction

Preprocessing In order to be able to compare and match fingerprints with one another, the audio data has to be converted to a common format before fingerprinting starts. Therefore, in the first preprocessing step, the audio

material is converted into a previously defined digital audio format. Usually, this is a raw, uncompressed format, for example Pulse Code Modulation (PCM) with 16 bits, one channel (mono) and a sample rate between 5000 and 44'100 Hz. After this, further preprocessing steps like filtering or amplitude normalization can be applied[27].

Framing and overlap Although audio signals are non-static signals, they can be considered as static over a short timespan. With this assumption, several consecutive samples are grouped into a frame. To minimize discontinuities at the beginning and end of every frame, a window function is applied. Consecutive frames are usually strongly overlapping which makes the resulting fingerprint robust to shifts or slight misalignments of the input data[28].

Linear transformation In a next step, a linear transformation is applied. In this process, feature vectors (the frames obtained in the previous step) are mapped to a new set of feature vectors. This process intends to remove redundancy and to represent the data in a form that brings to light important perceptual audio features[29]. Since the most important perceptual audio features can be found in the frequency domain, a spectral transform from time to frequency domain is usually chosen[30]. An often used transform for this purpose is the FFT.

Feature extraction The objective of this step is to gain some final acoustic vectors from the previously obtained time-frequency representation. In this process, the dimensionality of the feature vectors is further reduced. For an algorithm to be robust, the final acoustic vectors should diminish the distorting factors described in section 8.1 as much as possible and highlight the most distinct characteristics of the processed signal (the audio file or song that should be identified). A variety of different algorithms have been

[27] [Can+02a], p. 2.
[28] [Kha+14], p. 2.
[29] [Can+02a], p. 3.
[30] [HK02], p. 4.

implemented to solve this task. Many of them try to emulate the human auditory system. The human auditory system processes audio signals on their path from the ears to the brain in a way that makes them robust to noise and distortions[31] which makes those processing steps well suited for audio fingerprinting. Often, such processing measures start with the division of the FFT frames into a series of spectral bands that are linked to inner processes of the human auditory system, like the Bark or Mel scale. After that, certain features are extracted from each band. Such features include the signal energy of the bands, location of energy peaks inside bands, energy differences across bands or other power or energy measures like the spectral flatness measure[32].

Post-processing Finally, the postprocessing step converts the extracted features of every frame into a bit sequence. The bit sequence of one frame is called a sub-fingerprint. All sub-fingerprints combined form the final fingerprint which can then be stored in a fingerprint database[33].

8.5.4 Similarity measures

To identify an unlabelled file within a database of millions of songs, fingerprinting algorithms have to use highly efficient search strategies. If only the comparison of two fingerprints is required, as is the case for automated audio testing, the required algorithms are simpler. Two such similarity measures are implemented in [Lalb] and [KS] and are described in the following sections.

8.5.4.1 Bit error rate

[HK02] suggests to measure the similarity between two fingerprints by calulating the bit error rate (BER). The lower the BER between two fingerprints, the more similar they are. If the BER for two fingerprints is below a defined

[31] [Can+02b], p. 4.
[32] [Can+02a], p. 3.
[33] [Kha+14], p. 2.

threshold, they are considered to originate from the same audio material. The BER is typically used to measure the quality of a communication channel. It is calculated by the number of erroneous bits B_e divided by the number of total bits B_t received in a certain time frame[34].

$$BER = \frac{B_e}{B_t}$$

This measure can be applied to audio fingerprints by counting all erroneous bits between the test and reference fingerprint and dividing it by the total number of bits.

8.5.4.2 SimHash

SimHash is an algorithm used for similarity estimation of two different sets. It is used by Google to find duplicate or near duplicate web pages and was first described by Moses Charikar in [Cha02]. As the name suggests, SimHash creates similar hashes for similar input values, whereas a standard hash function creates a completely different hash for little changes in the input data. The following steps describe the process to derive a 32-bit SimHash[35]:

1. Initialize an integer array V of size 32 to zero. $V[m] = 0, \forall i$
2. Split the input data into features. In case of text input, a feature could be an individual word or a part of a word.
3. Use a standard hash function with a hash size of 32 bits to hash every single feature.
4. Repeat for every feature hash: If bit i is equal to 1, add 1 to $V[m]$. If bit i is equal to 0, subtract 1 from $V[m]$.
5. Calculate the SimHash as follows: $SimHash_m = \begin{cases} 1, \text{if } V[m] > 0 \\ 0, \text{if } V[m] \leq 0 \end{cases}$ where m denotes the m-th bit of the SimHash.

The similarity of two input sets can then be judged by the Hamming distance of their SimHash, the number of positions at which two bit-sequences differ.

[34] [IT].
[35] [Kel].

The smaller the Hamming distance, the bigger the similarity of two sets. Of course, the size of a SimHash can be freely chosen based on the used hash function and is not limited to 32 bits.
When calculating the SimHash of an audio fingerprint, every sub-fingerprint can be seen as a hashed feature. Performing step 3 is therefore not necessary. This means that step 4 is performed on and repeated for every available sub-fingerprint. To determine whether a test fingerprint matches a reference fingerprint or not, a Hamming distance threshold has to be defined.

8.6 Open source audio fingerprinting libraries

To make use of audio fingerprinting for automated audio testing in a project, an audio fingerprinting library or a remote audio fingerprinting API have to be integrated. Since the correct operation and connection to a remote service is beyond the control of the project's developers, this would add an additional factor of uncertainty. For this reason, only libraries that can be executed directly on the test device were considered for use in Pocket Code. Additionally, since Pocket Code is an open source project, the audio fingerprinting library of choice has to be an open source library as well.
Over the years, three open source audio fingerprinting libraries have emerged: Chromaprint [Lalb], pHash [KS] and Echoprint [The]. Because Echoprint does not seem to be maintained anymore, and only very little information about the theoretical background of the algorithm is available, only Chromaprint and pHash will be described in more detail in the following sections.

8.6.1 Chromaprint

Chromaprint is a audio fingerprinting library that is part of a complete open source audio fingerprinting service called AcousticID[36]. Besides Chromaprint, AcousticID also includes a crowd-sourced database of audio fingerprints and a web-service to easily search for matching fingerprints in the database.

[36] [Lala].

For the purpose of automated audio testing, only Chromaprint itself is of further interest. The following sections will give a short overview of the internal procedures of Chromaprint.

8.6.1.1 Preprocessing

In the preprocessing step, the incoming audio data has to be provided as 16-bit integer linear PCM. Chromaprint will not handle the conversion of other formats to PCM. The data can have any number of channels, as long as they are interleaved. This means that in the audio buffer, sample 1 of channel 1 is followed by sample 1 of channel 2 followed by sample 1 of channel 3 an so on. This pattern is then repeated for all the following samples. Once a correctly formatted audio buffer is provided to Chromaprint, the signal is converted to mono by taking the average value over all channels for every sample. Finally, the sample rate is converted to 11025 Hz[37].

8.6.1.2 Framing and overlap

Chromaprint uses a frame size of 4096 samples. With a sampling frequency of 11025 Hz, a frame therefore has a duration of 0.372 seconds. Consecutive frames are overlapping by two thirds, which means that every 0.124 seconds, a new frame begins.

8.6.1.3 Linear transformation

Chromaprint transforms each frame by applying a fast Fourier transform. The frame size $N = 4096$ and the sampling frequency $f_s = 11025Hz$ lead to a frequency resolution of $d_f = f_s/N = 2.69Hz$ and a maximum frequency of $f_{max} = f_s/2 = 5512.5Hz$. The FFT therefore spans a frequency spectrum of 0 - 5512.5 Hz.

[37] [Lal11].

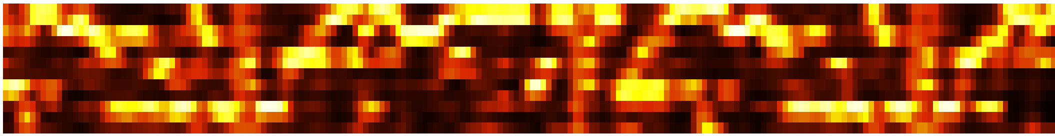

Figure 8.4: Chroma feature timeline of a song[38]

8.6.1.4 Feature extraction

Although Chromaprint computes a 4096-point FFT representing frequencies between 0 and 5512.5 Hz, only frequencies between 28 and 3520 Hz are used for the feature extraction. In a first step, the energy of every frequency sample of the FFT in this range is added to one of twelve bins. These bins, called "chroma features", represent the notes of a chromatic scale. In the simplest version, the energy of every FFT sample is added to the bin of the closest note. The energy of an FFT sample representing 450 Hz would be added to the bin of note A (440 Hz) as would an FFT sample of 900 Hz (which is one octave higher). In a more sophisticated version of the feature extraction, the energy of an FFT sample gets added to two bins. The energy of the 450 Hz FFT sample mentioned before would be divided and partially added to the bins of note A and A♯ because it lies between those frequencies. The bin of note A would be assigned a bigger share of the energy because 450 Hz lies closer to an A (440 Hz) than to an A♯ (466 Hz). With this process, a feature vector (the chroma features) is extracted for every FFT frame. Finally, some additional filtering and normalizing is applied to every feature vector. A visualization of this process can be seen in figure 8.4. It shows how the 12 bins of the chroma features evolve over the span of a whole song. The color gives an indication of the energy content in the individual bins of the chroma features (with white indicating high and black indicating low energy content).

8.6.1.5 Post-processing

The chroma feature representation is already pretty robust against distorted audio data and can highlight important similarities and differences between

[38] [Lal11].

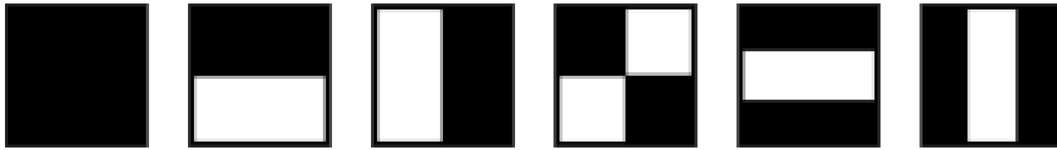

Figure 8.5: Basic patterns used in the filter window[42]

audio files. However, simply comparing these representations to determine the similarity between songs (for example by measuring correlation) might still be too inaccurate and slow. [KHS05] therefore recommends to process the chroma feature timeline with a set of filters whose responses are robust to distortions and preserve the important information of the processed audio data[39]. Chromaprint does this by moving a window of size 12 x 16 over the chroma feature timeline from left to right. The window therefore always includes 16 consecutive chroma feature vectors. At each position of the window in the timeline, a set of 16 predefined filters is applied. Every filter consists of the six basic patterns shown in figure 8.5[40], which can be arranged at any positions and in any size within the 12 x 16 sized filter window. The arrangement of the patterns in Chromaprint was done by a machine learning process. These six patterns were chosen to capture important characteristics of the chroma feature timeline, like energy differences in neighbouring chroma feature bins or energy differences across time within a particular chroma feature bin[41]. For each of the 16 filters, the energy contents inside the black and white areas are subtracted from each other and the result is mapped to a two bit number. The resulting bit sequence of length 32 (2 bits for each of the 16 filters) represents a sub-fingerprint. After this, the window is moved one step further to the right to create the next sub-fingerprint. The collection of all sub-fingerprints finally represents the complete fingerprint.

[39] [KHS05], p. 2.
[40] [Lal11].
[41] [KHS05], p. 2.
[42] [Lal11].

8.6.1.6 Similarity measure

Chromaprint supports the SimHash algorithm with 32-bit length to determine the similarity of two fingerprints. The Hamming distance of two fingerprints can be in the range of 0 - 32 and is small for similar fingerprints and big for different fingerprints. If the Hamming distance is bigger than 15, the corresponding audio files are considered to be completely different.

8.6.2 pHash

pHash stands for perceptual hash and is another fingerprinting library. A perceptual hash is defined as "a fingerprint of a multimedia file derived from various features from its content"[43]. Like audio fingerprints, perceptual hashes of multimedia files with similar features should be close to one another. pHash was mainly developed for the fingerprinting of visual content like images or videos, but does also contain fingerprinting services for audio and text. It can be used for the purpose of copyright protection, similarity search or digital forensics. In this case, only the audio fingerprinting service of pHash is of further interest. When analysing the audio fingerprinting algorithm of pHash, it becomes apparent that it is a slightly adapted implementation of the *Philips Robust Hashing* (PRH) algorithm[44], an algorithm developed by Dutch technology company Philips.

8.6.2.1 Preprocessing

pHash can handle multiple audio input formats. The audio data can either be provided in an uncompressed PCM format or as MP3. The input data is then converted to 32-bit floating point linear PCM. Additionally, a conversion to mono is performed in the same way as in Chromaprint. The sampling rate can be chosen freely. Of course, it is important to always use the same sampling rate to get comparable results. The original PRH algorithm works with a sampling rate of 5000 Hz[45].

[43] [KS].
[44] [HK02].
[45] [HK02], p. 4.

8.6.2.2 Framing and overlap

Like Chromaprint, pHash uses a frame size of 4096 samples. With its big overlap of 31/32 (almost 97%), the algorithm is very robust against timing differences.

8.6.2.3 Linear transformation

Before the transformation, each frame is weighted with a hamming window (a bell shaped curve). Then, a 4096-point FFT is performed. The frequency resolution and the maximum sampling frequency depend on the chosen sampling rate and can be calculated as shown for Chromaprint. In contrast to pHash, PRH uses a 2048-point FFT.

8.6.2.4 Feature extraction

pHash creates a 32-bit sub-fingerprint for every frame. First, the FFT of every frame is divided into 33 non-overlapping frequency bands. Then, the energies of all FFT samples in a particular band are added up. PRH operates with 33 bands between 300 and 2000 Hz and a logarithmic spacing. The logarithmic spacing is chosen because the human auditory system also operates on logarithmic bands called the "Bark scale"[46]. Again, pHash makes some changes to the original PRH algorithm. Instead of 300 - 2000 Hz, the 33 bands lie between 300 and 3000 Hz and instead of a logarithmic function an inverse hyperbolic sine is used for the spacing of the bands (which behaves very similar to a logarithm).

8.6.2.5 Post-processing

Tests have shown, that the sign of energy differences along the time and frequency axis is very robust against distorting factors. Due to this, all bits

[46][HK02], p. 4.

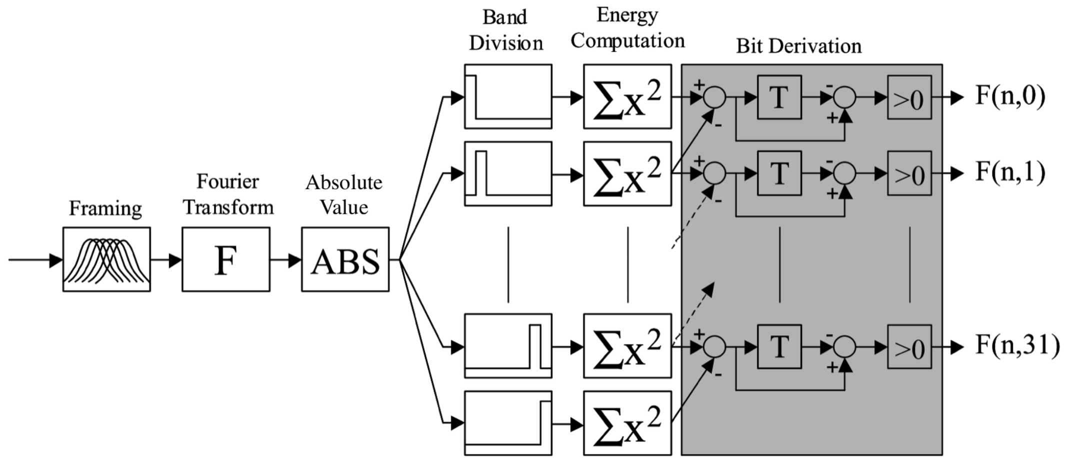

Figure 8.6: Flowchart of the pHash and PRH fingerprinting process[47]

of the final sub-fingerprints are calculated using these differences with the following formula:

$$F_{n,m} = \begin{cases} 1, \text{if } E_{n,m} - E_{n,m+1} - (E_{n-1,m} - E_{n-1,m+1}) > 0 \\ 0, \text{if } E_{n,m} - E_{n,m+1} - (E_{n-1,m} - E_{n-1,m+1}) \leq 0 \end{cases}$$

$F_{n,m}$ denotes the m-th bit of of the n-th sub-fingerprint, while $E_{n,m}$ stands for the energy in the m-th band of the n-th frame.
The whole process from audio data to sub-fingerprint is depicted in figure 8.6. T represents a delay element.

8.6.2.6 Similarity measure

pHash calculates the similarity of two files by measuring the BER which is then transformed to a confidence score with the following process:

1. Initialize the confidence score with a value of $p = 0.5$.
2. Divide the fingerprints into a series of blocks (a block size of 256 sub-fingerprints is recommended) and calculate the BER for each aligned pair of blocks.

[47][HK02], p. 4.

3. For every calculated BER that is smaller than or equal to a pre-defined threshold calculate: $p = p + (1 - BER)/M$ where M is the total number of block pairs. In this case the confidence score increases.
4. For every calcluated BER that is bigger than a pre-defined threshold calculate: $p = p - (1 - BER)/M$. The confidence score decreases.

The final confidence score is a value between zero and one with 0.5 being the threshold. Audio files with a confidence score above 0.5 are considered to be perceptually equal. pHash repeats the above procedure for different time offsets between the two fingerprints. The results is a vector of confidence scores, each entry representing the confidence score of a specific time offset. The highest value in the array is then taken to determine the perceptual equality of the two audio signals. This is done to account for possible time shifts between the signals or for the case, that the unknown signal is only a small excerpt of the reference signal.

9 Automated audio testing in Pocket Code on iOS

To properly test the functionality of the new audio engine, many new automated tests have been added during the redesign process. All tests were written with Xcode's test framework in combination with Nimble (see chapter 3). Tests were written with the good test properties in mind, making frequent use of test doubles and the techniques for their manual creation and maintenance. To cover as many of Pocket Code's audio specifications as possible with automated tests, three types of tests have been implemented: conventional unit tests, bit-exact tests and fingerprinting tests.
The fingerprinting tests clearly fall into the domain of integration testing, as they execute many units in combination, make use off external resources and take a long time to run. Bit-exact tests however, are somewhere between unit and integration tests. Although they also test the collaboration of a (much smaller) set of units, they execute much faster and in a more isolated environment. Fingerprinting tests, a distinct and complex form of parametric tests, were chosen over conventional parametric tests because they allow the use of arbitrary input signals. Because of that, they can theoretically be used for a much broader spectrum of test cases. However, as became apparent during test development, the complexity also comes with some drawbacks. The following sections will discuss how unit, bit-exact, and fingerprinting tests have been implemented in Pocket Code.

9.1 Unit testing

As described in section 1.1, unit tests intend to verify the logic, functionality and structure of isolated and small units of code. As such, the setup and

content of unit tests varies from case to case and depends heavily on the SUT. Generalising the functionality of Pocket Code's unit tests, or highlighting individual unit tests is therefore not possible or reasonable. However, some of the unit tests written for Pocket Code's audio engine follow a specific pattern and can therefore be divided into several categories. They have proven to increase the reliability of the audio engine's code without having to directly verify the audio output of the engine. The different types of implemented unit tests are discussed below.

Testing the values of node parameters Some of *CBAudioEngine*'s interface methods, like *changeVolumeBy()* or *setEffectTo()*, modify the values of certain node parameters. The verification of correct behaviour for those methods is done by checking the values of node parameters after execution of the respective interface methods. To verify the correct functionality of *changeVolumeBy()* for example, the unit test sets an initial value for the volume parameter of a subgraph output mixer, calls the *changeVolumeBy()* method for the according PCObject, and finally checks whether the volume of the mixer has been set to the correct value. To ensure correct behaviour over all possible input values, such tests are usually partitioned into different equivalence classes. Equivalence classes are ranges of input values that are expected to be processed in the same way by the SUT. Each equivalence class usually has at least one associated test case[1]. For the above example, there would be three equivalence classes. One class represents the inputs that set the volume to a value within the allowed volume range, and the other two classes represent the inputs that set the volume to a value above or below the allowed range.

Testing the transmission of interface calls In contrast to interface methods whose functionality can be verified by a return value or change in state, other interface methods simply delegate a piece of work to a target node. Unit testing such interface methods is not as easy. An example for this scenario is the *playSound()* interface method, which forwards the command to the appropriate audio player of a specific PCObject. Verifying that this call starts the audio playback of the correct audio player without having access

[1] [Bur03], p. 67.

to the audio output is nearly impossible. It is possible to verify however, that such interface calls lead to the right method calls on the target node, or in other words: It is possible to test the transmission of an interface call to the target node. This principle is essentially the spying principle discussed in section 2.1.4, where test spies verify the invocation of certain methods with defined parameters. In Xcode, this can be done by injecting `XCTestExpectations` into the class of the target node. Once the expected method has been called on that node, a fake implementation of this method fulfills the expectation and thereby informs the test case that the interface call was successfully transmitted to the target node. An example of this approach is given in [Mis17][2].

Testing the graph structure A number of unit tests verify the correct structure of the audio graph after the audio engine has been initialized or certain interface methods have been called. This is simply done by checking whether the correct objects are present in the `CBAudioEngine` or `CBSubgraph` classes. A structural unit test could for example check if the correct subgraph object exists in the `CBAudioEngine` subgraph table after calling the `playSound()` interface method for a certain PCObject. It could also test if a `CBSubgraph` object contains an instance of a sampler after invoking the `playNote()` interface method. Structural unit tests can also include checks that verify whether the nodes of a graph are connected properly to each other.

Verifying behaviour using indicator data When the correct behaviour of the audio engine cannot be verified with first-hand data, some unit tests use data that provides an indirect indication of the engine's behaviour. Test cases that do so, are the unit tests for the speech synthesizer. These tests verify whether the speech synthesizer is correctly speaking after calling the according interface method by checking a boolean `isSpeaking` flag on the synthesizer object. The `isPaused` flag on the other hand indicates, whether the speech synthesizer was successfully paused or resumed when calling the engine's `pause()` or `resume()` interface methods. The same

[2][Mis17], p. 86.

principle can be used for player or effect nodes, which have the `isPlaying` or `isStarted` flags respectively.

9.2 Bit-exact testing

Although bit-exact testing of audio data is often unfeasible due to small irregularities introduced to the audio signal (see section 8.1), there is still a way to make bit-exact testing work in Pocket Code for a certain type of audio tests. The reason for this is a feature called "offline manual rendering" which is part of the AVAudioEngine framework and was introduced by Apple with iOS 11[3]. This chapter will introduce the offline manual rendering functionality first, then outline the test scenarios where it can be applied, and finally describe how bit-exact testing has been implemented in Pocket Code.

9.2.1 Offline manual rendering

Offline manual rendering is an approach of operating an audio graph outside of a real-time context, making it possible to eliminate some of the difficulties and limitations of real time audio processing. When operating an audio graph in offline rendering mode, it is completely disconnected from the device's audio input or output hardware. This means that the graph's input and output nodes are not connected to any hardware device (microphone, speakers etc.). Audio data is therefore not rendered to the device's audio output but to the application which is operating the audio graph. The application is not only responsible of pulling the rendered audio data from the graph's destination node, but is also responsible of supplying the input data to the graph's source nodes. It is therefore driving all input and output operations of the audio graph. The retrieved and processed output audio data can then be saved to an audio file in the application's memory or on the device storage, from where it is accessible for further use[4]. The fact that the

[3] [Appk].
[4] [Appl].

audio engine is not operating under any real-time constraints has several advantages. First of all, as the audio engine is not dependent on any input or output devices, the audio data can usually be rendered and saved to an audio file much faster than real-time. Additionally, the rendered audio data gets predictable and deterministic. In a real-time context, the input audio data of a source node might not always be instantly available. The loading times of audio files from the storage can sometimes take a little longer or the audio graph might even be dependent on receiving the audio data from another system. In a real-time context, processing the same audio data therefore always leads to slightly different results. In an offline context, nodes can block the render thread and wait until all necessary data is available before they continue processing because the audio data does not have to be played back immediately. This approach always leads to identical output audio data as long as the graph is not using any time variant processing nodes. Apart from time varying processing nodes, the only factor that might still lead to unpredictable results is inconsistent floating point arithmetic as mentioned in section 8.1. However, as this version of Pocket Code is natively developed for iOS, it is safe to assume that all developers are using the same 64-bit processor architecture and the same compiler on their development machines. Floating point inconsistencies are therefore not an issue in this case. Apple itself only mentions floating point inconsistencies between their current Intel and former PowerPC computers, whose production has been discontinued in 2006[5].
Offline rendering can be used in different scenarios, such as the post processing and mixing of audio files or to use higher quality, resource intensive algorithms that would not be feasible to use in real-time. Apple also specifically mentions the debugging and testing of audio engine setups as a possible use case[6].

Using the offline manual rendering functionality of AVAudioEngine is relatively easy. First, the offline manual rendering mode has to be activated. While an audio engine is in the offline mode, its graph cannot be used to produce real time audio data. After that, the audio data that has to be processed must be scheduled for playback on the responsible source nodes.

[5] [Fie10].
[6] [App17].

Usually this involves scheduling the playback of audio files on audio player nodes at a certain time. Once this is done, rendering can begin. In this process, the application is continuously and repeatedly pulling a configurable amount of rendered audio data from the audio graph's output node into a buffer until all data has been rendered. The audio data that is rendered into the buffer is then usually passed to an audio file for temporary storage, but the application could also use it in other ways.
AudioKit also offers an offline rendering feature which makes use of AVAudioEngine's offline rendering implementation. Once more, AudioKit makes its usage even simpler as it takes care of activating the offline rendering mode, pulling the data from the graph's output, buffering the audio data and storing it in an audio file in one simple method call shown in listing 9.1.

```
func renderToFile(audioFile: AVAudioFile,
                   duration: Double,
                  prerender: (() -> Void)? = nil,
                   progress: ((Double) -> Void)? = nil)
```

Listing 9.1: Offline rendering with AudioKit

This method takes several parameters: An audio file where the rendered data is written to, the duration of the rendered audio data, a pre-render closure as well as a progress closure. The pre-render closure is called before rendering starts and is mainly used to start or schedule the playback of audio players or set some node parameters to the desired values. The progress closure on the other hand is called while rendering and can be used to report and display render progress[7].

9.2.2 Using offline manual rendering in Pocket Code

As described in the previous section, Apple specifically mentions testing and debugging as a possible area of application for offline rendering. Indeed, with it's predictable and stable audio output, offline rendering allows to perform a certain type of audio tests: tests that verify static setups of an

[7][Prob].

audio engine, or more specifically static setups of an audio graph. This verification is done by comparing the rendered audio file to a reference file that has been created earlier. If both files are identical the audio graph has been set up correctly and is generating and processing data as planned. As the audio graph is completely isolated from all audio hardware as well as external commands and instructions during offline rendering, changes to the configuration of the audio graph have to be made prior to rendering and cannot be done during rendering. The output volume parameter of a mixer node for example cannot change in an audio file that has been rendered offline, but can be changed at any time when rendering in real-time. The only action that can be scheduled for execution during offline rendering is the playback of audio player nodes. This means, that audio players do not have to start playing at the very beginning of the rendered audio data. If desired, they can be scheduled to start playback at a later time during the rendering process. Offline rendering currently only works with audio player nodes and not with source nodes that are controlled by MIDI signals. An automated, bit-exact test with offline manual rendering can be performed in three steps with AudioKit:

1. First, the audio graph that has to be tested is built and configured and audio players are initialized. This can either be done by separately connecting and configuring all nodes inside the test case or by calling the code responsible of building and initializing the graph structure in the SUT.
2. AudioKit's `renderToFile()` method is called and a closure is provided that starts or schedules the playback of all involved audio players.
3. The rendered audio file is compared with a previously generated reference audio file. If the two are identical, the audio graph has been setup correctly and processing worked as intended. If not, the graph is not configured as it should be.

This test procedure can be directly applied to Pocket Code. The following description will give a quick overview of how the procedure has been implemented in Pocket Code by explaining the example in listing 9.2.

```
public func testSetVolumeToExpectOnlyPlayerOfObject1ToChangeVolume() {

    // Calling interface methods to set up the audio graph's structure
    audioEngine.playSound(fileName: "sound1.mp3", key: "Background",
                          filePath: nil, expectation: nil)
    audioEngine.playSound(fileName: "sound2.mp3", key: "Object1",
                          filePath: nil, expectation: nil)
    audioEngine.setVolumeTo(percent: 40.0, key: "Object1")

    // Retrieving the audio players and render audio data to an audio file
    let player1 = self.audioEngine.subgraphs["Background"]!.audioPlayerCache
                  .object(forKey: "sound1.mp3")!.akPlayer
    let player2 = self.audioEngine.subgraphs["Object1"]!.audioPlayerCache
                  .object(forKey: "sound2.mp3")!.akPlayer

    let renderDuration = 2.0
    audioEngine.renderToFile(tape, duration: renderDuration) {
        player1.play()
        player2.play()
    }

    playRenderedTape(tape: tape, duration: renderDuration)

    // Calculating hash and comparing with reference hash
    let tapeHash = getTapeHash()
    expect(tapeHash) == "2895a0f3f1e84f972852a146eaad3cf4"

}
```

Listing 9.2: Bit-exact test case in Pocket Code

Setting up the graph structure A bit-exact test in Pocket Code always begins by setting up the audio graph structure which has to be tested. Although this could be done by separately instantiating and connecting all nodes in the test method, this would not make much sense. The objective of a bit-exact test in Pocket Code is to verify whether `CBAudioEngine`'s interface methods construct the audio graph in the correct way when called. The audio graph in bit-exact Pocket Code tests is therefore built by calling all necessary interface methods of `CBAudioEngine`. In listing 9.2, this can be seen at very beginning of the test method where the `playSound()` method is called for two different PCObjects and with two different audio

files. Additionally, the volume for one of the PCObjects is set. This test case therefore verifies that a change in volume only affects players of the PCObject that executed the *Set volume to* brick and not players from other PCObjects as well. In other words, it verifies that the audio graph has the correct setup after multiple PCObjects have executed *Start sound* bricks and one PCObject executed a *Set volume to* brick. The `playSound()` method is not called to actually play back the audio file, but to instantiate a new audio player and connect it to the audio graph. For this reason, although not visible in the code example, `playSound()` has been substituted by a fake implementation in Pocket Code's bit-exact tests, so that it does not start the playback of the audio file and only integrates the audio player into the audio graph.

Rendering audio data to a file Now that the audio graph setup has been completed, audio data can be rendered to an audio file. To do so, an audio file, called "tape" in listing 9.2, is passed to AudioKits `renderToFile()` method together with the render duration and a pre-render closure. The pre-render closure starts both previously created audio players right before rendering begins. Before calling `renderToFile()`, the audio players have to be retrieved from the audio engine's subgraphs. The tape variable is a global variable, which is why its instantiation is not visible in the test method. As offline rendering is not supported for source nodes controlled by MIDI signals, Pocket Code's drum and instrument samplers cannot be tested with offline rendering. The pre-render closures of Pocket Code's bit-exact tests can therefore only ever start audio players and not any other source nodes.

Comparing the rendered audio file with a reference file The rendered audio file can now be compared with a previously generated reference file. To simplify this process and avoid storing the complete reference audio file, this is done by hashing the contents of the rendered audio file and the reference audio file (see section 8.3.1). The hashes are then compared in the very last line of the test method. The hash generation has been extracted into the `getTapeHash()` method which is not shown in further detail. This method retrieves two UInt8 arrays from the rendered audio file,

which represent the audio data from the left and right channels. They are then fed to an MD5 algorithm to calculate the final hash value. Hashing is performed by a library called CryptoSwift[8], which provides a collection of many different cryptographic algorithms and an intuitive, easy to use interface. Determining the reference hash can be done when running the test case for the first time. The calculated hash of the rendered audio file is printed to the console and can be copied to the test case from there. The correctness of the reference hash and the rendered audio file can be verified by listening to the audio file which is played back within the `playRenderedTape()` method. This method is also helpful in case a test case fails, as the audio output might give an indication about the cause of the failure. If the playback of the rendered audio file during test execution is not desired, `playRenderedTape()` can also be removed or commented out.

9.3 Integration tests with audio fingerprinting

The bit-exact testing approach explained in the previous section has proven to be an effective, fast and reliable way to test the real audio output of Pocket Code's audio engine. Unfortunately, this approach can only be used to test a limited set of scenarios dealing with statically configured audio graphs.
However, many of the audio engine's specifications describe a sequence of events and how these events influence the audio output. The events are either triggered by executed bricks or users interacting with a running PCProject. Such scenarios are no longer dealing with static configurations, but with dynamic processes. To test whether these processes affect the audio output in the desired way, the audio output has to be analysed over the whole timespan of the process.
A simple example of a sequence of events that has been defined as the desired behaviour of Pocket Code in section 6.1.1 is the following:

1. A PCObject executes a *Start sound* brick.

[8] [Krz].

2. An audio player starts playing the desired PCSound and keeps on playing for a few seconds without interruption.
3. The same PCObject executes another *Start sound* brick with the same PCSound after a few seconds.
4. The audio player stops playing the previously started PCSound and starts playing it from the beginning again.

Testing this dynamic process with offline manual rendering is impossible. The renderer only knows the structure of the audio graph but is completely unaware of the external logic, events and commands that operate it. The only way to test the audio output of such a scenario is by actually running the sequence of events in real time, recording the audio output and verifying its correctness.
A real-time test scenario as the above no longer examines a single isolated unit under test, but rather tests the integration of the audio engine as a whole. As mentioned in section 1.1.2, integration tests come with a few difficulties as they often make use of operations like threading or file system access. The exact timing of such operations varies each time, as it depends on external factors like CPU load or the type of hardware used.
Unfortunately, time delay is one of the factors discussed in section 8.1 that makes bit-exact testing of audio material very difficult. In order to still allow the verification of audio output of dynamic processes, a test architecture based on audio fingerprinting was set up and evaluated. The architecture of this setup and the findings of the evaluation are discussed below.

9.3.1 Capturing the audio output of test scenarios with audio taps

A prerequisit for being able to analyse the real-time audio output of Pocket Code is the ability to capture the generated audio data. The AVAudioEngine framework provides a technique called "audio tapping" to do just that. An audio tap is an object which can be installed on the output buses of arbitrary nodes within the audio processing graph. Once the graph starts processing audio data, the audio tap pulls this data from the buses where it is installed, copies the data into a buffer and finally returns the buffer to the application

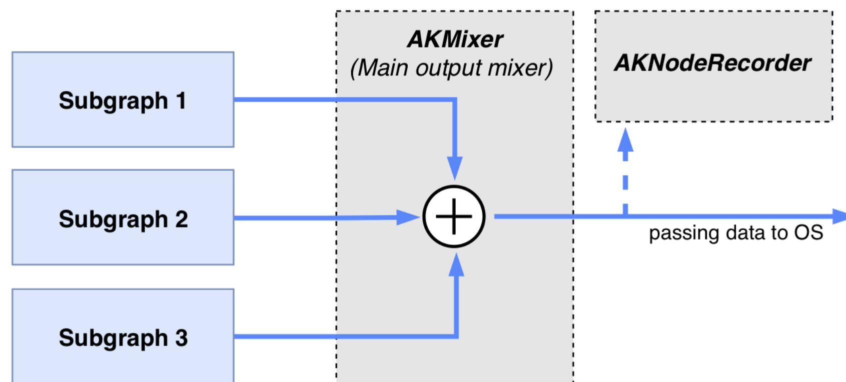

Figure 9.1: A node recorder installed at the output bus of the graph's main output mixer

within a callback block[9]. This callback is called repeatedly whenever the buffer is full and allows developers to use the captured audio data for their specific purposes.
AudioKit uses AVAudioEngine's audio taps to build its own implementation of the same functionality called *AKNodeRecorder*. Just like audio taps, *AKNodeRecorders* can be installed at the output buses of nodes, but come with a simpler interface. *AKNodeRecorder* does not expose the callbacks that contain the captured audio buffers. Instead, it aggregates the data of these callbacks internally and saves it to an audio file[10]. Developers simply have to pass a reference of an empty audio file of type *AKAudioFile* to the node recorder, start the node recorder and stop it again when the desired amount of data has been captured. Once finished, all the captured audio data is contained within the audio file. This procedure spares developers from manually handling and assembling small chunks of audio data in many consecutive callbacks.

To create audio fingerprinting tests in Pocket Code, an *AKNodeRecorder* is installed at the output bus of the graph's main output mixer (see figure 9.1) which ensures that the audio data from the entire graph is captured. The node recorder could theoretically be installed at any other location in the graph if a specific test scenario would require to do so.

[9] [App14b].
[10] [Proa].

9.3.2 Choice of the audio fingerprinting framework

To create an audio fingerprinting test architecture for Pocket Code, the two frameworks pHash an Chromaprint, both discussed in section 8.6, were originally considered. The final choice was made in favour of Chromaprint and was not based on the specific implementation details of the two algorithms but on a more practical reason: As both frameworks are written in C++, integrating those frameworks into an iOS or macOS application from scratch would have taken a considerable amount of time. Additionally, for the fingerprinting tests to run properly, both frameworks need additional configuration and setup code, which would have required even more efforts. However, the heavy lifting for a Swift integration of Chromaprint has already been done by another open source project[11]. This project takes care of integrating the Chromaprint library into an Xcode project, loading, decoding and reading audio files, feeding the audio data to the chromaprint algorithm, and coordinating all chromaprint method calls needed to retrieve the final fingerprint. The code of this project was used as a template for Pocket Code's own integration of Chromaprint with a few changes and amendments.

9.3.3 Creating and running audio fingerprinting tests in Pocket Code

With the help of Audio Kit's node recorder and the implementation of Chromaprint into the Pocket Code test environment, all preconditions to create audio fingerprinting tests in Pocket Code are met. To simplify the process of writing new fingerprinting tests, a small test framework was written that minimizes boilerplate code, and standardizes the creation and execution of new test cases. The following discussion explains all the steps involved in this standardized test execution procedure.

[11] [Dia16].

9.3.3.1 Audio fingerprinting test procedure

The procedure of running an audio fingerprinting test case in Pocket Code can be divided into several different steps which are always executed in the following order:

1. **Initialising the audio engine:** First, an object of type `CBAudioEngine` is initialized, which contains the audio graph's main output mixer node. An `AKNodeRecorder` is then added at the output bus of the main output mixer node to record all the audio data that is produced by the audio engine.
2. **Loading a PCProject XML file:** In a second step, the test case loads the test scenario, which is contained in an XML file and stored on the test device. Every PCProject is saved in an XML format which describes the structure of the PCProject (PCObjects, scripts and bricks defined within the PCProject) and the resources that are used (audio files and images). The test scenario should define a sequence of bricks that are executed during the test and trigger the audio output that has to be verified with the fingerprinting algorithm.
3. **Creating an executable PCProject:** Before being able to run a PCProject, the information found in the XML file has to be extracted and transformed into an executable PCProject. To represent a PCProject in the running Pocket Code application, Pocket Code uses a a class called `CBScene`, which is a class that contains all the instructions, resources and the runtime needed to run a PCProject. In this step, the information found in the XML file is parsed and used to build an executable `CBScene` object containing all the relevant data and logic.
4. **Starting the PCProject and the node recorder:** Now that the `CBScene` object is initialized, the PCProject can be executed. This is simply done by calling the `startProject()` method of `CBScene`. Before starting the PCProject, the node recorder has to be activated by calling its `record()` method.
5. **Waiting for a specified amount of time:** As the execution of a PCProject inside a test case is an asynchronous process, the test case has to wait until the PCProject has executed all bricks that are relevant for the test scenario. To do so, the test case is paused for a specified amount of time, waiting for the PCProject to execute its scripts while the node

recorder is recording the audio output. The test case is paused by running the main thread's run loop. A run loop is an event processing loop that exists for every thread. It is an infinite loop that is waiting for certain events and then processes and evaluates these events by executing event handlers[12]. The test procedure for audio fingerprinting tests doesn't require event processing but uses a run loop due to its capability to pause the test case for a desired amount of time without blocking the thread it is running on. This is important because some of the operations that are performed during the PCProject execution run on the same thread as the test case itself and would therefore be blocked if the test case's thread was suspended completely.

6. **Stopping the PCProject and the node recorder:** After the run loop has been running for the specified amount of time, it automatically stops running and the execution of the test case therefore continues. The test case has to do two things once it continues executing: stopping the PCProject and stopping the node recorder. This is done by calling the `stop()` method of the node recorder and the `stopProject()` method of `CBScene`.
7. **Calculating a SimHash:** Now that the PCProject has finished executing and the node recorder has stopped recording, the recorded audio data can be fed to the fingerprinting algorithm of Chromaprint. The result is a 32 bit SimHash, which is Chromaprint's default similarity measure (see section 8.5.4.2).
8. **Comparing the calulated SimHash with the reference SimHash:** Finally, the resulting SimHash is compared with a reference SimHash whose value was determined when creating the test case. If the Hamming distance of the two SimHashes is below a certain threshold, the test case fails.

9.3.3.2 Creating a new audio fingerprinting test

Audio fingerprinting tests are always implemented in a similar manner. An example for a test case in Pocket Code can be seen in listing 9.3. At first, the binary reference SimHash is defined. In a next step, the test case loads

[12][Appo].

an XML file which contains a PCProject structure and builds a `CBScene` object with the information from the file. All this happens inside the `createScene()` method of the small audio fingerprinting test framework that was created for Pocket Code. In the next line, the test case calls the `runAndRecord()` method, which is also part of the audio fingerprinting test framework. It is responsible of starting the node recorder and the PCProject, pausing the test case by activating the run loop, and stopping the node recorder and the PCProject once the run loop has finished running. This method also returns the recorded audio data to the `recordedTape` variable. Once the recorded audio data is available, the test case calls the `calculateSimilarity()` method, which feeds the audio data to Chromaprint and calculates the similarity between the calculated SimHash and the reference SimHash. The similarity is calculated by dividing the number of matching bits between the two SimHashes by 32, the total number of bits. The last line of code then asserts that the calculated similarity is greater than a defined threshold. A reasonable initial similarity threshold could be between 85% and 90%. If this proves to be too high, it can then be lowered accordingly.

Apart from writing the test method's code, two other things have to be done to finish the creation of a test case: creating an XML file which describes the PCProject that has to be executed and determining the reference SimHash. The easiest way to create the XML file is by running Pocket Code in the Xcode simulator and creating a PCProject with the desired structure, just like a normal user of the app would do on a real device. When interacting with Pocket Code in debug mode, the XML structure gets printed to the console with every change that is made to a PCProject. It can then be stored in an XML file within Pocket Code's test target. The only thing that might have to be changed in the final XML file is the path to the audio files that are used by the PCProject. Those files also have to be present in Pocket Code's test target.
To determine the reference SimHash, the test case has to be executed once. After execution, the calculated SimHash will be printed to the console and can be copied to the variable which holds the reference SimHash in the first line of the test case. To make sure that the audio output used to create the reference SimHash is correct, the `muted` flag of the `runAndRecord()` method has to be set to false. This way, the sound will be played during the

```
func testSomePocketCodeAudioScenario() {

    // Test setup
    let referenceSimHash = "01100100000011101001101010100100"
    let scene = self.createScene(xmlFile: "FileNameOfTestProject")

    // Run the program and record the audio output
    let recordedTape = self.runAndRecord(duration: 3,
                                         scene: scene,
                                         muted: true)

    // Calculate and verify the similarity
    let similarity = calculateSimilarity(tape: recordedTape,
                                referenceHash: referenceSimHash)
    expect(similarity) >= 0.85

}
```

Listing 9.3: Audio fingerprinting test case in Pocket Code

test execution and can be verified for correctness by listening to it. This flag is also helpful to find the cause of failure when a test case fails.

9.3.4 Findings from using audio fingerprinting in Pocket Code

After applying audio fingerprinting tests to Pocket Code and testing several features of Pocket Code's audio engine with Chromaprint, a few advantages, drawbacks and things to consider became apparent that are worth mentioning and keeping in mind.

Advantages The small audio fingerprinting framework that was built around Chromaprint can be used to test a very broad range of scenarios

with a consistent test procedure. Most audio scenarios that can be created by using Pocket Code are theoretically testable with Pocket Code's audio fingerprinting framework. The only exceptions are PCProjects that make use of TTS functionality or involve user interactions (*When tapped* or *When screen is touched* bricks) as the speech synthesizer output cannot (yet) be recorded and user interactions cannot be simulated during test execution. Another advantage is the simplicity of writing new tests. Creating a new audio fingerprinting test requires no more than five lines of code, the creation of an XML file and the determination of a reference SimHash. It is easy enough to be done by people that are not that skilled in software development or automated testing.

Limitations Audio fingerprinting tests also comes with a few difficulties and drawbacks. As the recorded audio data varies a little in each test run for the same test case, the calculated SimHash can vary too. This is why each test case checks whether the similarity between the calculated and the reference SimHash is bigger than a predefined threshold rather than checking whether the two SimHashes are exactly the same. Defining the appropriate similarity threshold for each test case can be a little difficult as the threshold depends on the setup of the PCProject, the audio data used within these PCProjects and the hardware used to run the tests. While the calculated similarities of audio fingerprinting tests with a correct audio output might never fall below 90% on the machine where the tests were created, the similarity can be lower on other machines, as the timing for executing a PCProject can be slightly slower or different.
Another disadvantage is, that it is impossible to draw any conclusions from a SimHash about the nature of the recorded audio data and vice versa. Chromaprint's post processing step and the SimHash algorithm alter the audio data in a way that is very intransparent for people not highly familiar with these processes. Due to this, it can be hard to choose the ideal audio files that make the test cases and similarity thresholds as robust as possible. A test case is robust if the SimHashes of failed and successful test outcomes are as different as possible. This makes it easy to set a suitable similarity threshold. If the SimHashes of failed and successful test outcomes are too close together, false positive or false negative test results will occur more often.

A better approach than comparing SimHashes might be to stop the fingerprinting process after the feature extraction step (see section 8.6.1) and to evaluate the similarity by calculating the BER on the chroma features extracted in this step. Making a connection between the chroma features and the original audio data is easier, as the chroma features are a frequency representation of the original audio data over its whole duration. Choosing suitable audio files to create a robust test scenario would be easier this way. Last but not least, a general drawback of real time audio tests is the duration of such tests as they take several seconds each to run. They should therefore be used sparingly and only in situations where no other equivalent testing options exist.

Considerations There are a few things to consider to make audio fingerprinting test cases as robust as possible. First of all, audio fingerprinting tests should only be applied to test cases that have exactly one successful scenario and a very small number of clearly defined failure scenarios. Writing an audio fingerprinting test with many or even an unknown number of failure scenarios makes it very hard or even impossible to find an appropriate similarity threshold. The test would have to be designed in a way such that the similarity of every single failure scenario is lower than the threshold to prevent false positive test outcomes. Having a look at the test scenario outlined at the very beginning of section 9.3, there would be three realistic outcome scenarios:

1. The test is successful, as outlined in the example.
2. The test fails because the audio player keeps playing the audio file in step 4 as if the second *Start sound* brick was never executed in step 3 (which is what happened in the original version of the audio engine).
3. The test fails because the second *Start sound* brick executed in step 3 does not stop and restart the existing player, but instantiates and starts a new one instead. This leads to the PCSound being played simultaneously by two players.

Another important strategy to increase the robustness of test cases is to introduce potential points of failure as early as possible when executing a PCProject for a fingerprinting test. This guarantees that the difference in

the recorded audio data is as big as possible between the successful and failed scenarios. Having a look at the same example as mentioned above, this would mean, that the second *Start sound* brick in step 3 should be executed very early during test execution. If this would not be the case and the second *Start sound* brick would be executed 6 seconds into a test with a total duration of 7 seconds, there would be only 1 second of audio data where the difference between a successful test and a failed test could be audible. The SimHashes of the failed and successful outcomes would therefore be very similar as the recorded audio material of those outcomes is also mainly the same (apart from the last second).
However, introducing the potential point of failure as early as possible during test execution is not enough. Even if this rule is followed, the audio data of the successful and failed outcomes might still be very similar if the audio files used in a test case are not selected carefully enough. The audio files have to be selected such that the recorded audio data of successful and failed test outcomes differ as much as possible. Imagine if the audio file used in the above example would be a simple sine frequency playing for the whole duration of the audio file. It would be very hard to detect whether the audio player started playing from the beginning again or just kept on playing, as both scenarios would produce almost the same audio data (a continuously playing sine wave). A general guidance as to how audio files should be selected is not possible as this depends on the specific test scenario. The fact that it is rather difficult to understand how Chromaprint processes audio data in its post processing and SimHash steps, adds an additional level of difficulty to the proper selection of audio files. Trial and error, which means comparing the SimHashes for successful and failed outcomes of a test case for different sets of audio files, is therefore a valid approach.

10 Conclusion and future work

This thesis has studied the creation and automated testing of an audio engine on iOS and in particular for Pocket Code, a mobile programming application for children and teenagers. In this context, the different frameworks for audio playback and processing available on iOS were introduced which reach from high-level frameworks with limited functionality, to complex low-level frameworks with a vast range of tools. To choose the appropriate audio framework for the iOS version of Pocket Code, the specifications of Pocket Code's existing and future audio features have been gathered. This was done by analysing the behaviour of Scratch, a popular visual programming language whose functionality serves as a template for many of Pocket Code's own features. A comparison between the required specifications and Pocket Code's existing audio functionalities have shown many discrepancies between the intended and actual behaviour and uncovered many bugs. Additionally, this research has shown, that Pocket Code's original audio architecture was not capable to meet the needs of new audio features, leaving the audio engine in a non-expandable and bug-ridden state. These findings have shown the need for a complete redesign of the audio engine.

In a second step, the above findings were used to create a new and improved audio engine architecture. To do so, a range of audio frameworks were assessed for their suitability in the redesign process. AudioKit was ultimately chosen as the most suitable framework for the redesign. Its modular, graph-based approach provides maximum flexibility and expandability as well as real-time graph configuration while still offering an easy to use interface. With the exception of Pocket Code's TTS functionality, AudioKit allowed for the implementation of all existing and new audio features in accordance with the required specifications.

The analysis of the original audio engine not only uncovered many bugs and weaknesses, but also an insufficient test coverage. This thesis therefore

also contained research and implementation of different automated testing strategies specifically tailored to testing audio functionality. The automated test setup for the redesigned audio engine was then developed based on these findings. Developing such a test setup requires an understanding of the fundamental principles of automated testing, which was provided by highlighting the different testing levels, isolation strategies, and core principles for writing good automated tests. The thesis then further discussed how automated tests are implemented within Xcode's own testing framework and introduced a number of third party tools for automated testing on iOS. Automated testing of audio functionalities and audio data turned out to be a difficult task due to factors like time delays, time variant processing elements, floating point arithmetic and noise. From the little literature available on the topic, three audio testing strategies were identified: bit-exact audio testing, parametric audio testing and audio fingerprinting. Two of those strategies, bit-exact testing and audio fingerprinting, were finally used to test Pocket Code's redesigned audio engine. This was achieved by using AudioKit's offline manual rendering mode for bit-exact tests, and AudioKit's `AKNodeRecorder` in conjunction with the open source audio fingerprinting library Chromaprint for audio fingerprinting tests. Additionally, the audio engine was also tested with unit tests that test the graph structure, the transmission of interface calls and verify the audio engine's behaviour by using indicator data.
All of these test strategies turned out to be vital for testing the redesigned audio engine, as each strategy was used to test a specific set of specifications. Normal unit tests were used to test the internal structure and processes of the audio engine but are unable to verify the direct audio output of the engine. Bit-exact tests were therefore introduced to test the real audio output of the audio engine and proved to be a reliable and fast testing strategy. However, this strategy is restricted to testing static configurations of the audio graph only and does not work with any other source nodes apart from audio player nodes. Finally, audio fingerprinting was used to test the real audio output of specifications that deal with dynamically changing configurations of the audio graph. While the unit and bit-exact tests have proven to be very reliable in their specific test areas, audio fingerprinting tests yielded mixed results. Although very easy to implement and theoretically suitable for a broad range of test scenarios, audio fingerprinting tests led to inconsistent test results, especially when executed across different

hardware. The audio fingerprinting tests were meant to work with arbitrary audio input data, but it became apparent, that not all audio data is equally suited to produce stable and meaningful fingerprints. The high complexity of Chromaprint's fingerprinting algorithm made it hard to foresee the impact of the input audio data on the resulting fingerprints and therefore made it hard to find the best suited audio input data for specific test scenarios.

To remedy the discovered weaknesses of the audio fingerprinting implementation in Pocket Code, certain measures could be evaluated. To lower the complexity and increase the transparency of the fingerprinting algorithm, the approach discussed in section 9.3.4 could be tested which would stop the fingerprinting process after the chroma feature extraction and evaluate the similarity by calculating the BER rather than using the SimHash approach. This might produce more stable and predictable test outcomes, but would also lead to bigger fingerprints that have to be compared, as the sub-fingerprint timeline is no longer reduced to a 32-bit sequence.
Another interesting measure would be to implement pHash into Pocket Code's audio testing setup and comparing it with the current Chromaprint setup. pHash operates with a significantly different algorithm, especially in the feature extraction phase where the energy of the FFT samples is calculated in 33 different bands compared to the 12 chroma features in Chromaprint. Also, pHash uses a significantly larger overlap in the framing and overlap phase which should lead to higher resistance against timing differences. Finally, pHash's similarity measure is a confidence score based on the BER of all calculated sub-fingerprints. This confidence score is calculated for different offsets of the compared fingerprints, taking into account and eliminating possible time shifts between the two.
The small test framework which was created for audio fingerprinting tests also provides opportunities to create a vast number of other parametric audio tests. The audio data recorded by AudioKit's audio tapping architecture does not necessarily have to be processed by a fingerprinting algorithm. Instead, it can be analysed with arbitrary methods that extract other desired parameters from the recorded audio files. By only extracting the pitch of the recorded data at certain points of the audio file for example, the complexity of the analysis can be severely reduced. However, such an approach needs careful selection of the used input audio files, as the audio data needs to contain the distinctive information which the analysis algorithm extracts. A

parametric test that extracts the pitch would therefore best be conducted with sinusoidal input signals rather than complex sounds or noise.

Apart from further improving Pocket Code's audio testing setup, there is also additional functionality that could be integrated into the application. Scratch contains a number of audio features that are not yet available in Pocket Code. On the one hand, it contains additional speech bricks that let users change the language and the voice of spoken text. On the other hand, it also provides an audio editor that lets Scratch users edit the waveforms of audio files by cutting them or applying certain audio effects. The edited audio files can be saved as Scratch sounds (the equivalent of PCSounds) which can then be used in Scratch projects (the equivalent of PCProjects). AudioKit provides all necessary tools to implement a similar audio editor in Pocket Code as it contains offline manual rendering capabilities, all necessary audio effects, and classes to visualize waveforms. The additional speech bricks could simply be implemented by doing some further configuration on the speech utterance objects which are handed to the speech synthesizer.

To further improve Pocket Code's audio engine, it is recommended to regularly keep an eye out for updates of AVAudioEngine and AudioKit. Future updates of these libraries could contain valuable new features and improvements which could be relevant for Pocket Code. One feature especially worth looking out for is a speech synthesizer node that can be integrated into an audio graph structure.

Bibliography

[AA12] Chris Adamson and Kevin Avila. *Learning Core Audio*. 2012. ISBN: 978-0-321-63684-3.

[Ach14] Sujoy Acharya. *Mockito Essentials*. 2014. ISBN: 978-1-78398-360-5.

[Appa] Apple Inc. *AVAudioEngine*. URL: https://developer.apple.com/documentation/avfoundation/avaudioengine (visited on 02/17/2019).

[Appb] Apple Inc. *AVAudioMixerNode*. URL: https://developer.apple.com/documentation/avfoundation/avaudiomixernode (visited on 03/08/2020).

[Appc] Apple Inc. *AVAudioNode*. URL: https://developer.apple.com/documentation/avfoundation/avaudionode (visited on 03/08/2020).

[Appd] Apple Inc. *AVAudioPlayer*. URL: https://developer.apple.com/documentation/avfoundation/avaudioplayer (visited on 02/11/2019).

[Appe] Apple Inc. *AVAudioPlayerNode*. URL: https://developer.apple.com/documentation/avfoundation/avaudioplayernode (visited on 02/20/2018).

[Appf] Apple Inc. *AVAudioRecorder*.
URL: https://developer.apple.com/documentation/avfoundation/avaudiorecorder
(visited on 10/07/2019).

[Appg] Apple Inc. *AVAudioUnitMIDIInstrument*.
URL: https://developer.apple.com/documentation/avfoundation/avaudiounitmidiinstrument.

[Apph] Apple Inc. *AVSpeechSynthesizer*.
URL: https://developer.apple.com/documentation/avfoundation/avspeechsynthesizer
(visited on 02/07/2019).

[Appi] Apple Inc. *Defining Test Cases and Test Methods*.
URL: https://developer.apple.com/documentation/xctest/defining_test_cases_and_test_methods
(visited on 07/09/2019).

[Appj] Apple Inc. *NSCache*.
URL: https://developer.apple.com/documentation/foundation/nscache
(visited on 09/11/2018).

[Appk] Apple Inc. *Offline manual rendering documentation*.
URL: https://developer.apple.com/documentation/avfoundation/avaudioengine/2881253-renderoffline
(visited on 01/04/2020).

[Appl] Apple Inc. *Performing Offline Audio Processing*.
URL: https://developer.apple.com/documentation/avfoundation/audio_track_engineering/performing_offline_audio_processing
(visited on 03/08/2020).

[Appm] Apple Inc. *Speech Synthesis Manager*.
URL: https://developer.apple.com/documentation/applicationservices/speech_synthesis_manager
(visited on 04/27/2020).

[Appn] Apple Inc. *Testing Asynchronous Operations with Expectations*. URL: https://developer.apple.com/documentation/xctest/asynchronous_tests_and_expectations/testing_asynchronous_operations_with_expectations (visited on 07/27/2019).

[Appo] Apple Inc. *Threading Programming Guide*. URL: https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/Multithreading/RunLoopManagement/RunLoopManagement.html#//apple_ref/doc/uid/10000057i-CH16-SW1 (visited on 12/26/2019).

[Appp] Apple Inc. *What Is Core Audio?* URL: https://developer.apple.com/library/archive/documentation/MusicAudio/Conceptual/CoreAudioOverview/WhatisCoreAudio/WhatisCoreAudio.html (visited on 10/07/2019).

[Appq] Apple Inc. *Writing Test Classes and Methods*. URL: https://developer.apple.com/library/archive/documentation/DeveloperTools/Conceptual/testing_with_xcode/chapters/04-writing_tests.html (visited on 07/27/2019).

[Appr] Apple Inc. *XCTest*. URL: https://developer.apple.com/documentation/xctest (visited on 07/21/2019).

[Apps] Apple Inc. *XCTestCase*. URL: https://developer.apple.com/documentation/xctest/xctestcase (visited on 07/27/2019).

[App14a] Apple Inc. *AVAudioEngine in Practice*. 2014. URL: https://devstreaming-cdn.apple.com/videos/wwdc/2014/502xxvo7vov799k/502/502_avaudioengin

e_in_practice.pdf
(visited on 10/08/2019).

[App14b] Apple Inc. *AVAudioEngine in Practice - Transcript*. 2014. URL: https://asciiwwdc.com/2014/sessions/502 (visited on 10/08/2019).

[App17] Apple Inc. *What's New in Audio*. 2017.

(Visited on 12/30/2019).

[Aud] AudioKit Team. *AKSampler Documentation*. URL: https://github.com/AudioKit/AudioKit/blob/master/docs/AKSampler.md (visited on 10/22/2019).

[BA05] Kent Beck and Cynthia Andres. *Extreme Programming Explained*. 2nd. 2005. ISBN: 0-321-27865-8.

[Bat] Paul Batchelor. *Soundpipe GitHub*. URL: https://github.com/PaulBatchelor/Soundpipe (visited on 03/08/2020).

[BC13] Richard Blewett and Andrew Clymer. *Pro Asynchronous Programming with .NET*. 2013. ISBN: 978-1-4302-5920-6.

[Bec99] Kent Beck. *Kent Beck's Guide to Better Smalltalk*. 1999. ISBN: 0-521-64437-2.

[BKe+13] H. B.Kekre et al. "A Review of Audio Fingerprinting and Comparison of Algorithms." In: *International Journal of Computer Applications* (2013). DOI: 10.5120/12022-8054.

[Bol+15] Sylvie Boldo et al. "Verified Compilation of Floating-Point Computations." In: *Journal of Automated Reasoning* 54.2 (2015), pp. 135–163. ISSN: 15730670. DOI: 10.1007/s10817-014-9317-x. URL: http://link.springer.com/10.1007/s10817-014-9317-x.

[Bri] Brightify. *Cuckoo GitHub*.
URL: https://github.com/Brightify/Cuckoo
(visited on 10/02/2019).

[Bry] David Bryant. *WavPack Website*.
URL: http://www.wavpack.com
(visited on 10/22/2019).

[Bur03] Ilene Burnstein. *Practical Software Testing*. 2003. ISBN: 0-387-95131-8.

[Cab86] Richard Cabot. "Automated Measurements of Loudspeaker Small Signal Parameters." In: *AES Journal* 2402 (1986).

[Cab87] Richard Cabot. "Automated Measurement of Compressors and Expanders." In: *AES Journal* 2513 (1987).

[Can+02a] Pedro Cano et al. "A review of algorithms for audio fingerprinting." In: *Proceedings of 2002 IEEE Workshop on Multimedia Signal Processing, MMSP 2002*. 2002. ISBN: 0780377133. DOI: 10.1109/MMSP.2002.1203274.

[Can+02b] Pedro Cano et al. "Robust Sound Modelling for Song Identification in Broadcast Audio." In: *Audio Engineering Society Convention 112* (2002).

[Cha02] Moses S. Charikar. "Similarity Estimation Techniques from Rounding Algorithms." In: *Proceedings of the thirty-fourth annual ACM symposium on Theory of computing - STOC '02* (2002), p. 380. ISSN: 07349025. DOI: 10.1145/509961.509965.
URL: http://portal.acm.org/citation.cfm?doid=509907.509965.

[Cre17] David Creasey. *Audio Processes: Musical Analysis, Modification, Synthesis, and Control*. 2017. ISBN: 978-1-138-10013-8.

[CS] Perry R. Cook and Gary P. Scavone. *Synthesis ToolKit Website*. URL: https://ccrma.stanford.edu/software/stk/ (visited on 10/09/2019).

[Cyc] Cycling '74. *Max Website*. URL: https://cycling74.com/products/max/ (visited on 01/21/2019).

[Dia16] David Dias. *Chromaprint integration for Swift GitHub*. 2016. URL: https://github.com/NeoTeo/fingerprinter-chromaprint (visited on 12/13/2019).

[Dör] Erik Dörnenburg. *OCMock Website*. URL: http://ocmock.org (visited on 10/02/2019).

[Dre17] Daniel Drescher. *Blockchain Basics - A Non-Technical Introduction in 25 Steps*. 2017. ISBN: 978-1-4842-2603-2.

[Fea04] Michael C. Feathers. *Working Effectively with Legacy Code*. 1st. 2004. ISBN: 0-13-117705-2.

[FF05] Ira R. Forman and Nate Forman. *Java Reflection in Action*. 2005. ISBN: 1-932394-18-4.

[Fie10] Glenn Fiedler. *Floating Point Determinism*. 2010. URL: https://gafferongames.com/post/floating_point_determinism/ (visited on 01/01/2020).

[GS17] Shekhar Gulati and Rahul Sharma. *Java Unit Testing With JUnit 5*. 2017. ISBN: 978-1-4842-3014-5.

[Hay04] Linda G. Hayes. *The Automated Testing Handbook*. 2nd. 2004. ISBN: 978-0-9707-4650-4.

[Het88] Bill Hetzel. *The Complete Guide To Software Testing*. 2nd. 1988. ISBN: 0-471-56567-9.

[HK02] Jaap Haitsma and T Kalker. "A Highly Robust Audio Fingerprinting System." In: *Proceedings of the 3rd International Society for Music Information Retrieval Conference (ISMIR'02)* (2002), pp. 107–115. ISSN: 0929-8215. DOI: 10.1.1.103.2175. URL: http://ismir2002.ismir.net/proceedings/02-FP04-2.pdf.

[Int] International Catrobat Association. *Catrobat*. URL: https://www.catrobat.org/ (visited on 02/08/2019).

[IT] IT Wissen. *Bit error rate*. URL: https://www.itwissen.info/BER-bit-error-rate-BFR-Bitfehlerrate.html (visited on 02/04/2019).

[JAH01] Ron Jeffries, Ann Anderson, and Chet Hendrickson. *Extreme Programming Installed*. 2001. ISBN: 201-70842-6.

[JM14] Reiss Joshua and Andrew McPherson. *Audio Effects - Theory, Implementation and Application*. 2014. ISBN: 978-1-4665-6028-4.

[Kac13] Tomek Kaczanowski. *Practical Unit Testing with JUnit and Mockito*. 1st. 2013. ISBN: 978-83-934893-7-4.

[Kel] Mat Kelcey. *The SimHash Algorithm*. URL: https://matpalm.com/resemblance/simhash/ (visited on 02/04/2019).

[Kha+14] Shweta Kharat et al. "Audio Fingerprinting and Review of its Algorithms." In: *International Journal of Computer Technology and Applications* 5.April (2014), pp. 662–667. arXiv: ISSN:2229-6093.

[KHS05] Yan Ke, Derek Hoiem, and Rahul Sukthankar. "Computer Vision for Music Identification." In: *Proceedings of Computer Vision and Pattern Recognition (CVPR'05)* (2005), pp. 597–604.

[Koi16] Roxane Koitz. *Formula Composition and Manipulation in Educational Programming Languages for Children and Teenagers*. Master's thesis, 2016.

[Krz] Marcin Krzyzanowski. *CryptoSwift GitHub*. URL: https://github.com/krzyzanowskim/CryptoSwift (visited on 01/04/2020).

[KS] Evan Klinger and David Starkweather. *pHash Website*. URL: https://phash.org/ (visited on 01/29/2019).

[Lala] Lukáš Lalinský. *AcousticID*. URL: https://acoustid.org/ (visited on 01/29/2019).

[Lalb] Lukáš Lalinský. *Chromaprint GitHub*. URL: https://github.com/acoustid/chromaprint (visited on 01/29/2019).

[Lal11] Lukáš Lalinský. *How does Chromaprint work?* 2011. URL: https://oxygene.sk/2011/01/how-does-chromaprint-work/ (visited on 01/29/2019).

[Lin02] Johannes Link. *Unit Testing in Java*. 2002. ISBN: 1-55860-868-0.

[Low17] Doug Lowe. *Java All-in-One For Dummies*. 2017. ISBN: 978-1-119-24779-1.

[Mal+09] Davide Maltoni et al. *Handbook of Fingerprint Recognition*. 2009. ISBN: 978-1-84882-253-5.

[Mar08] Robert C. Martin. *Clean Code*. 2008. ISBN: 978-0-13-235088-4.

[Masa] Massachusetts Institute of Technology. *Scratch software instrument samples*.
URL: https://github.com/LLK/scratch-vm/tree/develop/src/extensions/scratch3_music/assets
(visited on 10/22/2019).

[Masb] Massachusetts Institute of Technology. *Scratch Sound Effects*.
URL: https://en.scratch-wiki.info/wiki/Sound_Effect
(visited on 10/21/2019).

[Masc] Massachusetts Institute of Technology. *Scratch Website*.
URL: https://scratch.mit.edu
(visited on 02/05/2019).

[Mes07] Gerard Meszaros. *xUnit Test Patterns*. 1st. 2007. ISBN: 978-0-13-149505-0.

[MID] MIDI Association. *Summary of MIDI Messages*.
URL: https://www.midi.org/specifications/item/table-1-summary-of-midi-message
(visited on 01/22/2019).

[Mis17] Abhishek Mishra. *iOS Code Testing*. 2017. ISBN: 978-1-4842-2688-9.

[Moc] Mockito. *Mockito Website*.
URL: https://site.mockito.org/
(visited on 10/01/2019).

[MW06] Steven John Metsker and William C. Wake. *Design Patterns in Java*. 2006. ISBN: 0-321-33302-0.

[Nat02] National Institute of Standards and Technology. *Software Errors Cost U.S. Economy $59.5 Billion Annually*. 2002.
URL: http://www.abeacha.com/NIST_press_release_bugs_cost.htm
(visited on 11/05/2018).

[NK05] Patrick Niemeyer and Jonathan Knudsen. *Learning Java*. 3rd. 2005. ISBN: 978-0-596-00873-4.

[Oro15] Gergely Orosz. *Swift: The Only Modern Language without Mocking Frameworks*. 2015.
URL: https://blog.pragmaticengineer.com/swift-the-only-modern-language-with-no-mocking-framework/
(visited on 10/02/2019).

[OSB99] Alan Oppenheim, Ronald Schafer, and John Buck. *Discrete-Time Signal Processing*. 2nd. 1999. ISBN: 0-13-754920-2.

[Osh14] Roy Osherove. *The Art of Unit Testing*. 2nd. 2014. ISBN: 978-1-6172-9089-3.

[Oxf] Oxford Dictionaries. *Oxford Dictionaries*.
URL: https://en.oxforddictionaries.com/definition/unit
(visited on 11/13/2018).

[PLP12] Nils Peters, Trond Lossius, and Timothy Place. “An automated testing suite for computer music environments.” In: *Proceedings of the 9th Sound and Music Computing Conference, SMC 2012*. 2012. ISBN: 978-3-8325-3180-5.

[PP10] Christoph Paar and Jan Pelzl. *Understanding Cryptography*. 2010. ISBN: 978-3-642-04100-6.

[Proa] Aurelius Prochazka. *AudioKit GitHub*.
URL: https://github.com/audiokit/AudioKit
(visited on 01/21/2019).

[Prob] Aurelius Prochazka. *AudioKit Reference*.
URL: https://audiokit.io/docs/
(visited on 01/01/2020).

[Puc] Miller Puckette. *Pure Data Website*.
URL: https://puredata.info/
(visited on 01/21/2019).

[Quia] Quick Team. *Nimble GitHub*.
URL: https://github.com/Quick/Nimble
(visited on 10/03/2019).

[Quib] Quick Team. *Quick GitHub*.
URL: https://github.com/Quick/Quick
(visited on 10/03/2019).

[Rob68] D. A. Roberts. “High Speed Automated Test Set.” In: *AES Journal* 576 (1968).

[Sen05] Sen:te. *Test first, develop later!* 2005.
URL: http://www.sente.ch/software/ocunit/
(visited on 07/06/2019).

[SFZ] SFZ. *SFZ Website*.
URL: https://sfzformat.com
(visited on 10/22/2019).

[Sha] Shazam. *Shazam Website*.
URL: https://www.shazam.com/
(visited on 01/25/2019).

[SSR08] Klaus Solbach, Ali Sen, and Michael Reiner. *Conception and implementation of a test system for automated tests of audio outputs with the method of acoustic fingerprinting*. Tech. rep. 2008.
URL: https://www.uni-due.de/imperia/md/content/hft/theses/vortrag_sen_ali.pdf.

[Tak05] M E Takanen. *Automated system level testing of a software audio platform*. Master's thesis, 2005.

[The] The Echo Nest. *Echoprint GitHub*.
URL: https://github.com/echonest/echoprint-server
(visited on 01/29/2019).

[Tho18] Mattt Thompson. *AVSpeechSynthesizer*. 2018.
URL: https://nshipster.com/avspeechsynthesizer/
(visited on 02/07/2019).

[TKB18] Frank Tsui, Orlando Karam, and Barbara Bernal. *Essentials of Software Engineering*. 4th. 2018. ISBN: 978-1-2841-0600-8.

[WH12] Matt Wynne and Aslak Hellesoy. *The Cucumber Book - Behaviour-Driven Development for Testers and Developers*. 2012. ISBN: 978-1-934356-80-7.

[Zöl02] Udo Zölzer. *Digital Audio Effects*. 2002. ISBN: 0-471-49078-4.