P8 Project

Tempo Player - Tunes for running

P8 Project

Group SW802F15

Software

Department of Computer Science
Aalborg University

Spring 2015

AALBORG UNIVERSITY
STUDENT REPORT

Department of Computer Science
Selma Lagerløfs Vej 300
9220 Aalborg Ø
Phone (+45) 9940 9940
Fax (+45) 9940 9798
http://www.cs.aau.dk

Title:
Tempo Player - Tunes for Running

Subject:
Mobile Systems

Project period:
2015-02-02 – 2015-05-27

Project group:
SW802F15

Participants:
Alexander Drægert
Christoffer Nduru
Dan Petersen
Kristian Thomsen

Supervisor:
Ivan Aaen

Printings: 6

Pages: 53

Appendices: 24

Total pages: 90

Abstract:

The purpose of this project was to make a runner's running experience less tedious and more inspiring, by playing music which fits their running pace. The project is based on the research of Edworthy and Waring, which concluded that music tempo affects a runner's pace and enjoyment. The project will produce an Android application capable of playing music, detecting step-per-minute, and navigation without looking at the screen. The application will be developed through Extreme Programming, as we try to adapt it to our project.
The project resulted in an Android application playing music and detecting steps-per-minute. We found Extreme Programming to be a productive methodology. We did not fully adapt the practices small releases, continuous integration, and collective ownership. We fully adapted and experimented with pair programming and planning game.

Source code:
https://github.com/SW802F15/SourceCode/tree/79117cbbfdb588ac03c1d13471e8398698fcad64

Preface

This report is written by four 8th semester software engineering students at Aalborg University.

The source code is, per Aalborg University policies, available by searching for this project on `http://projekter.aau.dk/projekter/`.

Furthermore, it is available as a zip archive through GitHub:

- `https://github.com/SW802F15/SourceCode/archive/79117cbbfdb588ac03c1d13471e8398698fcad64.zip`
- `https://goo.gl/FYk886` (shortening of above URL)

We would like to thank our supervisor Ivan Aaen for providing guidance and feedback.

Aalborg, May 27, 2015

Alexander Drægert

Christoffer Nduru

Dan Petersen

Kristian M. Thomsen

Contents

1 Introduction

Running is a popular form of exercise, however, it can be a tedious and uninspiring endeavour. To improve the experience, Edworthy and Waring [7] found that *"... participants enjoyed what they were doing [running] more when they were listening to music of any sort when compared to when they were not."*

It was further concluded by Edworthy and Waring [7] that the volume and tempo of the music influenced the running experience. They concluded that the running pace for novice runners, while listening to relatively low-tempo music, was slower than when not listening to music. Additionally listening to high-tempo music resulted in a faster running pace, compared to when not listening to music.

This conclusion is in disagreement with the conclusion of Yamamoto et al. [19] which suggests, that *"... music had no impact on mean power output."*. Yamamoto et al. [19] measure the running pace by mean power output, nevertheless, they did not see any impact on the running pace by listening to music.

As a result, we can not definitively conclude whether music of different tempo will affect the running experience differently. However by adhering to Edworthy and Waring [7]'s conclusion, we can only improve the running experience, since Yamamoto et al. [19] concludes there can be no negative impact, by playing music.

Today many runners use their smartphone as a music player, which can either be placed in their hand, pocket, or on their arm. The sensors in a smartphone enable monitoring the pace and speed of the runner. Based on this knowledge the first problem can be stated:

How can we provide music with an appropriate tempo, compared to the current pace, to the runner through the use of a smartphone?

Operating a smartphone while running is difficult, especially if it is placed in the pocket or on the arm of the runner. In order for the runner to operate the smartphone; play, pause, stop, or change song, the runner would have to stop running, or focus more than normally. This can disrupt the runner's form, and lead to injuries and accidents. Based on this knowledge the second problem can be stated:

How can a smartphone application be operated without disrupting the runner's form and/or concentration?

According to Gartner Inc. [10] *"A full 66 percent of large scale projects fail ..."*, and although this is not a large scale project, some of the same pitfalls exist. One way to avoid some of these pitfalls, is *"... using a structured systems development methodology ..."*, since it according to Dorsey [5] *"... is one of the critical success factors in a systems development project."*.

We will in this project focus on the development methodology `Extreme Programming` (XP). XP is used because its an interesting methodology and it is development oriented. Furthermore, XP's requirements of self-organising teams, iteration length, and team size fit well with this project. Based on this knowledge the third problem can be stated:

How do we adapt the structured systems development methodology, Extreme Programming, to our project?

Methodology 2

Extreme Programming (XP) is a software development methodology created by Kent Beck. This chapter is based on his book *Extreme Programming Explained (1999)* [3], William Wake's book *Extreme Programming Explored (2000)* [17], and Don Wells' interactive introduction *Extreme Programming: A Gentle Introduction (2013)* [18].

2.1 Extreme Programming Explained

Extreme Programming is supposed to be a lightweight, efficient, and fun approach to developing software. One of its core points, and one of the reasons we chose XP, is the fact it contains development oriented practises.

XP is based on the four values *Communication, Courage, Feedback, and Simplicity*, which are implemented through the use of the 12 practises:

40-hour Week	Coding Standards	Collective Ownership
Continuous Integration	Metaphor	On-site Customer
Pair Programming	Planning Game	Refactoring
Simple Design	Small Releases	Testing

These practises support each other (e.g. a coding standard supports pair programming), in such a way it creates a synergistic effect. The complete map of which practises support each other, can be seen in Figure 2.1.

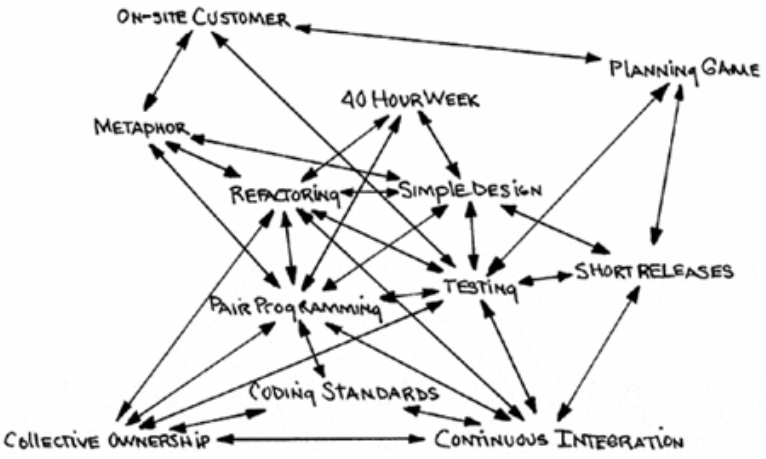

Figure 2.1: Map of the practises and how they support each other. From *Extreme Programming Explained (1999)* [3, p. 70].

40-hour Week is the suggested length of a work week in an XP project. The relatively short week is practised because one needs to be on the top of their game when working. Being well rested and energised should reduce the number of mistakes, as well as increase patience when pair programming.

Coding Standards reduce the time it takes to understand code written by others. A coding standard also ensures that all developers write code the same way. This facilitates refactoring, pair programming, and encourages collective ownership.

Collective Ownership allows everyone to refactor and review the code. When no one person owns part of the code, collaboration is enforced while tedious change requests are avoided.

Continuous Integration is used to build the program several times a day. This keeps everybody up to date with the latest code changes, avoiding development on fragmented versions.

Metaphor is used as a shared vision for the project. The metaphor is used to standardise the names of variables, methods, and classes. This standardisation should help team members to intuitively understand the purpose of the variable, method, or class.

On-site Customer is an integral role in an XP project. The on-site customer is responsible for making decisions about priorities, requirements, and answering questions.

Pair Programming is the way to code in an XP project. It is when two developers work together on the same computer. While one writes the code, the other analyses and comes with suggestions for improvement. Development should be seen as a dialogue between the two developers.

Planning Game is the event where the customer writes user stories, after which the developers estimate them according to cost.

Refactoring is the practise used to improve the quality of the internal code without changing the external behaviour. Refactoring should also improve readability, and thereby maintainability.

Simple Design is when every piece of code justifies its own existence. The code should not have any duplicated logic, and have the fewest possible classes and method.

Small Releases are small and frequent versions of the software, which are tested by the customer. These releases incrementally complete the system.

Testing refers to how the stories and releases are not complete, before they pass their respective tests. It is a mindset in which, unit tests are developed before the code (Test-Driven Development).

2.2 Extreme Programming versus Scrum

Scrum is one of, if not the, most popular agile methodology today. Almost all developers know what Scrum is, hence we will compare XP and Scrum. The two are similar, both being agile methodologies with self-organising teams, and similar meeting practises, however there exists some significant differences.

The most significant difference, is that Scrum does not have any development oriented practises. XP on the other hand, has plenty of development oriented practises. Practises such as pair programming, test-driven development, refactoring, automated testing, simple design, and so forth. The major difference originates, acording to Johnson [15], from Scrum being a project management framework, whereas XP is purely a software development methodology.

Another difference is the flexibility of the iteration, or sprints as they are called in Scrum. When a sprint is planned and started, the sprint backlog is not susceptible to change. The iteration backlog in XP can be changed, as long as the swapped features are not in progress.

The last significant difference is the role responsible for prioritising the work order. In Scrum the Product Owner prioritises the product backlog, but the developers choose the work order. In XP the customer prioritises the product backlog, and the developers must work following this prioritisation.

2.3 Adaptation of Extreme Programming

This project is developed with an educational purpose, to which some requirements and restrictions exist. Partly due to these requirements and restrictions, we made some adaptations to XP for it to fit inside the project parameters.

2.3.1 40-hour Week

The purpose of this practise is to ensure energetic developers every day, and not to overwork them to the point of burnout as explained by Wake [17, p. 58]. We will implement this practise by timeboxing the project time. This way we will still be able to ensure rested and energetic developers, regardless of courses and other non-project matters.

2.3.2 Code Standards

Creating a formal code standard is a time consuming task. We expect that creating a formal code standard, when we already have an informal code standard, will be too time consuming compared to the benefits. The informal code standard has been created and expanded through our previous experiences of working together.

2.3.3 Collective Ownership

As explained by Wake [17, p. 54], this practise supports flexible and quick changes, however some risks still exist. The problems most likely to occur in this project are: lack of expertise, violating private space, and personal pride.

The problem with lack of expertise occurs when a problem is just passed on, without anyone having any expertise in the overall problem. By using Pair Programming the

expertise will be spread to the entire team, ensuring everyone obtains knowledge about the overall problems.

The problem of violating each others' private space, refers to when developers need to access and work in the same document. This can cause developers to stay at their current version and only code there, in order to avoid conflicting documents. However, by using Continuous Integration it will be impossible to be stuck on an old version. By using the Testing practise, it is ensures changes will not compromise the existing code.

The problem with personal pride is not directly addressed by XP. This problem can occur if a developer becomes proud of their work, and does not want to see it "ruined" by others. This is a problem we will have to watch out for and to remind ourselves of XP's saying: "Be brave".

2.3.4 Continuous Integration

According to Wake [17, p. 57] this practise should be automated through the setup of a dedicated build server. Making this an automated task, should ensure the developers working on the current version. The integration itself should not be automated, but the build and testing should, i.e., when a story is completed, the changes are integrated into the current build. The build server then automatically builds and tests the build. If the tests are not at a 100%, the developers must fix their code or discard it, to ensure the build always passes 100% of the tests.

Even though finding and setting up a build server would be the correct way of doing things, we will manually integrate and build the code from a dedicated branch. We will do this because, we believe we can achieve the same benefits with the manual approach, without using time to set up the build server. This is only viable because of the small size of the project, which results in short build times and a limited risk of fragmented versions.

2.3.5 Metaphor

The metaphor is an effective way to get a shared vision for the project. A meaningful metaphor helps create and name the correct objects and actions as described by Wake [17, p. 87].

We see this project as a music player that plays music with a tempo matching the user's pace. This metaphor might change throughout this project.

2.3.6 On-site Customer

Since this is a semester project, we will not have an actual customer. In place of the on-site customer, we will act as a surrogate customer. This means that we are responsible for the priority, requirement decisions, and creation of acceptance tests, as well as other tasks normally done by the customer.

2.3.7 Pair Programming

To fully utilise pair programming, we will borrow two sets of monitors, keyboards, and mice. We will connect these to our laptops to create two workstations ready for pair programming.

There exists several recommendation of how to pair program most efficiently. We will try to experiment with different aspects of the practise to experience the advantages, and

disadvantages of the different recommendation, as we suspect this is the fastest and most efficient way to learn.

2.3.8 Planning Game

Planning is important. It lets you know how fast you work, what to do next, provides an effective means of communication with the customer, and allows you to quickly respond to changes if necessary.

In XP planning happens in two stages. There is the release planning and the iteration planning. The release planning event, as explained by Beck and Fowler [4, p. 40], is where the customer writes all the user stories, he wants the program to cover. The developers then estimate the time it will take to implement the different stories. Then the customer prioritises the stories by what should be done first. Finally the stories to be done in the release are selected based on project velocity, and priority of each story.

The iteration planning event, as explained by Wells [18, Iteration Planning], is where the customer selects which of the stories, from the release plan, he wants implemented in the upcoming iteration. Usually the highest prioritised stories are chosen, but often these are superseded by the stories not completely implemented last iteration. The customer is asked to choose a number of stories, based on the project velocity, so the developers are not overworked.

A story can be estimated individually, or collaboratively. Both methods have strengths and weaknesses. We decided to collaboratively estimate stories, as that approach allows us to use our combined experience to find the right estimation.

2.3.9 Refactoring

As mentioned by Wells [18, Refactor Mercilessly], XP prescribes refactoring throughout a project's life cycle. We will refactor on the spot when a bad smell is identified, to prevent building on top of bad code. Although refactoring is important, we will evaluate each bad smell by severity and time needed to fix, before correcting it. We will then decide if it is worthwhile to correct the bad smell, compared to implementing new features. Otherwise it will be created as a task/story.

2.3.10 Simple Design

To implement a simple design, we will structure our code in modules. Each module will contain multiple classes, which all will be short and readable. To improve readability we will name methods after their function and only implement features with an immediate purpose. This will prevent having code that is never used.

2.3.11 Small Releases

The purpose of this practise is to reduce the risks if changes in the use domain occur, as explained by Wake [17, p. 61]. Although releases are frequent, they must only contain complete features.

This project contains three major features, which we will try to match to approximately three releases. With each release we will have a complete feature, which then will be user tested and the feedback will be implemented in the next release.

2.3.12 Testing

Testing covers both unit testing and acceptance testing, both are important parts of XP.

Unit tests are created to ease the refactoring process, by ensuring the external behaviour has not been compromised after a change has been made. We will, as recommended by XP, make use of Test-Driven Development (TDD).

Acceptance tests are created to ensure the product satisfies the customers needs. Acceptance tests are written by the customer, which we will do ourselves, seeing as we use ourselves as a surrogate customer.

Tempo Player - The Tour 3

Tempo player is an application for the Android platform, which is able to play and select music from an already existing library of songs, which fits the user's running pace measured in steps per minute (SPM). It uses the accelerometer for calculating the user's running pace. The application's main screen can be seen in Figure 3.1.

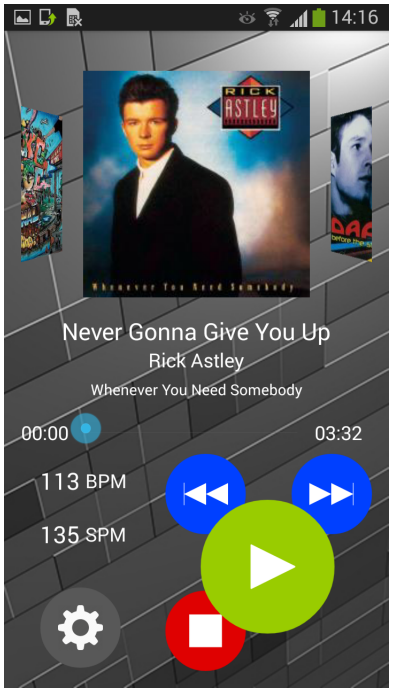

Figure 3.1: A screenshot showing Tempo Player's main screen.

The *BPM* represent the song's "Beats Per Minute". This information is retrieved from a web service and therefore the application initially requires an internet connection to function as intended. After the songs' BPMs are retrieved, they are stored in an SQL Lite database for later use. The *SPM* represent the user's current number of steps per minute.

The Android device should be mounted on the user's arm and when the next button is pressed the BPM of the next songs played will match the user's SPM if possible, otherwise nearest BPM is selected.

When the user stands still for more than 2 seconds, the SPM are set to 0, since no steps are taken. See Section 7.3.1 for details.

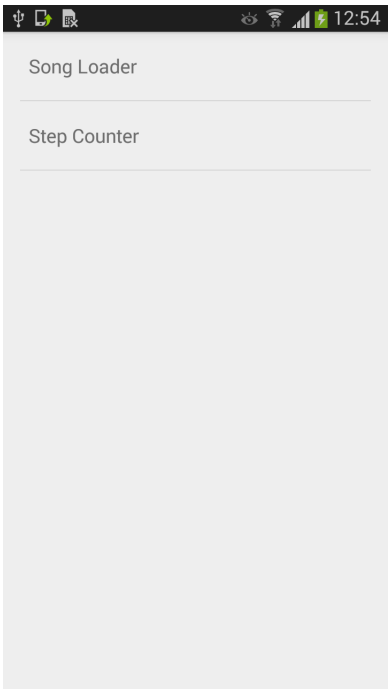

Figure 3.2: Screenshot showing the settings screen.

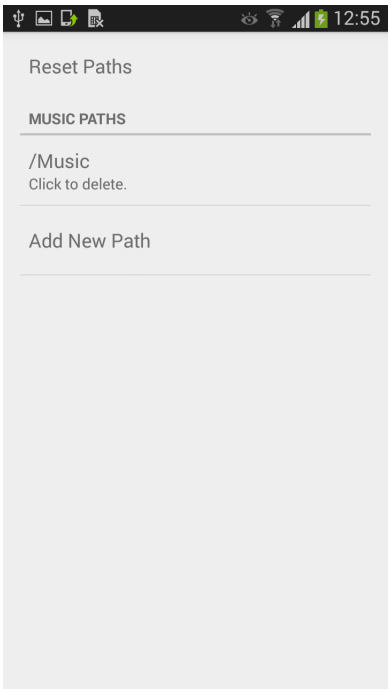

Figure 3.3: Screenshot showing the "Reset paths" screen of the settings screen.

When standing still all songs in the library will be available to the user in a shuffled order.

As shown in Figure 3.2, the user can adjust some settings of the application. Currently, the only adjustment the user can make is in which directory the application looks for songs. This settings screen is shown in Figure 3.3.

It is also possible to seek to a position in the currently playing song by dragging the seek bar to a location along the time line.

3.1 Usage Scenario

A typical usage scenario of the Tempo Player could look as such

1. The user starts music from the main screen, turns of the screen, and puts the smartphone in a armband, which is located on the user's arm.
2. The user starts walking at a brisk pace.
3. After a while, the non-graphical user interface (NGUI) is used to change to the next song, by tapping twice on the screen.

   Since the application has now calculated the SPM, the next song has a BPM value which fits the user's pace.
4. The user accelerates, and starts jogging at a slow pace.
5. The song finishes, and the application automatically plays the next song which fits the current pace.

6. The user accelerates, and is now running. The screen is tapped twice, and faster music, which fits the new pace, is played.

7. Finally, the user gradually slows down and eventually stops moving completely. A long press (500 ms) is performed on the screen, and the music stops. The exercise session is over.

Releases 4

Over the course of this project we have been working in iterations of two weeks, which is approximately one week worth of work on the project. To follow the practise *small releases* there is one release for every two iterations. In this chapter we will cover the goals and findings of each release.

In this project, we decided to work with selected parts of Essence from `Essence: Pragmatic Software Innovation` [2]. Essence is a methodology rooted in the pragmatic paradigm, as opposed to the agile or traditional paradigm. Essence is a response to a world view, where the context is constantly evolving, and there is no single best criteria to evaluate a product. Essence focuses on creating value for the customer, which is done by facilitating innovation. It relies on an iterative workflow, and adds an extra layer for evaluation of the process.

We look to Essence to evaluate our use of XP practise, this is done by review/retrospective after each iteration.

Further we describe our releases with Essence configurations. In Essence, a configuration is a look at the product after an iteration, you then move from configuration to configuration as you go though the iterations.

There are four views in Essence, each with a focus:

- `Paradigm`: Challenge from a user perspective.
- `Product`: Design of the product.
- `Project`: Planning.
- `Process`: Idea development and evaluation.

4.1 1st Release

When the project just started a formal release was not planned. We had a general idea of the core functionality of the application, and that is what we worked towards. During the first iteration we mostly researched how to build the application, but the work that was done at the end of the 2nd iteration was fundamentally a release. As we got more comfortable with XP, and learned how to better make use of short releases, around the 3rd iteration, we started planning actual releases. To make a foundation for the following releases, the first release can be described as follows:

The application should be able to handle playing music and navigating a play list, using a graphical user interface.

This release is slightly smaller than the ones to follow, and its main purpose was to get the application prepared for the next releases. Because the release is as simple as it is, it is not analysed further.

4.2 2nd Release

With the core music player application completed in the 2nd iteration we outlined the configuration for the 3rd and 4th iterations in the configuration table seen in Table 4.1, which is based on `Essence: Pragmatic Software Innovation` [2, ch. 3]. The figure should be read vertically, and is meant to provide an overview; each field is explained in more detail in this section. This approach presents the different aspects of the release, including what problem the application should solve and how, and it provides a basis for further development. The configuration table outlines the planned release, and not the finished release.

- **Paradigm Focus**: The challenge is to improve the running experience by matching the tempo of the music with the pace of the runner. The application is supposed to be used while running.
- **Paradigm Overview**: Stakeholder is the user who is using the application, i.e., the runner.
- **Paradigm Details**: For this release there are two main scenarios to fulfil. Firstly the application should automatically select appropriate songs (songs with a matching BPM) to play as the user runs at varying paces. Secondly it should be possible to customise the song pool from which songs are selected.
- **Product Focus**: The application is first and foremost a running pacer, meaning the purpose is to match music tempo with the user's pace, measured in steps per minute. It is possible to add an option for interval training so the application no longer adapt the music's tempo based on pace, but encourage the user to adapt his or her pace to predetermined intervals of low and high tempo music. Additional training programs could also be added. The application can also serve as a step counter, not playing any music but just recording steps taken and displaying it as either total steps or steps per minute. Similarly it can just play music without taking SPM into account.
- **Product Overview**: The application design should be based on a familiar music player design, and it should have the expected functionally from a music player, but new elements (such as SPM) should expand on the familiar design. The navigation buttons should be placed, so they are easy to use while moving.
- **Product Details**: This configuration consists of 3 components: the music player, a music library, and a step counter. The music player is used to play songs and serve as the primary interface for the user. The music library is one or more folders with music files on the device. The step counter makes use of the device's accelerometer to register steps taken.
- **Project Focus**: The overall vision of the application is as a running pacer which matches music to the user's running pace.

	Paradigm	**Product**	**Project**	**Process**
Focus	*Reflection* • Challenge: Can we improve the running experience? • Use context: Running while listening to music from a smartphone.	*Affordance* • Running pacer. • Option: Interval Trainer / Training programs. Step Counter (no music). Music Player (no running).	*Vision* • Vision: Running pacer by use of music. • Use step counter to match song to running pace.	*Facilitation* • Focus on immediate benefits to user.
Overview	*Stakeholders* • A runner who enjoys music.	*Design* • Running pacer with music player-like design and functionalities.	*Elements* • Grounds: Music that fits your pace provides a better running experience. • Warrant: When running it is human nature to match pace with the tempo of the music playing. • Qualifier: Songs matching current pace required at all times. • Rebuttal: Use context: Changing running pace during a song. Limitations: It is necessary to alter the music's tempo or transition to other songs.	*Evaluation* • Procedure: Iteration review with surrogate customer. • Criteria: Evaluate immediate functionality based on acceptance tests.
Details	*Scenarios* • Automatically fade into songs, which fit running pace. • Use personal collection of music files as a basis for exercise/running.	*Components* • Music player • Music library • Step counter	*Features* • Running pacer • Music player • Step counter	*Findings* • Controlling the device while running can be difficult.

Table 4.1: Configuration table for 2nd release.

- **Project Overview**:
  Grounds: As described in chapter 1, music can have an influence on the running experience. Having music that matches your running pace makes sure it does not have a negative impact on the exercise session, and having the application find appropriate music automatically makes the process of finding good running music less tedious.

Warrant: Many people prefer running while listening to music, and it will often result in people trying to match their running pace with the music playing.
Qualifier: It is important that the song playing always, or as much as possible, matches the user's pace, so their running experience is not disrupted.
Rebuttal: The user may change pace in the middle of a song, and this poses a problem. It could be solved by altering the music's tempo just enough to match the new pace, without distorting the sound of the music too much. Another solution could be to transition to another piece of music, when the new pace is confirmed stable.

- **Project Details**: The application should work as a running pacer, a music player, and a step counter.
- **Process Focus**: Through an iterative process there will be focus on the most important features as decided by the customer, providing immediate benefits to the user.
- **Process Overview**: We evaluate the application from the customer's point of view at the end of each iteration. The evaluation is based on the fulfilment of acceptance tests.
- **Process Details**: While running it can be difficult to navigate the application, as there is limited feedback when using a touch screen. Instead it should be possible to use gestures or taps to control the application, giving it a more consistent user experience.

4.2.1 Release Evaluation

At the end of the 4th iteration the application had the desired components: Music player, a music library (referred to as the song scanner module), and a step counter. Each of the components worked on their own, but had not been properly connected, resulting in only the features *music player* and *step counter* being done; the *running pacer* requires a combined effort from the other two features. Overall, the application ended in a state that required few changes to bring a lot of value to the customer.

4.3 3rd Release

The 3rd and final release, a modification of the 2nd release, is outlined in Table 4.2. Most fields are unchanged, as most requirements did not change. Those that did change are written in bold green text.

- **Paradigm Details**: Based on the findings from the 2nd release, a new scenario emerged: When the user is running with the screen off, it should still be possible to control the application.
- **Product Overview**: The product design should include a way to interact with the application without a graphical interface.
- **Project Details**: While the components are the same, a new feature is needed for the music player: A non-graphical user interface, allowing the user to navigate the application while running and without looking at the screen.

	Paradigm	**Product**	**Project**	**Process**
Focus	*Reflection* • Challenge: Can we improve the running experience? • Use context: Running while listening to music from a smartphone.	*Affordance* • Running pacer. • Option: Interval Trainer / Training programs. Step Counter (no music). Music Player (no running).	*Vision* • Vision: Running pacer by use of music. • Use step counter to match song to running pace.	*Facilitation* • Focus on immediate benefits to user.
Overview	*Stakeholders* • A runner who enjoys music.	*Design* • Running pacer with music player-like design and functionalities. • **Non-graphical UI making interaction possible without looking at the screen.**	*Elements* • Grounds: Music that fits your pace provides a better running experience. • Warrant: When running it is human nature to match pace with the tempo of the music playing. • Qualifier: Songs matching current pace required at all times. • Rebuttal: Use context: Changing running pace during a song. Limitations: It is necessary to alter the music's tempo or transition to other songs.	*Evaluation* • Procedure: Iteration review with surrogate customer. • Criteria: Evaluate immediate functionality based on acceptance tests.
Details	*Scenarios* • Automatically fade into songs, which fit running pace. • Use personal collection of music files as a basis for exercise/running. • **Control the music player with taps, while the screen is turned off.**	*Components* • Music player • Music library • Step counter	*Features* • Running pacer • Music player • Step counter • **Non-GUI based interaction**	*Findings* • **Lack of tactile feedback in non-graphical interface makes it harder to control.** • **If the user stops moving, it may be nice if the song fitting the last stable SPM keeps playing.**

Table 4.2: Configuration table for 3rd release.

- **Process Details**: Without any feedback, e.g. a vibration, it is hard to determine if the command given in the non-graphical user interface was registered, making it

harder to control properly.
Additionally, it is necessary to take an appropriate action if the user stops running. This could be recording their last stable pace, and keep playing songs matching that tempo.

4.3.1 Release Evaluation

In addition to the components of 2nd Release, the 3rd Release included a non-graphical interface to answer the findings of the 2nd release. Every component stub had been replaced, so the application were properly connected. The application is now fully functional as a *running pacer*, though new findings revealed new paths to improve the application.

5 The Music Player Module

The music player module is an essential part of the application. Its responsibility is to play music which fits the user's pace. It has a GUI through which the user can navigate a playlist. By combining the functionality of the dynamic queue, and the step counter module, the application can choose songs that match the user's pace.

5.1 Android Terminology

Here are some Android specific terms that will be used in the implementation sections. These explanations are based on the Android API Guide Glossary [13] and the Sensor Overview [12].

Activity

A screen in the application responsible for handles UI events. It is implemented as a Java class with a layout primarily defined in XML.

Service

A service runs in the background with the purpose of handling long-running, persistent actions, e.g. playing music when the application is in the background. It does not provide a user interface.

Sensors and Listeners

The device can have a number of sensors, e.g. an accelerometer, and these can be accessed by listeners. A listener implements methods like `onSensorChanged()`, which is called with a value when the sensor receives a new input. Listeners are also used to detect input events like button clicks and touches.

5.2 User Stories

The following are the most interesting user stories used to develop the Music Player module. Some of the stories depend on other modules. It is, for example, not possible to "Play music files" without the Song Scanner, which is described in Chapter 6.

TITLE: Play music files.
PRIORITY: High
STORY: As a user, I, want to listen to the music files stored on my device.

ACCEPTANCE CRITERIA:
- I am able to play a music file stored on the device.
- I am able to regulate the volume when listening to the song.
- I am only able to open music files with the application.

TITLE: Navigate songs.
PRIORITY: High
STORY: As a user, I, want to be able to skip the current song, listen to the previously played song again. Further I want to be able to stop and/or pause the playing song. It should also be possible to seek in the playing song.

ACCEPTANCE CRITERIA:
- A `Next` button must exist, which skips to the next song.
- A `Previous` button must exist, which skips to the previously played song.
- A `Stop` button must exist, which stops the music.
- A `Pause` button must exist, which pauses the music, so it can be resumed from the paused position.
- A `SeekBar` must exist, which can seek in the playing song.

TITLE: Keep playing when screen is off.
PRIORITY: High
STORY: As a user, I, would like to save power by turning off the screen while the music is playing.

ACCEPTANCE CRITERIA:
- The music keeps playing after the `Power` button is pressed.

TITLE: Adjust music tempo.
PRIORITY: High
STORY: As a user, I, want the music to match my pace even when the playing song's tempo is not exactly my pace.

ACCEPTANCE CRITERIA:
- The application must be able to match my pace with appropriate songs, when the smartphone is held in either the hand, the pocket, or on the arm.

5.3 Implementation

The Music Player module consists of a number of classes, where the most notable are `DynamicQueue`, `MusicPlayerActivity`, and `MusicPlayerService`. `DynamicQueue` handles navigation, and it makes sure the queue of songs matches the user's running pace. `MusicPlayerActivity` shows a UI on the screen and handles user inputs, including navigation and volume changes. `MusicPlayerService` handles the actual playback of songs, and because it is implemented as a service, it is possible to keep playing songs in the background, i.e., without requiring `MusicPlayerActivity` to be visible.

5.3.1 DynamicQueue

This class is named based on the fact that it will update dynamically, depending on the pace of the user. `DynamicQueue` utilises two queues to keep track of previously played and upcoming songs. Each queue has a fixed size, so only the desired amount of songs will be stored at any given times, and only songs matching a given tempo are loaded into the queue for upcoming songs. Additionally, to prevent simply replaying the same songs over and over, all the songs in the two queues must be unique.

5.3.2 MusicPlayerActivity

This is the central point of the application, responsible for managing the rest of the application, and giving the user a way of interacting with it. It starts the `MusicPlayerService`, calls the `DynamicQueue` for songs to play, attaches listeners to the buttons, making them clickable. It also checks that there are enough songs in the specified folder, sending the user to the settings menu if this is not the case.

In order to manage the responsibilities associated with the activity, general initialisation has its own class `Initializers`. `Initializers` is responsible for setting `onClickListeners` for the GUI, and preparing the first song in the queue and coverflow.

5.3.3 MusicPlayerService

The `MusicPlayerServce` handles the playing of the music files. This is done with the `MediaPlayer` class, which is part of the Google Android [11] API. The service is responsible for setting up event listeners, and acts as an interface to the `MediaPlayer`.

We only used one of the two listeners available for the `MediaPlayer`, the `onCompleteListener`. This listener is called when a file reaches its end. In this event the method attached to the listener, selects the next song in Queue, by calling the `nextSong` method.

The `onCompleteListener` is also called when an error occurs while the `MediaPlayer` is playing. In our application an illegal state error could happen if one calls `stop` while the `MediaPlayer` was not playing, triggering the `onCompleteListener`, and changing the song to the next in the queue. This was handled in the `MusicPlayerService` by ignoring `stop` or other calls, if the `MediaPlayer` was in the wrong state.

The `MediaPlayer`'s state diagram can be seen in fig. 5.1. The application ends up in an illegal state every time a method is called from a state, which does not have a corresponding transition. An example could be that `prepare` was called from the `started` state. One would have to call `stop` before `prepare` could be called again.

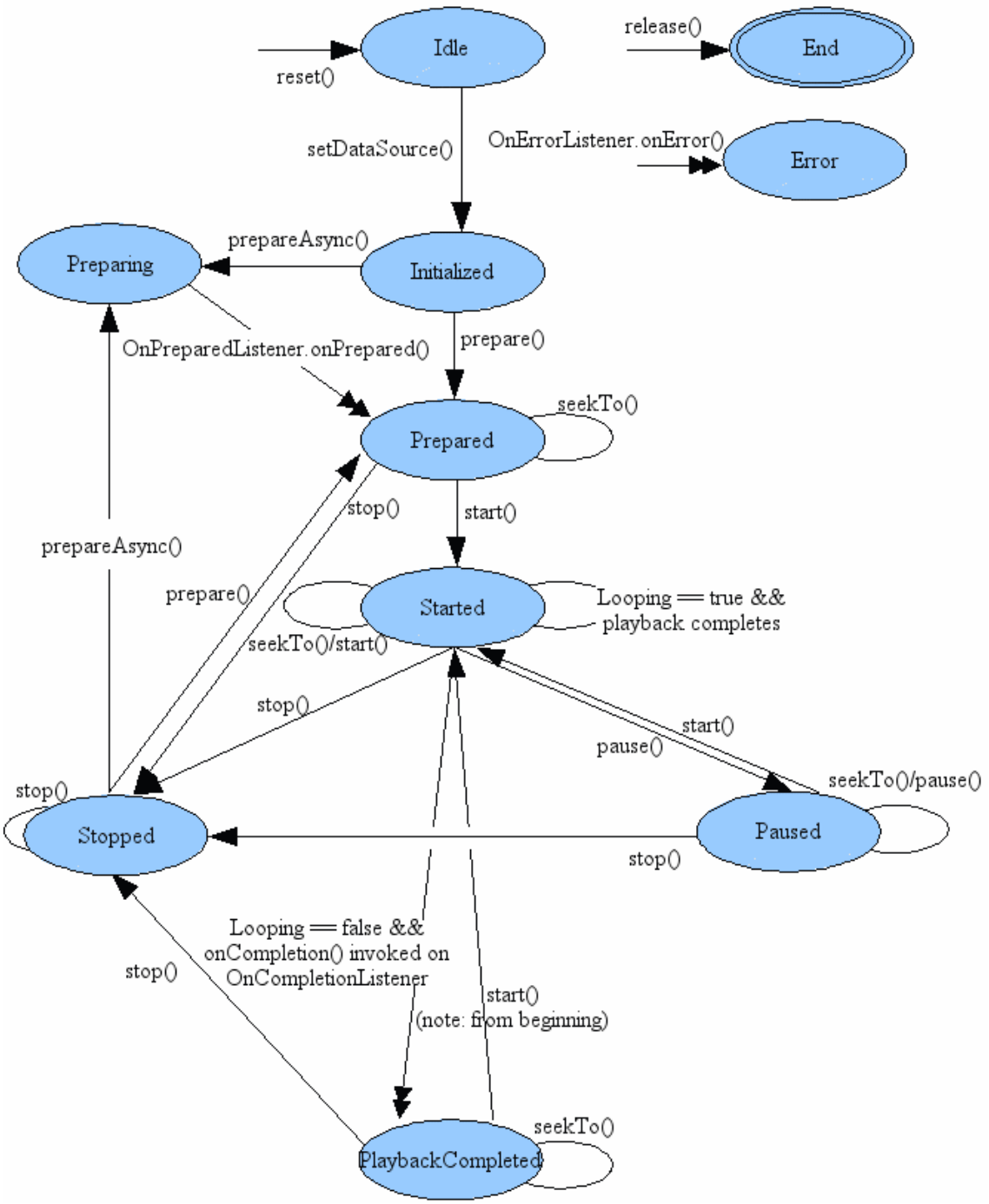

Figure 5.1: A state diagram which shows a MediaPlayer object's life cycle. Taken from Google Android [11].

The Song Scanner Module 6

The song scanner module's main responsibilities are to find music files in specified directories, look up their tempo (BPM) online, extract album art from the files, and lastly save them in the database if they do not exist already.

6.1 User Stories

Title:	Locate music files on device.
Priority:	High
Story:	As a user, I, want the application to search the directories I specify for music files.

Acceptance Criteria:

- When I start the application for the first time, it finds the music files stored in /sdcard/Music.
- I can add or remove to the directories the application searches for music files.

Title:	Obtain tempo for song.
Priority:	High
Story:	As a user, I, want the application to automatically obtain the tempo (BPM) of the music files located.

Acceptance Criteria:

- Obtain 113 as BPM for Rick Astley's "Never Gonna Give You Up", when locating the 12 music files stored in /sdcard/Music.
- If no BPM is found for a song, it will not be playable.
- I am able to manually set BPM for a song.

6.2 Implementation

The Song Scanner is part of the `DataAccessLayer` module, which consists of three classes `Song`, `SongDatabase` and `SongScanner`. `Song` holds information about songs. `SongDatabase` is an abstraction of our SQLite database. `SongScanner` is responsible for keeping the database and the music files in the selected folders synchronised.

6.2.1 Song

An instance of `Song` represents an music file loaded into memory. The `Song` class consists of relevant data fields, such as `file path` and `BPM`, as well as constructors whose parameters are used for the database and song scanner.

Most of the information needed for a `Song` instance are contained within the music file itself in the form of `ID3 Tags`. An informal ID3 standard is specified by Nilsson [16]. The constructor used by `SongScanner` takes a music file as a parameter, and extracts data (title, artist etc.) from the file.

6.2.2 SongDatabase

The `SongDatabase` is used to interact with our SQLite database. The database contains a single table for songs. This table is used to store all data for a song. In addition to the data extracted from the file, we also store a path to a cover image, and BPM.

`DynamicQueue`, described in section 5.3.1, relies on `SongDatabase` to handle queries for songs with a BPM within a specific range. In the event a row in the database does not have a corresponding music file, the row is deleted. This can happen if a user no longer wishes to keep a file in his or her collection, and removes it from the selected folders.

6.2.3 SongScanner

The `SongScanner` has two tasks to perform; scan for files and find BPM online.

The first task involves a recursive search though selected folders for music files, constructing `Song` objects, and then insert them into the database. After the `Song` object is constructed the `SongScanner` attempts to extract a cover image from the file, and save the cover in a separate file. It then adds the cover's file path to the `Song` object. This behaviour reflects the idea that the cover image was supposed to be downloaded online. This feature was not implemented, as we found out that it is common to store the album cover in the file.

The second task is finding BPM for songs in the database. This is done by querying for all songs with an unknown BPM, and then searching online for it. We used an API from EchoNest [6], to perform a lookup using the artist name and song title. This is repeated every time the application is launched for songs with associated BPM.

The Step Counter Module 7

The step counter module's responsibility is to count the number of steps a user takes with a smartphone, using a built-in accelerometer. In our implementation of the step counter, we decided to split the task into a data collection task, a testing of algorithm task and an implementation-on-Android task. The reason for doing so is that it makes sense to test different algorithms on the same data set, if they are to be compared.

7.1 User Stories

7.1.1 Step Counter

Title:	Detect the pace of the user.
Priority:	High
Story:	As a user, I, want the application to detect my pace and display it on the screen.
Acceptance Criteria:	• When I run the screen displays my SPM.

Title:	Select song based on pace.
Priority:	High
Story:	As a user, I, want the application to select song to play based on my pace.
Acceptance Criteria:	• When I run the application automatically selects songs matching my pace.

7.2 Technology Exploration

7.2.1 Data Collection

As there is no built-in step counter library available for Android API version 16, we had to choose an algorithm. We found the precision could vary greatly from algorithm to algorithm, so we had to compare algorithms to each other. In order to do so we first had to generate some sample data collected from the smartphones' sensors.

The first step of data collection is to figure out what data to collect, to do this we take a look at the available sensors on an Android smartphone, Google Android [12] gives an overview of these. We decided to gather data from three sensors: accelerometer, gravity and gyroscope. We also measured the time delay between each sensor reading.

7.2.1.1 Collection Procedure

To generate the sample data we had a person perform the following tasks while having the smartphone strapped to the arm.

1. Walk 100 steps. *Walk: to advance on foot at a moderate speed; proceed by steps; move by advancing the feet alternately so that there is always one foot on the ground.*[1, Walk]
2. Jog 100 steps. *Jog: to run at a leisurely, slow pace.*[1, Jog]
3. Run 100 steps. *Run: to go quickly by moving the legs more rapidly than at a walk and in such a manner that for an instant in each step both feet are off the ground.*[1, Run]
4. Sprint 100 steps. *Sprint: to run at full speed.*[1, Sprint]
5. Alternating. Walk 20 steps followed by jogging 20 steps, running 20 steps, sprinting 20 steps, and walking 20 steps.
6. Standing still with the phone for 2 minutes.

The purpose of task 1-5 is generate data the algorithms should analyse and correctly calculate the 100 steps taken. The data from task 6 should be interpreted as 0 steps taken.

These tasks can then be repeated for all carrying positions, different smartphones and multiple persons. In practise the tasks were carried out with two people, the smartphone Samsung Galaxy S III and in one carrying position, strapped to the arm.

After the data collection, we are now ready to test algorithms against each other.

7.2.2 Algorithm Selection

To compare algorithms against each other we looked at the precision with which an algorithm could predict the number of steps taken in a data set from a particular task. The algorithm we found to work best was an adaptation of the pedometer described by Zhao [20].

Algorithm Modification The algorithm described by Zhao [20] is for a pedometer working directly with the ADXL345 chip. We, however, have to rely on a sensor listener provided by the Android platform instead. This means a few differences in our measured data:

- Our data sampling frequency is significantly lower.
- Our data is filtered for noise.

7.3 Implementation

The step counter is implemented as a single class extending the `Service` class in Android. The service is started and bound from the `MusicPlayerActivity`, meaning in our implementation it is launched and closed together with the `MusicPlayerActivity`. On its creation the `StepCounterService` registers the accelerometer sensor in order to read data required for step detection.

7.3.1 Step Detection

To determine whether a step is taken, the approach taken by Zhao [20] is implemented. Our implementation first calculates a threshold, which corresponds to the threshold line in Figure 7.1. The threshold is calculated by subtracting the minimum accelerometer measurement from maximum measurement, from the array where the measurements are stored.

According to Zhao [20, p. 2] a step has occurred if there is a negative slope in the acceleration graph and the acceleration curve crosses the threshold. It is determined whether a negative slope has occurred, by comparing the latest accelerometer measurement with the previous one. If the last measurement was above the threshold, and the current is below the threshold, a negative slope which crossed the threshold and thus, a step has occurred.

As Zhao [20, p. 2] we assume that a person can either run as fast as five steps per second or walk as slowly as one step every two seconds. This is handled in the implementation by checking the time since the last measurement. If less than 200 milliseconds have passed since the last step, a new step is not detected. If more than 2 seconds have passed the SPM array is updated with a 0 to indicate that the person is not moving (0 steps per minute).

Our implementation utilises a sensitivity limit so measurements which are very close to each other are not detected as a new step, even though they fulfill the requirements mentioned earlier. This is done to filter out small deviations in measurements which might falsely be identified as a new step. The current limit value is 2, and was found experimentally.

7.3.2 Calculation of Steps Per Minute

The gab between steps are used to calculate the number of steps the user takes per minute (SPM).

First, the amplitude is calculated which represents the change x, y, z values obtained from the accelerometer. The formula used for this is

$$amplitude = \sqrt{x^2 + y^2 + z^2}$$

The array containing accelerometer measurement data is then updated with the newly calculated value. The data in this array is used for determining whether a step is taken or not, the implementation of which is described in Section 7.3.1.

If we detect a step we look when the previous step was taken and calculate SPM from that, this value is then stored in the array containing the SPM measurements. Afterwards, the average value of the SPM array with the new value is calculated, and the GUI is updated with a new value through a GUI manager.

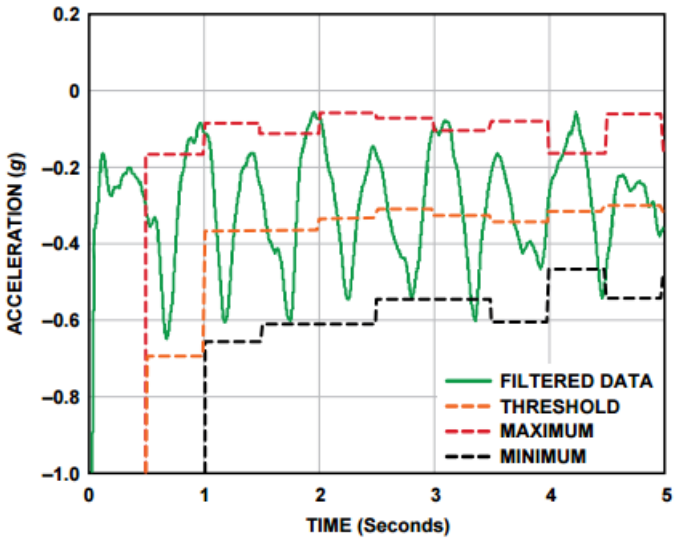

Figure 7.1: An acceleration plot from Zhao [20, p. 2] showing filtered data from a pedometer worn by a person walking.

If no step is taken and more than 2 seconds have passed, the implementation interprets this as if no steps are being taken and the SPM array and GUI is updated with the value 0.

8 Non-graphical User Interface

The non-graphical user interface (NGUI) provides a way of using the application without having to look at the screen. As mentioned in Chapter 1, this is a required feature as the user is likely not able to look at the screen while running, but they still need to operate the application.

Several approaches can be taken when it comes to creating a NGUI, e.g. an interface which is controlled by voice, taps on the screen, or gestures, can be implemented. Taps were chosen over gestures, or voice control as they are easier to perform while running.

8.1 User Stories

The user story on which the implementation of the non-graphical user interface was based on is:

Title:	Enable navigation with screen turned off
Priority:	High
Story:	As a user, I would like to save power by turning off the screen, while still being able to navigate songs when the screen is off. A screen is turned off when the power button is pressed.
Acceptance Criteria:	• Play, pause, stop, previous and next functionally should still be available after the `power` button is pressed.

8.2 Implementation

The implementation of when the screen is turned off, is a pseudo-implementation. On stock Android we found that it is not possible to keep the Android listener running, when the screen is turned off e.g. by the power button. If this functionality should truly have been implemented, the phone would need to be rooted.

In the current implementation the screen is made to look dark, by creating a completely opaque overlay. The NGUI can be activated by pressing the menu button on the Android device. After the overlay is activated, the screen turns dark, and the application starts handling taps in a custom manner as described in Section 8.2.1. Two classes are involved in the implementation of the NGUI:

- `ControlInterfaceView`, the dark view responsible for handling touch events.
- `TapCounter`, counts taps and call the corresponding method, e.g. one tap is for `pause/play`.

The overlay, and thus our custom handling of taps, can be deactivated by touching the back button again after which the opaque overlay is removed.

8.2.1 Handling of Taps

The detection of taps is done by listeners provided by the Android platform. Two types of taps are handled. The two types are: a single tap, which is when the screen is touched for a brief period and long press, which is when the screen is touched for more than 500 ms. Long press is solely used for the `stop`, while single tap used for `play`, `pause`, `next`, and `previous`. Single tap calls an increment method on the `TapCounter` class.

This method does two important things: it increments a global tap counter, and it starts a timer. The timer runs in the background, which enables new taps to be registered. Every time, within 500 ms of the last tap, a tap occurs, the global counter is incremented. After the time has passed, a method, `doTapAction()` is called with the accumulated number of taps as a parameter. The method then decides whether to play, pause etc., and performs the associated response, seen in Table 8.1. The 500 ms were determined experimentally.

Action	**Response**
`Single Tap`	`Play` if paused or stopped or `Pause` if playing.
`Double Tap`	`Next`, skips to the next song.
`Triple Tap`	`Previous`, plays previously played song.
`Long Press`	`Stop`, stops the song and resets the progress.

Table 8.1: Tap actions possible while NGUI is activated.

After an action has been performed, the global tap counter is reset, and the NGUI goes back to the state it was in before any taps occurred. The application is ready for a new action.

9 Test

Testing is important in software development. It is critical because it is a way of ensuring the software developed is of a certain quality. This can either be the quality of the code itself (unit tests) or the degree to which it fulfills a customer's requirements (acceptance tests). Additionally, if used correctly, testing can verify that a change to the code does not break existing functionality.

9.1 Acceptance Test

As a way of making sure that a user story is fulfilled, Wake [17, p. 126] suggests that an acceptance test should be written. An acceptance test describes how the application should perform in a given situation for a user story to be complete, and the progress of the project can be measured by the amount of passing acceptance tests.

The acceptance tests should be written by the customers, possibly with some help from a dedicated tester, to make sure the application does exactly what they expect it to do. The acceptance tests should preferably be automated, and if necessary they can be implemented by a programmer. The idea of making the acceptance tests automated, is to make it possible to use them as regression tests. This makes it possible to verify that refactoring and/or changes to other parts of the code, did not break the functionality of features that have already been implemented.

The important thing when writing an acceptance test is to make sure it is completely unambiguous and reproducible. An example of a user story and the associated acceptance test could be:

Title:	Keep playing when screen is off.
Priority:	High
Story:	As a user, I would like to save power by turning off the screen while the music is playing.
Acceptance Criteria:	• The music keeps playing after the `Power` button is pressed.

- Precondition:
  - The application must be launched in the MusicPlayer Activity.
- Procedure:

  - Press `Play` button.
  - Verify music is playing.
  - Press `Power` button.
  - Verify screen turns off.

- Postcondition:
  - Verify the music is still playing.

9.2 Unit Test

The project was developed using a test-driven approach (TDD). This means that unit tests were written *before* any actual implementations were written. The purpose of writing tests before any actual code was written, was to ensure the quality of the code, and make sure that only the needed functionality was added. So when a new feature needed to be added, a test was written, run and it should then (as expected) fail. Then, just enough functional code was written to make the test pass.

Another purpose of using unit tests, is regression testing. When a change is made to the code base, things can easily break - especially in a complex system. When unit tests are in place, a change can be made and the unit tests can be run to check whether the system still passes all the tests, after then changes have been made. This of course requires a test framework to be in place. In the project, Android's JUnit extension from Google Android [14] was used.

An example of how a unit test was used, is when we wanted to make sure the code, which loaded a song from the database, functioned correctly. The test approach looked as follows:

- Load valid song
  - Verify whether valid song is loaded
- Load invalid song
  - Verify that invalid song is not loaded
- Load song with null value
  - Verify that song is not loaded

So if, at some point, someone refactors the method which loads a song into the database and accidentally removes the part of the code handling a null check of the song, the unit test will fail. This is because the code no longer has the same functionality as the previous version of the code. Therefore, the code change needs to be rewritten allow the same functionality as before the change.

9.2.1 `Reflection` for Private Methods

In our case, we deemed it necessary to test private methods. This came to be because we had an exception in a private method, and we decided to write a test to catch this exception. There is some debate as to whether it is good practise to test private methods or not, but ultimately we decided to do it. It could be argued that it is the over-all behaviour of the system, and not the implementation which should explicitly be tested. In that case, testing of the public interface and not private methods should not be done.

9.3 Implementation Observations: Unit Tests

Unit tests were a good way to allow refactoring, since it made it easy to ensure the software functionality was not altered after a refactoring. However, that required extensive unit tests, which we did not have in some cases. This caused some trouble in form of recurring bugs. To solve this we could create a kind of testing convention, so for example when a method takes parameters, its test always makes boundary, and null checks of the parameters.

Boundary testing is for example when a test case for a method uses inputs just below, at and above the lowest and largest limits of the possible input values its parameters can have. This could for example be `INT_MIN` and `INT_MAX`, if a method has a parameter which is an integer.

We also found that even though unit tests enable regression testing to be performed, the degree of quality assurance provided by the tests, rely wholly on the quality of them. If the unit tests for a particular piece of code were not correct or written thoroughly, an alteration of the code which broke something in the system, might not cause the unit test to fail, even though the behavior of the system has changed.

Even though the tests might take a while to write, we found that they were generally worth the time to write. The reason for this was that some of the time invested in writing them was regained later because the tests caught errors that we would otherwise have spent a lot of time debugging to find.

Refactoring 10

In this project we experimented with refactoring as a scheduled task. The goal of refactoring is to make code more readable and more maintainable. We encountered some bad smells in our code, such as code duplication, and we tried to remedy them in our refactoring task.

10.1 Bad Smells

According to Fowler [9], a bad smell is

> *.. a surface indication that usually corresponds to a deeper problem in the system*

Examples of bad smells are duplicated code, and long methods. Below each of them are explained.

Duplicated Code is then when similar code structure occurs in multiple places in a program. Most of the time there is no reason to have nearly identical code in multiple places. Often, the code can be extracted, with little or no modified and a new method can be created which is simply called from all methods with the similar code.

Long Methods are likely to be harder to understand, since readability also affects maintainability it is clearly an unwanted smell. Therefore it is usually desirable to break up a long method into multiple smaller ones. Usually all that is needed is to shorten a long method, is to extract the method and put the parts that seem to fit together in new methods.

10.2 Implementation of Refactoring

After a review of the project's source code, we created a list of the bad smells identified. Of these, the two most frequently occurring were

- Duplicated code
- Long method

In the following subsections, concrete examples of how we dealt with them are illustrated.

10.2.1 Refactoring of Duplicated Code

The method in Listing 10.1 shows a method which suffered from the duplicated code smell. As can be seen in the sample, the two for-each loops perform identical operations on a list. They both iterate over a list and if an element from that list is contained in another list, it is removed from that list.

```
1  public List<Song> getMatchingSongs(int num, int thresholdBMP){
2      ...
3
4      for (Song song : _prevSongs){
5          if(songs.contains(song)){
6              songs.remove(song);
7          }
8      }
9
10     for (Song song : _nextSongs){
11         if(songs.contains(song)){
12             songs.remove(song);
13         }
14     }
15     ...
```

Listing 10.1: Example of duplicate code before refactoring.

Listing 10.2 shows the method after method extraction has been performed on it, and two method calls placed in place of the duplicated code.

```
1  public List<Song> getMatchingSongs(int num, int thresholdBMP){
2      ..
3      removeDuplicateSongs(_prevSongs, songs);
4      removeDuplicateSongs(_nextSongs, songs);
5      ...
```

Listing 10.2: Example of duplicate code after method extraction from Listing 10.1.

Listing 10.3 shows the extracted method, which performs the same operation as the two identical for-each loops did.

```
1 private void removeDuplicateSongs(List<Song> songList, List<Song>
      songs){
2     for (Song song : songList){
3         if(songs.contains(song)){
4             songs.remove(song);
5         }
6     }
7 }
```

Listing 10.3: Example of extracted method from refactored code.

As can be seen, the refactored method is both shorter, and more readable because it not has a method name which explains exactly what the for-each loop does. Thus, both readability and maintainability has been improved.

10.2.2 Refactoring of Long Method

The long method shown in Listing 10.4 is an example from the project source code, which was refactored. It suffered from the long method bad smell. When this smell is found, it can indicate that a method is trying to handle too many responsibilities.

Method extraction was performed on the method, and this resulted in three new methods. As can be seen in Listing 10.5 the code from lines 2 - 4 in Listing 10.4 were extracted, to a new method with minor a modification.

```
1 final public void selectNextSong() {
2        if (nextSongs == null || nextSongs.size() == 0) {
3            nextSongs = getMatchingSongs(_lookAheadSize,
                 _BPMDeviation);
4        }
5        if (nextSongs.size() == 0 && prevSongs.size() > 0){
6            prevSongs.clear();
7            selectNextSong();
8        }
9        if (nextSongs.size() == 0){
10           return;
11       }
12       if (currentSong != null){
13           prevSongsSizeBeforeAdd = prevSongs.size();
14           prevSongs.add(currentSong);
15           if (prevSongs.size() > _prevSize){
16               prevSongs.remove(0);
17           }
18       }
19
20       currentSong = nextSongs.get(0);
21       nextSongs.remove(0);
22       nextSongs.add(getMatchingSongs(SINGLE_SONG, _BPMDeviation).
             get(0));
23 }
```

Listing 10.4: Example of a method which is too long, and suffers from the long method smell.

```
1 private void updateCurrentSongFromNextSongs() {
2     _currentSong = _nextSongs.get(0);
3     _nextSongs.remove(0);
4     _nextSongs.add(getMatchingSongs(SINGLE_SONG, _BPMDeviation).get
          (0));
5 }
```

Listing 10.5: Example of extracted method from refactored code Listing 10.4.

The result of the method refactoring is can be seen in Listing 10.6. The methods it has been refactored into now only handle one responsibility each, and the method in the code sample looks clean, because it mostly contains method calls. It is therefore more readable than the original, unrefactored code, from Listing 10.4

```
1 final public boolean selectNextSong() {
2     if (!refillQueueWhenEmpty()){
3         return false;
4     }
5
6     moveCurrentSongToPrevious();
7     updateCurrentSongFromNextSongs();
8
9     return true;
10 }
```

Listing 10.6: Example of extracted method from refactored code in Listing 10.4.

Methodology - Reflection 11

This chapter contains the reflections and discussions about our use of XP in this project. The discussion is centered around the twelve main practises of XP, but with extra focus on two practises that were extra important for the project, namely pair programming and planning game. These two practises are especially interesting because they required us to change our usual way of working considerably, and extra effort was put into following these particular practises. This discussion is based on the meetings held during the project, the summaries of which can be found in Appendix A.

11.1 Pair Programming

Throughout the project we experimented with and observed several interesting things. We experimented with things like *workstation setup* and *scheduled partner switches*. During the project we made interesting observations about *interpersonal compatibility*, *roles*, *quality*, and *productivity*.

11.1.1 Workstation Setup

We have experimented with different kinds of workstation setup. The first setup we used was by exclusively using our laptops without any peripherals, i.e., two people sharing one laptop. This resulted in difficulties for the co-driver, as it was difficult to see what was happening on the screen, due to their low quality. Additionally, due to the size of the laptop, it gave a sense of the driver working on his personal computer while the co-driver just observed quietly in the background, giving little sign of collaboration.

The second setup we used was a purely digital solution, where we used TeamViewer to share our screens. We used this setup to counter the bad viewing angles of the screens, by placing each developer in front of his own screen. Further it should improve the collaboration, by allowing each developer to take an active roll in the development process. We experienced this as a more comfortable way of working, now that we were not slumping over a single laptop. However, it gave rise to the tendency of hiding behind ones screen, even getting distracted and doing non-work related activities, leading to even less collaboration than with the first setup. Consequently, working with TeamViewer proved to hinder productivity more than it helped.

The third and last setup we used was two workstations each consisting of a monitor, a keyboard, and a mouse. This solved some of the problems we encountered with the two other setups: there were no longer problems with bad viewing angles or sitting positions,

and the co-driver could no longer hide behind his own screen. Additionally it removed the need for one person being forced to work on another's personal laptop, which could have been seen as a personal space violation.

Based on our experiences it was clearly the third setup that was the best. It provided, by far, the best work environment, which allowed us to focus on some of the other challenges associated with pair programming. On top of this it improved the communication between the pair, resulting in better quality code, through instant code review and better collaboration. In contrast, a bad work environment, as with the first two setups, makes collaboration harder, which strongly diminishes the benefits from using pair programming.

11.1.2 Scheduled Partner Switches

Switching partners often is an important aspect of pair programming, as it helps getting new eyes on a problem and strengthens the feeling of collective ownership, by having more people work on the same piece of code. For this we also experimented with three different approaches.

In the first approach, we changed partners whenever it felt like a good idea, but with no other restrictions than having to try doing it often, it was easy to forget. This resulted in the same pair often sticking together for one or even several days, which was even more problematic, especially when one of the pairs was stuck with a problem that could easily be solved by changing partners.

To solve the problem of having too few partner switches, we experimented with having a fixed time of the day, where the switch occurred. The change was an improvement in terms of improving the feeling of collective ownership as well as preventing one pair of getting stuck with a difficult problem for too long, but it also resulted in having to split a pair in the middle of finishing a task. Being forced to stop working during a good flow often had a negative impact on the pair, causing them to be less useful for their new partners.

The problems connected to switching partners at a fixed time led us to make a small modification to the approach, resulting in the third and final approach: using a fixed time, but allowing each pair to finish their current task. This approach definitely reduced the problems from the second approach, but it required some extra care in execution, as it would some times lead to one pair waiting for the other to finish a long task. Ideally no tasks should be long enough to have this be a big problem, but in reality a task can easily prove more difficult than anticipated.

Although the third approach provided the best balance of forcing us to switch partners often while still allowing a good work flow, it is not an ideal approach. We are not sure how the ideal approach would look like, but it may depend on the team using it.

11.1.3 Interpersonal Compatibility

Pair programming requires tight cooperation between two people, and for that to run smoothly they need good communication and understanding of each other's way of working. In other words, their personalities need to be compatible. This is not something we have experimented with, but we have made some interesting observations.

In our team, each person generally got along well, but when two specific members were paired up, some problems were observed. The problems did not result in any big

problems, but they did have a negative impact on the productivity of that pair. One of the persons was decisive and responsive, and the other more cautious and reflective. Some times a pair like that can support each other, making up for each other's shortcomings, but in this case their work flow differed in such a way, that they would become frustrated with each other. For example, the reflective person may be driving, and they are trying to solve a difficult problem. The driver sits back and starts thinking about how to solve the problem. Meanwhile, the decisive person already has three ideas that might work, and he is itching to try them out, quickly jotting down some code and hitting the hotkey to compile the program to see if it worked. Eventually the reflective driver takes care to write the right code and slowly moves the cursor to the compile button, to make sure that the solution does indeed work.

Both would likely have solved the problem at the same speed in the end, but now the decisive person is left with a feeling that the process was too slow, and if he had opened his mouth to suggest his ideas, he would have disturbed the reflective person's thoughts, similarly leaving him with a feeling that they were slowed down.

Some of these problems could maybe be alleviated if we had been better at switching roles (i.e., letting the other person drive), but sometimes it can be very difficult to force two conflicting personalities to work together, and we believe it is important to try and make a team where conflicting personalities either are not present, or at least where they are not forced to work together. Avoiding a certain combination in a pair may, in conclusion, be preferable to making sure all possible combinations of pairs are used throughout the project.

11.1.4 Roles

The pair consists of two roles: a driver and a co-driver. The driver takes care of the coding and the co-driver keeps a sense of perspective and continuously reviews the code while the driver writes it. For the pair programming to work it is there are communication between both roles.

The driver role was in charge of writing the code. We, at first, experienced that it would put some pressure on the driver, as he was worried about making mistakes. This problem solved itself with experience and collaboration. Further, the technical skills of the driver, must be relatively high to understand and implement the ideas of the co-driver. In situations where the technical skills of the driver were insufficient, the inadequacy was overcome through communication or a role change was performed.

The co-driver role was continuously reviewing the code and making sure it made sense in the larger perspective. To achieve this, his role was often to initialise the communication. For some it took a bit of practise to be able to break the ice, but once the fear of intrusion was overcome, it was no problem at all. Another difficulty we observed, was when the co-driver was too eager to share his thoughts, that he broke the driver's flow, just to tell him about a typo. This difficulty was remedied through experience, however, we are still not completely able to tell when the driver is "in the zone". We believe with experience this occurrence will be so rare, it will not affect the pair programming much. Another problem we observed was, when the co-driver had difficulty expressing his thoughts, due to the lack of a technical vocabulary. These types of problems were usually solved by writing prototypes or having longer discussion session.

A general problem was the difficulty of instantly being the co-driver after a pair switch. After a pair switch, one would often lack perspective and therefore not always realise the

reasons for the drivers actions. There are two options for solving this problem: either the driver takes some time to explain the situation to the co-driver, or the co-driver spends some time trying to get an overview of the situation, while the driver continues programming. Both of those options do, however, have negative side effects. The former results in a time-wise overhead, and the latter would result in a period where pair programming is not really taking place, resulting in lower quality code, as the co-driver has to focus on understanding rather than providing a code review.

11.1.5 Quality

One of the major arguments for using pair programming is that is improves the quality of the code produced. Although we have no metrics to prove it, we feel like the code we produced was generally of a higher quality than what we usually produce when programming individually. This increase in quality, is partly because of the instant code review given by the co-driver, which results in less need for refactoring, and easier maintenance of the code. Further, the solutions are often more elegant when pair programming than when programming individually.

11.1.6 Productivity

One of the major concerns about pair programming is whether it is productive enough to restrict the work force of two developers for a single story. We are not in a position to provide a definite conclusion on this, but in our opinion it depends on the team using it.

We experienced and rectified some problems with using pair programming in our team. First off, when a problem was outside the expertise of both developers, they would both search the internet for an answer. This would usually lead to both developers reading the same material and experimenting with the same solutions, which is unproductive, since two developers are doing the same task, while one of them might as well have spent their time on another task.

Secondly, we discovered that if part of the pair got distracted, the other would likely get distracted as well. This gave rise to some situations where both developers unknowingly supported each other's unproductiveness. These situations often happened when the co-driver was exhausted and then got easily distracted. To avoid problems like this, both partners has to help each other keep focus, and tell each other to get it together if necessary. The lack of focus may, however, be contributed to a number of different influences, such as too few breaks or too little communication. The underlying issues need to be taken care of on a case to case basis.

While some times having problems with distractions, we more often experienced the pair motivating each other to keep working. Having a partner when working on something gives a sense of obligation, and as long as neither start getting distracted, pair programming works great for keeping a good work morale.

When working on a project, some of the stories will be trivial to implement. In these situations a single programmer could easily write high quality code by himself, making pair programming less beneficial. Determining when a task is too trivial to require pair programming is difficult, and the resulting overhead from trying to determine this, may be better spent on actually implementing the story while pair programming.

It is when implementing complicated stories, that pair programming really shines. It allows for a synergetic effect between the driver and co-driver, allowing them to work

better than the two would do separately. The driver has a sparring partner for helping with ideas, structure, and perspective, and as such he is free to concentrate on the how to implement the story. The co-driver will through communication understand what the driver has in mind, helping him improve his suggestions. We experienced this synergetic effect quite a number of times, leaving us with the feeling of pair programming being more productive than individual programming.

11.1.7 Summary

In summation we used the workstation setup as planned in Section 2.3.7 and compared it to other workstation setups. We tried experimenting with different aspects of the practise, as we had planned in Section 2.3.7. We found the practise to be very dependent on which team is using it. We will therefore recommend others to experiment with the different aspects to find what works for them.

11.2 Planning Game

Planning is a useful tool. We used it primarily as a tool for measuring our velocity, to get an overview of our progress, and to know if we were falling behind schedule. Additionally it helps keep the project progressing as the customer desires, by allowing the customer to continuously create user stories, which should be implemented.

11.2.1 The Customer's Role

The main responsibilities of the customer, in regards to planning, is to create user stories and prioritise in which order they should be implemented. As stated in Section 2.3.6, we used a surrogate customer, which meant we were responsible for the tasks normally appointed to the customer.

The first task we needed to do, was create the user stories. The creation of the user stories went fairly easy, but we later found that these were much larger than desired. We would therefore split them into several smaller, more well defined, user stories. We felt that having an oral agreement was a more flexible tool for creating issues, so a lot of the time the stores were not written down. Taking the time to write the issues down would have helped us make sure, that there was total agreement about what each story should achieve, but it would also result in having to spend more time on rewriting the story if changes occurred, possibly discouraging us from making changes.

After the user stories were estimated it was our responsibility, as surrogate customers, to prioritise in what order the user stories should be implemented. Usually we prioritised them as we suspected an on-site customer would, according to cost/benefit. However, after the basic modules of the application were implemented, we started letting our developer hearts influence the decisions. We started to prioritise the user stories by novelty, which essentially means by what we found new and interesting to implement. This caused the product to be more of a proof-of-concept product instead of the initial intended usable product. Although the product did not end up as what was intended from the get go, it showed us that XP was capable of embracing change.

11.2.2 Making Estimations

To properly plan an iteration, it is important to have precise estimations of the issues at hand. It allows the customer to easier prioritise each user story, and it allows the developers to have an idea of how many issues they can finish during the iteration.

Estimating an user story correctly is, however, a difficult task. Many variables play a role in how long it takes to implement an user story, and it is often impossible to foresee them all. To try to reduce this problem we employed the practise of planning poker, where each member of the team gives their own (anonymous) estimation, and the final estimation is discussed afterwards. We found that in the first iterations we underestimated the amount of time needed for the more difficult issues, while we generally estimated correctly when it came to trivial issues. The underestimation was likely linked to our inexperience, and we tried to bring our new gained knowledge to the next planning meetings. This resulted in us beginning to overestimate the harder stories instead, as we were overcompensating for the fact that we underestimated them before. As time went on, we got closer to being correct with medium hard stories, but the harder ones kept being estimated either too high or too low.

As a way of trying to simplify the estimation process, we switched from estimating a story to take a number of hours to take a number of units instead, each unit being half a day. This allowed us to better put the estimation into perspective in terms of development time, which in turn helped giving us a better idea of how long time a given story should get. Switching to units helped us by making the process of estimating shorter, due to the increased granularity. It did not, however, seem to have a big impact on the precision of the estimations - many of the longer stories still took longer than estimated. It is had to give a perfect explanation as to why that was the case, but one big influence could come from the short total length of the project not giving us enough experience; we only had time to use units for estimations for a few iterations.

11.2.3 Code Velocity

The code velocity is a metric describing how fast the team can implement features. It is an important metric, as it is used to plan what should be implemented next, but also when the customer can expect the product finished. Without an approximate velocity, it is nearly impossible to plan an iteration, let alone a release.

The code velocity is measured, as the sum of the units, for all implemented user stories from last iteration. That does by definition need some way of tracking how much time is used to implement each user story. For the first iteration we manually tracked the implementation time. This method was not anywhere near precise, as we approximated most implementation times, since we forgot to track breaks and the like. This caused us to have difficulty determining the amount of work we could do in an iteration. Therefore we decided to use an automated time tracking tool.

From the second iteration onward, we used the time tracking tool EverHour. This gave us much more precise implementation times, all the way down to the second. However, although our implementation times were much more precise, our code velocity was still off. We found this to be because of the time units used to measure velocity. We had used hours, minutes, and seconds as measurement, e.g. we got an code velocity of 64 hours and 27 minutes. Even by rounded off to the nearest hour, our velocity would be way off.

We studied our user stories and found that their implementation times differed slightly in comparison to the estimated times, but the accumulated difference was quite high. We determined it was the granularity of the measuring units, which were at fault. We implemented the use of half-a-day units instead, and found the code velocity to be quite accurate, now only disrupted by isolated errors of estimation.

11.2.4 Summary

Doing all of these tasks associated with planning takes a lot of time, and the question remains whether it is worth it or not. Based on our experiences, the short answer to that is yes. It *does* take a lot of time, and that time could have been spend on developing more features. There would, however, be a big risk of ending up developing a large number of features that are no longer desired. As mentioned in Section 2.3.8, planning lets you know your velocity, plans for the future, and allows for responding to change rather quickly. This means that with proper planning, it is possible to make sure the customer ends up with a product that fits the needs they have at the end of the project, rather than the ones they had at the beginning.

11.3 Other practices

The following practices were still an important part of our development process, but they have gotten less focus than *pair programming* and *planning game*.

11.3.1 40-hour Work Week

For the duration of the project we tried to keep fixed work hours. Occasionally we worked overtime to get an important task done. We experienced that when we worked overtime, productivity per hour lessened both on the day itself and the next day. There was, however, not a clear correlation between overtime and burnout. We also found that productivity during the day and the next, after working overtime depended on the person and the task being worked on. If the person was interested in the task, more work was done.

Overall, we think that overtime should be avoided if possible, but also that it is sometimes necessary to get a task done.

11.3.2 Code Standard

In Section 2.3 we mentioned that creating a formal code standard would be too time consuming to do, and we ran with this assumption for the first few iterations. After a while, when refactoring and collective ownership became more prevalent, we realised that the lack of a code standard resulted in code some times being slightly inconsistent, potentially making it harder to understand. This led us to talk about creating a more formal code standard, and we began writing it down. In the end, however, we did not spend much time on looking at the finished document, but the thorough discussion of the subject led us to writing more uniform code.

11.3.3 Collective Ownership

For most of the duration of the project, there was not a complete feeling of collective ownership. We attempted to address this by enforcing mandatory commits to the code repository. This, however, was not done consistently because it was not a part of our daily routine. Therefore it was often skipped. Furthermore the inconsistency in coding style

did not help, however this changed after we started focusing on refactoring and coding standards.

In Section 2.3 there were three risks identified related to `Collective Ownership`. Regarding the first problem, lack of expertise, we experienced few occasions where we found ourselves hindered when writing code, and found it necessary to ask other group members for help. This is a sign we had issues sharing expertise about the code base, to all members of the group.

The second problem of violating each other's private space, by editing and crashing the document another developer was using, happened a couple of times. These incidents were resolved without any conflict, although we discovered our test framework was not enough.

The last problem of personal pride did not surface. We believe this was due to us having prior work experience with each other.

11.3.4 Continuous Integration

Instead of setting up a build server, we ran the entire test suite when pushing to the master branch. This worked well when the test suite was small. When it grew bigger and slower, it took too long to run the entire test suite. This meant that only the relevant test modules were run frequently, which meant that the full test suite was run less frequently. Midway through the project we discovered a periodic bug, which was not easy to trace because we needed the log a build server could have generated. This could have given information about which commit caused the bug to appear.

Another benefit of a dedicated build we discovered, is when an addition or change is required to the test set. Our approach was to manually share test data as required by the stories we worked on. This lead to a fragmentation of the test data. This later resulted in surprises when new tests would not run.

One of the purposes of Continuous Integration is to avoid fragmented development branches. This was not achieved for the project, since each task had a branch of it own. In most situations this was not an issue, as we would merge into the master branch when a task was done. The problem arose when a task was big, and it remained isolated on a branch for multiple days.

In order to prevent this problem in the future, we have considered moving day to day work into a single branch. Alternately a similar approach is to force integration more frequently. This would result in more smaller conflicts, but severe fragmentation should not occur.

11.3.5 Metaphor

The application icon, the project title or "pacer" can be seen as metaphors, however we never established a formal metaphor. This was not a problem because every group member had the same vision for the product.

11.3.6 On-Site Customer

For this project we have used surrogate customer, however stepping into the customer role proved difficult. Because we did not properly use a surrogate customer, we did not discuss the project as much as we could have. Additionally we created our acceptance tests late in

the process. They should have been created earlier and as a consequence, they are likely to be based on the implementation.

Overall this has not been a big issue, because we were working towards finishing the Minimum Viable Product. The contents of the MVP was been biased towards novelty, whereas a customer might have been more interested in price, usability, and value (cost/-value).

11.3.7 Refactoring

During the project, we were not very good at refactoring existing code when adding new functionality. As a result, the quality of the code did not live up to our expectations. To solve this, we created a low priority refactoring task, which was estimated to a few hours each iteration. This turned out to be a bad approach since low-priority tasks were usually neglected.

In the group we agreed to remedy bad smells on sight. This, however, was not done because it would have taken a considerable amount of time to do. We further postponed refactoring, in favor of completing the second release faster. This did not help getting refactoring done.

We created a refactoring plan in order to formalise the refactoring process. Bad smells were prioritised and the ones with highest priorities had to be fixed. This provided some concrete goals for what we wanted to achieve, with explicit refactoring tasks. There was a need to standardise the code structure, and spurred on the creation of our code standards. The refactoring tasks were time consuming but did improve the readability and maintainability of the code. New code written after the code standardisation was also of higher quality than the code prior to it. One of the reasons we had a hard time getting refactoring started, was likely the lack of code standards.

We made the mistake of mixing the tasks of refactoring and stub removal, and this increased the complexity and thus the time spent on the task. In hindsight, we believe the process would have been faster if they were done separately.

11.3.8 Simple Design

Overall, we only wrote code for the functionality we needed. As a consequence, a single issue arose where we forgot that a method was not general, and therefore it was used incorrectly in that particular case.

We find that we gained more from this practise than we lost implementing it. At times out habit, we tried to predict functionality in the future, which was never used.

11.3.9 Small Releases

We had difficulties taking advantage of small releases, since there we had no customers to provide feedback between releases.

11.3.10 Testing

Testing helped enforcing several other XP practises, namely refactoring, collective ownership, pair programming, and continuous integration.

The XP practise of collective ownership was enforced when we wrote tests, since there was less risk that a change made by someone in the group broke the system unnoticed, since the test would detect it. This gave group members courage to change code written by others and thus allowed a feeling of ownership of all the code.

Test also enforced the pair programming practise. It gave group members a common understanding of the task to be solved. This understanding was achieved by writing test cases together, before the actual implementation of the application itself began.

Likewise, continuous integration was enforced by testing. All test cases could quickly be run after a change was made, so as to be sure nothing had been broken.

Conclusion 12

In this chapter, we will try to answer the questions asked in Chapter 1. This will be done in two separate parts: one for the application and one for the methodology.

12.1 Application

An Android application was developed in this project to address the challenges stated in Chapter 1. The first challenge was expressed as the following question:

How can we provide music with an appropriate tempo, compared to the current pace, to the runner through the use of a smartphone?

It was found that it is possible to create an application consisting of several modules working together to achieve the desired result, i.e. an application that plays back music that matches the user's pace:

- **Music Player**: This module is the core of the application, responsible for playing the desired music. Using a GUI it is possible to navigate through a playlist manually, and with a combination of the built-in dynamic queue and the step counter module, it can automatically play back songs that match the user's running (or walking) pace.
- **Song Scanner**: The song scanner is responsible for locating music on the device and maintaining a music library.
- **Step Counter**: This module uses the device's sensors to calculate the user's steps per minute.

The user is expected to use the application while running, and in addition to the functionality, it would be advantageous to be able to control the application while using for its intended purpose. That requirement led to the second question:

How can a smartphone application be operated without disrupting the runner's form and/or concentration?

This was achieved by creating a non-graphical user interface, allowing the user to tap the screen, when it is turned off, to control the music player: play, pause, next, previous.

12.1.1 Future Work

In its current state, the application fulfils the requirements stated by the questions, but as with most software there are improvements to make. Some possible improvements are:

Better Flow Because each module was mainly developed on its own, some problems exist with their interfaces. For example, the step counter measures the user's pace, but the dynamic queue does not update the queue quickly enough, so the application only works optimally when running at a constant speed. Additionally, the application currently starts playing when the play button is clicked, instead of when the user starts running, thereby not matching the music to the running pace immediately. These problems could be solved by implementing a smooth transition to a new song when the pace is changed.

Manipulation of Music Tempo To help the application cope with minor changes in pace, the tempo of the song could be changed to match the new pace more precisely. Implementing this change requires a lot of knowledge to not distort the music.

GUI Improvements Many improvements can be made for the GUI, as relatively little focus was put on this aspect of the application. One of the most notable things that would benefit from an improvement is button placement. Especially because the application is meant to be used on the go, it would be an advantage to place the buttons in a way that minimises the risk of clicking a button unintentionally.

NGUI Improvements The current implementation of the NGUI could be improved by adding support for headphones with built-in buttons, and let the user use them to interact, just as with the screen.

Training Programs Using music to steer the user's pace, rather than the other way around, could be a way to create training programs. These programs could be interval training, with musical indications of when to speed up and slow down, or it could be possible to use the tempo of a song to challenge the user to run just a little bit faster.

Integration with Other Services Integrating the application with for example a streaming music service would allow having access to more music with less hassle. Allowing the user to pick out some specific genres or even individual tracks would improve the experience even more.

Polishing Existing Features Many current features can be improved. For example it could be made possible for the user to input the BPM of songs with missing data. An idea that would provide more features with minimal programming could be to add an option to use the step counter and music player on their own, i.e., let the step counter output a count of total steps instead of only steps per minute, and allow using the music player with a traditional play list.

12.2 Extreme Programming

We chose to use XP because it is a methodology that is aimed towards managing the development process both in regards to organisation and programming. In this section we will try to answer the third problem presented in Chapter 1:

How do we adapt the structured systems development methodology, Extreme Programming, to our project?

Although XP is written by Kent Beck, several others, including himself, have embraced it and modified it, in an attempt to adapt it to their own needs. This means there exist several suggestions as to how to use XP.

We tried experimenting with some of these suggestions, to see how each practise fit with our project. We ended up focusing on the practises *pair programming* and *planning game*, but did of course also make observations of the other ten practises.

We found that some practises influenced each other, and these correlations between the practises made for some interesting possibilities of adaptation. In the end we found a combination which worked well for the project, but many areas still needed improvement to be really good.

In conclusion, XP can be adapted by using it, and it should be continuously adapted to fit the project's needs. There is always something that can be done to make it work just a little bit better, while still adhering to the main practises.

Appendix

Temporary Work Sheets A

A.1 Methodology

Review 1

12. feb - 20. feb Normalt vil sprints være 2 kalenderuger. Dette var lidt kortere da vi havde færre forelæsninger i perioden. Indtil 12. feb. brugte vi på opsætning og planlægning - inklusiv projektidéer.

Vurdering af estimeringer: Vi estimerer trivielle ting fint. Mere komplekse issues estimeres for lavt grundet manglende ekspertise (vi ved ikke hvordan de skal implementeres), hvilket resulterer i at der er brug for at udføre tidskrævende research, samt åbner op for ikke åbenlyse fejl, der er svære at fikse. Vi brugte 50% længere end estimeret.

Hvad har vi udrettet? Vi har lavet de fleste issues - dog mangler vi GUI (#20), database (#16) og song scanner (#17). De tre issues vi ikke nåede blev tilføjet i løbet af sprintet. Equalizer og Playlist er heller ikke lavet, men de var ikke i sprint backloggen.

Programstatus på master Programmet på master mangler stadig gui og database, så selvom funktionerne er der, er der ikke nogen måde at benytte dem på. Appen crasher af og til. Der er meget redundant kode (både i test og produktionskode), ubrugte constructors og metoder.

Retrospective 1

12. feb - 20. feb

A.1.0.1 Coding Standard

We have tried to make the code readable by using field-standards (e.g. _privateVariable). Currently some variable names are too short / undescriptive. We handle standard conflicts when they are discovered. However a general coding standard has been discussed and decided upon. It was decided that it was not needed to make a specific code standard document.

The coding standard contains decisions about brackets.

A.1.0.2 Metaphor

Not really relevant for the project because the only people looking at the code are us (the developers) and the supervisor and censor (software people). We instinctively name the classes and methods based on their functionality, so no problems have yet appeared.

A.1.0.3 Refactoring

We have not at this time had the need or reason to refactor.

A.1.0.4 Simple Design

We have very complicated test cases with much replicated code. This is obviously a bad thing, but we contribute this to our inexperience with testing in Android Studio. We have not prioritised this practise, but we found that the production code (not test) was simple and without replication.

A.1.0.5 Pair Programming

We should limit ourselves to using one computer/monitor and stop using teamviewer. Optimally getting an external monitor to put between us as well as a keyboard and mouse. We have to change partners more often than we did so far. We will be better to do all work in pairs, as we until now have solved trivial problem individually. We have found it difficult to solve some problems in pairs, as pair programming assumes the pair knows about/what to program. We have in these cases assigned the problem to one person (e.g. GUI - no one had any knowledge about the problem, so no matter how long the pair would discuss (about nothing), no solution would be viable. Further the problem of discussing thing you don't know about.)

A.1.0.6 Collective Ownership

We have worked with 'collective ownership-like' approaches before, in that the entire group is responsible for all the code at the exam. Before we however used the practise of not editing (or reading) code written by other group members. We will in future sprints incorporate reading the code produced by others and review or refactor it if necessary.

A.1.0.7 Testing

Sometimes we forget to test first. This is bad - we should be more aware of testing first. We should be better at using setup and teardown. We have to focus on having the tests act as a specification. We should not put much effort into "test-to-fail". If it is trivial (boundaries of int, string, etc.) it is okay, but we have to let the tests drive our development rather than hinder it.

A.1.0.8 Continuous Integration

Every time we merge with master we should run all tests and make sure everything runs. Also run the app and make sure it does not crash or have other issues. We have in this sprint solved some assignments right after another without pushing to master or creating new branches (Play/Next/Prev all in one branch because it was easy). We shall be better to push to master when an assignment is solved and create a new branch for the next assignment.

Review 2

23rd February - 6th March The first day was spent on reading up on XP. We did not label our issues on github with iterations. We should do that (we have the paper issues).

Estimations:

- We had trouble properly timing our tasks, so we started using everhour on the last day.
- All work related time is tracked.
- We will start counting work hours per person rather than per task.
- We will not use estimations/time trackings from this iteration in our estimation of the next.

What Did We Accomplish?

- We got stuck for a while on GUI - there were many unpredicted problems.
  - The basic GUI almost works for now, and effort should be greatly reduced, so we can focus on other parts.
  - We should have split this issue up in several subtasks.
- Overall we have a functioning (though unstable) music player.
- About half of the tasks for this iteration were not done.

Status on Master

- It is possible to get the coverflow out of sync.
- App can crash if next/prev keys are spammed.
- Overall functioning well enough for demonstration purposes.

Retrospective 2

23rd February - 6th March

A.1.0.9 Coding Standard

If we have any conflicts we will write it down in a coding standard document. We agreed with Ivan that we basically have a "de facto" (informal) coding standard by working together so long.

A.1.0.10 Metaphor

Ivan argued that our problem statement was a sort of metaphor, and we should not discard this practise entirely (as we suggested). The system could also be seen as a pacekeeper and/or a personal trainer.

A.1.0.11 40-hour Work Week

About halfway in the iteration we found that we would not be able to finish all tasks and we decided to take a few late days (to get just a little bit more done - we did not expect to finish all tasks either way). We found that the productivity was relatively low after hours and the quality of work was so low it was redone the next day. Further we found that the day after was less productive as we were tired. The fact that the next day was lecture and we worked on report (which is very boring) could have influenced to process to the worse. We found that working until dinner was of average productivity, but the productivity and quality seriously dropped after dinner.

Breaks:
We have experienced that some are too intrigued by the problems at hand that they 'forget' to take a break and stretch their legs. This have led to people 'burning out' before the end of the work day. Another reason to improve breaks is that sometimes people easily gets distracted while researching. It is then important that we We should regard each other, so we do not interrupt a member which are in the zone.

A.1.0.12 On-site Customer

After consideration about the responsibilities of a customer, we decided that Niels (Dan's friend) would not have enough time to fulfil the role. We have therefore decided that we will completely use a surrogate costumer.

A.1.0.13 Small Releases

We have not implemented this practise yet.

A.1.0.14 Planning

We have experienced that we have an informal prioritising system, but this can cause problems when choosing new tasks, as these are chosen based on the developers curiosity and interest. The priorities of the tasks should therefore be defined by implementing a stack-like structure in order to ensure the most important tasks are done first.

We had problems with timing our productive work. To solve this we found the time tool Everhour. This will then help us be more precise about estimations. We decided to discard the old estimations and measurements due to their imprecision and to avoid 'muddying' our future estimations and measurements. (Note: we now use the combined time of two developers for estimation and measurement.)

In this iteration we mainly worked on one large task (GUI), which should have been decomposed into many smaller tasks. Further we should be better to create new tasks instead of just added found issues and development to a todo list.

XP makes use of the planning game for estimation of use cases. We first knew the game as an individual exercise (explained), but later found, in the planning XP book, a collaborative version was described. We made/make use of planning poker as it has some advantages over the individual version of planning game. We later found that planning poker was very similar to the collaborative version of the planning game.

A.1.0.15 Refactoring

We found that our quality is not good enough, so we will in the future have explicit tasks for refactoring. For now we will assign a number of hours for next iteration, but after that we will write a new issue for refactoring when we discover code smells in areas that are not part of the current issue and these will be estimated and prioritised for the following iteration (maybe they will be fixed as parts of other issues). Code smells in methods relevant to the current issue will be fixed on sight.

A.1.0.16 Simple Design

Same situation as iteration 1. We should look into this maybe probably...

A.1.0.17 Pair Programming

We are no longer using teamviewer. We are working on getting monitors. We should remember the dialogue when working - looking at the driver is not always enough.

Pair programming expects people to know what they are doing - trying out new/unknown code can be difficult. In the future, when in need of research, we will accept breaking with pair programming, whereafter each developer will try things on his own. When a solution is found, the pair will form again and continue from where they left off.

A.1.0.18 Collective Ownership

We are still in the mindset of 'I wrote it, it's my code'. This has resulted in some methods not being refactored because 'It was [name]'s code, I better not touch it'. We will of course try to break this mindset by enforcing refactoring.

A.1.0.19 Testing

We have mainly developed GUI in this iteration, so we have not created many tests, as we have no idea to automatically test GUI. We also already decided to not focus alot on GUI, so we argue that GUI tests are less important.

There were of course made tests for the changes in dynamic queue and database.

A.1.0.20 Continuous Integration

As mentioned, we have mainly worked on GUI this iteration. This means that we have used the "#20 GUI..." branch as a surrogate master branch. We have done this because the GUI was not ready to be pushed to master. By assuming the GUI branch was master, we have used continuous integration daily, as we merge and build it multiple times a day. Further we find this practise to be beneficial in larger projects with multiple teams, as we are only a team of 4, we almost never work on more than two branches at the same time.

A.1.1 Review 3

9th March - 20th March Summary of extended meeting at the bottom of this document.

Estimations:

- We started using Everhour to track hours
- We recorded 50 hours (45+5) but counted there would be 168 hours in the iteration. After subtracting time used for stand-up meetings, supervisor meetings, lunch breaks, and small breaks we have about 1 hour per person per day that is not used for anything productive. The time is most likely spent on a combination of general project discussions, switching tasks, and procrastination/not starting when a break ends.
- Some days we did not do anything project related - either because of lack of motivation, illness, or course related stuff (that should have been done at home).
- Our estimation of trivial tasks were approximately twice as long as the actual time.
- We had some tasks take much longer time than estimated, given that we ran into problems with a library.
- In the future we will talk about possible solutions for each task in order to estimate risks to according to possible problems.

What Did We Accomplish?

- We made a music library and a working test suite, further we made it possible to test private methods.
- Made a configuration table
- Wrote report.

Status on Master

- Some song scanner capabilities are added.
- App can crash (it always could, but it needs to stop.)

A.1.1.1 Extended Meeting

Minimum Viable Product Antagelser:

- 70 timers arbejde per iteration. Dette svarer til ca. 20 units.
- Der skal skrives en rapport.
- Der skal laves en app.

MVP:

- Appen skulle kunne tælle skridt.
- Appen skulle kunne tilføje sange til sangbiblioteket fra en brugervalgt mappe eller standard mappen (Music).
- Appen skulle kunne matche en sang til et tempo.
- Appen skulle kunne kontrolleres til at kunne: Play, Stop, Pause, Next, Previous. Uden brug af en tændt skærm.
- Appen skulle kunne afspille musik og hvad der dertil tilhører (skift til næste efter slut, etc.).
- Det er antaget at MVP er funktionel fra armen.
- Appen skal kunne automatisk hente BPM data om en sang fra internettet.

Protocols: New Issues, Switching Issues, Changing Issues During an Iteration Snak og undersøg om det er 'lovlig' at skifte opgave uden den igangværende er færdig. Snak og undersøg om det er 'lovlig' at ændre på issues der er aftalt (samt hvordan dette kommunikeres effektivt).

If an issue (without any relation to existing issues) is discovered:

- If it is very important, discuss it amongst the group and decide whether it should replace an existing issue.
- If it is not important, the issue is added to the next iteration planning.

When choosing a new task to work on, it is important to select tasks in progress, if any. This can be understood as issues in progress are prioritised highest. Remember it is okay to 'take' tasks from other members if they do not work on the task assigned to them.

If (part of) an issue is deemed not important to the project, it should be discussed and agreed between ALL members of the team. If not all members are present, the missing members are contacted to set of a meeting. If no response, then pause the issue and discuss it when response.

Pushing to Master and Fixing Crashes How do we make sure the maser is in a good place.

Fix master (crash without test files). It is very important that we test more comprehensive tests, e.g. input null, "", -1, file exists. Further it is important we check to see if the tests are comprehensive before we push to master.

ps. read about practise of deleting branches.

Releases Aftal dato for næste release.

Release should contain:

- See Software Innovation 3 - Configuration Table
- Step Counter + Music Player + Song Scanner
- 10 April. (next iteration end)

Report Tilpas konfigurationstabel. Lav toc.

Architecture Modularisering af projektet (med interfaces?)

We will refactor and create an architecture after release.

Retrospective 3

9th March - 20th March

A.1.1.2 Coding Standard

In methods it is done as:

```
1 if (statement) {
2     //Do stuff
3 }
4 else {
5     //Do stuff
6 }
```

Listing A.1: Coding standard for statements and loops.

Trivial getters and setters (or other methods that simply do a return) should be on one line.

Use long variable names please. So no 'am' for AudioManager, use 'audioManager' as variable name.

A.1.1.3 Metaphor

Nothing in particular for this.

A.1.1.4 40-hour Work Week

We have encountered sickness, so we have not even reached the 40 hours.

A.1.1.5 Small Releases

We have decided to ignore our earlier "release" in order to plan a real release at the end of the next iteration (10. april).

A.1.1.6 On-site Customer

We have made preparations for us to handle the On-site Customer role by simulation.

A.1.1.7 Planning

We don't see the point of individual estimations as individuals rarely finish issues alone.

If an issue (without any relation to existing issues) is discovered:

- If it is very important, discuss it amongst the group and decide whether it should replace an existing issue. The issue is then estimated by the pair handling it.
- If it is not important, the issue is added to the next iteration planning.

If an issue (with relation to existing issues) is discovered:

- It should be estimated by the pair which handles it.

A.1.1.8 Refactoring

Given that we will be examined in the code, we have decided to refactor more than suggested by XP. Further we have decided to refactor and make a new architecture after the release. This architecture should be of a simple design. See simple design.

A.1.1.9 Simple Design

We have found that the individual 'modules' in our code are not strongly defined. We plan on solving this by implementing each module through interfaces - thereby giving a clear overview of the public methods for each class and the inputs and outputs for each class and method. This will improve the independence of each module making it possible to change an entire module without affecting the overall program.

A.1.1.10 Pair Programming

We found that the forced timed pair programming switch didn't sit right with us. We experience multiple times that we should switch just before the current issue was done. This created a lot of overhead. We therefore decided to use a task-based approach where switches only are made between issues or between issues estimated to take more than 3 hours.

We have used the practise of, when in problems, splitting the pair, where each partner then makes some prototypes to solve the problems. Then the pair reunites and solves the problem together.

We have had some sickness this iteration. This meant that we have not been pair programming when the members were working from home. This was not a problem since the tasks solved by individuals were sufficiently trivial. This gives rise to the question of when a task is trivial enough to not pair program.

A.1.1.11 Collective Ownership

We are still stuck of the old method of blaming others and being defensive of own work. We will once again try to better ourselves.

A.1.1.12 Testing

We found our previous testing method was insufficient in regards to private methods and we changed the method so private methods are now tested. Also we need to be more aware of testing first, as we use TDD. We need to be more comprehensive when testing, and not only sticking to a specification approach. e.g. input null, "", -1, file exists.

For boundary tests as input it is okay to test them by creating an array with all the desired values, whereafter an loop iterates over and calling the method with all the values.

A.1.1.13 Continuous Integration

We have had some problems making sure the master branch is stable. See the extended meeting from Iteration Review 3 (Pushing to Master and Fixing Crashes). This has been a problem all the time, but the issue was not caught until now.

A.1.2 Review 4

23rd March - 10th April Summary of extended meeting at the bottom of this document.

Estimations:

- We expected to work 70 hours, 67 hours on code and 3 hours on report. It turned out we worked 71 hours in total, 68 hours on code and 3 hours on report. This estimation is concluded to be spot on, as the measurements have small variations, which can make up for the extra hour.
- We have solved issues for 82 estimated hours in this iteration (non report only, nor extra bug fixed (4 hours)).
- Based on estimations of the issues of the iteration, we can conclude that when an issue is estimated between 8 (10) and 20 hours, the time actually used is approximately 60% of the estimated time. However when we estimate non-trivial short issues (2-4 hours) we estimate fairly low in comparison to the actual used time. i.e. two issues estimated to 2 hours took 8 hours and 3,5 hours respectively.
- The small deviation in time between estimated and actual used time, can be traced to the granularity of our estimations. We have a granularity where we estime in actual hours in up to 2 hours. More than 2 hours are estimated in units of half a day. 4 hours are half a day, 8 hours are an entire day, 10 hours are a day and a half ... We used these numbers because of the restrictions of the online planning poker tool used. This actually works okay, as it gives the opportunity to estimate trivial issues to less than half a day, which obviously will give a lot of overhead.

What Did We Accomplish?

- We finished the step counter
- We fixed the out-of-memory issue with the album covers.
- We created the GUI for settings and SongScanner.
- We fixed a number of small bugs.
  - Next song after completion.
  - Handle missing songs.
  - Stop button skipped to next song.
  - Set filename as song title if no other title is available from the MP3 file itself
- We now only need the following to finish our MVP
  - Acceptance tests
  - Control without looking at screen
  - Finish BPM online tests
  - General unit tests
  - Settings for the step counter, i.e. sensitivity

  - Refactoring. Modulation, and correct encapsulation.

Status on Master

- Working and tested
  - Step Counter
  - Music Player
  - Song Scanner

- Issues still on master
  - Step Counter
    - Works as intended
  - Music Player
    - Invalid state exceptions still happen on master (warnings in log)
    - prev and next pauses song when playing, it should continue playing the new song.
    - Seekbar seekTo functions correctly, but have display issues jumps when dragging.
  - Song Scanner
    - Missing online BPM lookup in Song Scanner.
    - Song Scanner Settings not used in app.

Retrospective 4

23rd March - 10th April

A.1.2.1 Coding Standard

In this iteration we decided to postpone the decisions and documentation of the coding standards to next iteration, because we intend to allocate time for refactoring next iteration.

A.1.2.2 Metaphor

Given the size of the project and the fact we all have worked together before, we are generally on track with the unspoken metaphor, which is essentially just what the program is. i.e. the music player is called music player. Besides the small team makes it easy to communicate if and when conflicts of understanding occur. All this makes the metaphor an implicit understanding between the team members.

A.1.2.3 40-hour Work Week

Besides single and uncorrelated episodes, we have not had any problems with energy. Some of this energy can also be attributed to the fact we have mostly coded and not written much report this iteration.

We suspect this situation can change in the coming iteration, due to the fact we are going to write a lot of report, which is boring and tiring. We will reflect upon the results in the next iteration retrospective.

A.1.2.4 Small Releases

The end of this iteration is our first real release. It is not a small release, but the app was not in a "releasable" state previously. We plan on making smaller releases from this point on.

We do not have any actual customers to show the release to. This have been a factor in the slow release.

Although we have no customer to test and evaluate our releases, we still benefit from doing them. First off, the small release forces us to merge and keep the master up-to-date.

Every merge to master is accompanied with making all test pass. The requirement of small release forces frequents merges to master, resulting in up to date (all the new features) version of the app.

Further, using small releases gives greater incentive for the team to develop and complete concrete, delimited functionality as requested or agreed with the customer. i.e. If the customer wants a feature (pacer), this feature (pacer) is developed as a separate module to be changed/improved later without corrupting other modules. The feature is also completed, if possible, before release, so it is not going to dangle as otherwise could happen.

A.1.2.5 On-site Customer

What has the absence of a customer meant for our project? Because we have not properly used a surrogate customer, we have not discussed the project as much as we could have (feedback), and we have not gotten any acceptance tests done. Overall this has not been a big issue, because we have been on track - moving towards finishing our MVP.

How have we dealt with the assignments normally dedicated to the customer?
Prioritising:
We have prioritised use cases sorted by new interesting non-trivial features (novelty), whereas a customer might sort use cases by price, usability, and value (cost/value). This is because we do not gain much from implementing trivial features or nearly identical to existing features.

Acceptance Tests:
We have yet to make any acceptance tests, so this assignment has not been filled.

User stories / Use Cases:
We have not used user stories, instead we have used issues with a small explanation of the functionality. This is a bit like user stories, but not entirely. This has worked fine, as we only have ourselves to answer to.

Minimum Viable Product:
We decided on the MVP based on requirements from the study regulations and our own interests based on novelty.

Have we made decisions or changes which could not have been done by a customer in real life? We decided not to include the interval trainer because it would be trivial to implement, given its similarity to the already existing pacer. A real customer

would might have wanted this interval trainer features, because of its high value and low cost. As explained, we prioritised based on novelty and not cost/value, hence this decision.

A.1.2.6 Planning

We estimated 70 hours of work for this iteration, which is a bit more than we spent last iteration, but since we had some problems with people being ill last iteration, we decided to assume we would spend more time, which turned out to be very accurate, as we used 71 hours. We made some notes on planning in iteration review 4 and 5.

A.1.2.7 Refactoring

Not much energy was spent on refactoring - most things were spent on new functionality - because next iteration is going to be dedicated to refactoring. Minor bad smells were corrected.

We plan on making a thorough refactoring plan, for the next iteration, based on Fowler [8].

A.1.2.8 Simple Design

Simple design has been followed fairly well in this iteration, but because of the lack of refactoring some parts of the program are a little more messy than they should be - the plan is to fix this in the next iteration.

A.1.2.9 Pair Programming

After setting up monitors and keyboards our pair programming has improved. Sometimes trivial/small tasks were solved by single persons, but most of the time pair programming is used. Some of us have a tendency to forget changing drivers often, and while we could change partners more often we have done so fairly regularly. We changed partners when it felt natural rather than on set times.

Why do we change pairs with each issue? We experienced that changing from one issue to the next before completing the first, was awkward. This can however decrease the feeling of collective ownership, hence the developer will come in less contact with each implementation.

Why did the other way not work? We experienced that when changing issues, the developers flow could be broken, which would result in wasted (overhead) time.

Is there an overhead when using Pair Programming as much as we do? When should we use Pair Programming? How do we minimise the overhead? Both Beck (find source) and our experience agree Pair Programming results in overhead. However this overhead should be minimised as much as possible. We have therefore decided to not pair program when implementing trivial issues. The triviality of issues is determined by the individual or pair in the process of programming.

Should we change drivers often? We do not think we need a fixed time frame one can be driver, sometimes it makes sense to drive for a long period of time if one possesses a relatively better understanding of the issue. i.e. As Becks example where a experienced person is paired with a inexperienced person. First the inexperienced person observes

and learns what is going on, later the inexperienced person becomes experienced and can contribute and drive. This scenario can occur when changing partners between issues.

In turn, it also makes sense to switch often when there is a similar level of understanding in the pair since it ensures ideas from both programmers will be heard.

What impact has it had not doing so? The idea behind changing drivers is that it enforces collective code ownership (really?). This, however, requires that partner switch is done appropriately so that one person does not remain on the issue all the time.

A.1.2.10 Collective Ownership

Collective ownership has not influenced our project a lot in this iteration, but we have gotten a stronger feeling of the code actually being collectively owned - there is not much feeling of something being someone's code.

We still assign issues based on who have done what By letting an experienced person work with a less experienced person does not require overhead, we reduce the amount of overhead related to having to learn the new worker. It also prevents some errors/misunderstandings associated with having to "learn" the code. We have a tendency to have one person always stuck on the issue, and alternating the partners, instead of letting the first "partner" team up with a different person. This allows the person to follow his idea for a solution to the end, but also makes it feel more like "his" code.

A.1.2.11 Testing

As always we have not been testing the GUI. Our tests have gotten fairly big, and therefore they are beginning to take a while to run (could be solved by using an integration server, see Continuous Integration).

We have had some trouble remembering to do test first - especially when our methods get complicated. In those cases we have sometimes written the tests after finishing the functional code. This has especially been the case when we were unsure what we needed to test (i.e., when we did not know how the method was supposed to work). Basically we ended up using spikes, but instead of throwing out the code we ended up using it and writing tests.

A.1.2.12 Continuous Integration

We are still doing as we did to begin with: Merge with master and run tests. But the tests are starting to take a long time to run. Ideally we want a dedicated integration server, but it is not realistic for us to set one up at this point. In the end we will have to be more selective with when we run our tests.

Is this way of doing it viable? How long is it viable? Why? This way of doing it would not be viable for a larger project, but the cost of setting up an automated build server for a project of this size would most likely be bigger than sticking to the manual approach.

What impact has this way of doing it had? We get very frustrated when shit doesn't work, because it takes forever to do over.

We are experiencing some intermittent exception from an unknown source. This would have been caught by an automated test tool, at least given us a log of when and where it first appeared. Instead we are now using time debugging everything, this had of course not been acceptable in a larger project.

A.1.3 Review 5

13th April - 24th April Plan:

- Use the length for each unit in hours instead of the hours used in planning poker.
- Based on experience, 70 hours an an itera of work each iteration. continue to plan 70 hours of work each iteration. continue to plan 70 hours of work each iteration.

Estimated Planning Poker Time	Time Allocated in Project and Github
0 hours	N/A
0.5 hours	N/A
1 hours	1 hour (min estimate, trivial)
2 hours	N/A
3 hours	1 Unit - Half a day - 3 hours
5 hours	N/A
8 hours	2 Units - a day - 6 hours
13 hours	3 Units - a day and a half - 9 hours
20 hours	4 Units - two days - 12 hours
40 hours	5 Units - two and a half days - 15 hours
100 hours	6 Units - three days - 18 hours
??	Unknown

Summary of extended meeting at the bottom of this document.

Estimations:

- We did not really estimate anything this iteration.
- We forgot to track the time used for most of the iteration. This was because we refactored loosely and did not create issues for all assignments.
- We used a lot of time on review of retrospective of last iteration, this should be tracked and estimated as well.

What Did We Accomplish?

- Overall this iteration was much less productive than planned - we expected the more "boring" tasks to result in less productivity, but we were unable to handle it in a satisfying way.
- We made a refactoring plan, ensuring somewhat similar refactoring patterns.
- We refactored a small part of the program.
- We fixed (worked around) a periodic bug because the cover flow hangs, which we solved by just turning off the screen.
- We discovered that we have not quite tested well enough.
- All in all we have been unproductive and have been too easily distracted.

Status on Master

- Some issues were resolved on master
  - online BPM
  - periodic crash doing tests

- Issues still on master
  - Music Player
    - prev and next pauses song when playing, it should continue playing the new song.
    - Seekbar seekTo functions correctly, but have display issues jumps when dragging.
  - Song Scanner
    - Song Scanner Settings not used in app.
  - Tests are incomplete
    - BPM in DB not tested

Retrospective 5

13th April - 24th April

A.1.3.1 Coding Standard

We have expanded our coding standard by writing down the general structure and bad smells. Otherwise we have used verbally agreed upon standards.

It has become clear this would have benefited us much earlier in the project, essentially saving much of the time spend this iteration on refactoring. Though some of the considerations done to reach this structure, was not possible to consider before encountering the problem. However, we now this is a good structure, hence it can be used from the get-go in future projects.

A.1.3.2 Metaphor

We discussed metaphor at the supervisor meeting. It was noted that both the program title and program icon can be seen as metaphors. This is due to the fact the title and icon should illustrate the purpose and functionality of the program.

We have not made any changes to our previous title or icon, but upon reflection it is likely we change the title in the future.

We have encountered several issues when writing the report. Many of these issues, and the difficulty getting started, can be contributed to the missing metaphor or vision for the report. Contrary to the program, we do not share, or have discussed, any common metaphor of the report

A.1.3.3 40-hour Work Week

The 40-hour work week practise is used to combat burnout. In this iteration we clearly worked much less than 40 hours a week, however, we have still experienced burnout.

One major culprit is when one is not sure about what to write (typically report) or what to do. Last retrospective we assumed the lack of energy when writing report was solely due to lack of interest. Now we believe it is both because of the lack of interest, lack of shared vision, and bit because of our pre-assumed idea of the report writing being boring.

Another reason for our burnout is the lack of shared vision as explained in metaphor last paragraph.

Further we did not plan thoroughly, hence a lot of work needed to be done before actually starting on an issue. The has caused many to be less enthusiastic and just not start working on any issues.

This has also shown us how important planning and structure really is.

A.1.3.4 Small Releases

Although creating a release last iteration, we have not used it for anything at all since. This is due to the fact we have not gotten any external testers or customers to test it.

The reason for us not taking the role as surrogate customer and testing it, is we have tested the shit out of it when creating it and every time we debugged and ran it, hence we did already know how and where it works.
This leads to a very low usability, given that novelty is prioritised higher. The scale is very much tipped in favor of novelty, given we have no user to complain.
This decision is not made actively, but because we are heading for an exam and not actually distributing the program, we have experienced that novelty is rated higher. Furthermore, the only usages of the program is done by us at the demonstration, hence the low limit of usability.

If we should have followed XP to the letter, we should have taken the time to (user) test it thoroughly as a customer.

A.1.3.5 On-site Customer

When taking the role as on-site customer, we have a tendency of polluting our decisions based on what is interesting to develop. We, as a surrogate customer, focus more on what is interesting for our exam than on things that may be interesting for an on-site customer.

e.g. Usability issues are prioritised low.

The acceptance tests, we should create as a surrogate customer, will be done as a mini project in TOV. The reason for us to focus on acceptance test in TOV, is because we cannot reflect upon it before trying it, and it is an important part of testing in XP.

A.1.3.6 Planning

Time Tracking: We started tracking time for a couple of iterations ago.

Why time track? The initial reasons for time tracking was estimation improvement and code velocity. Since then we have tried to use the metrics for measuring actual work hours and conclude and improve upon these, if possible.

What is relevant to time track? We have discussed whether we should time track meetings and the likes. On one hand these meetings can be used to measure actual work hours, hence improving the general working speed. On the other hand it may cause more overhead without any insurance it actually can improve general working speed.

A thing worth noticing is each second is time tracked, but this can be misleading and lead to nitpicking instead of what is important. To counter this, Bech through XP, suggest to measure in units (of half a day). This is however only recommended to code issues.

Big Brother vs. Anarchy Since we started tracking time we have found more and more things we wanted to track. This can result in more overhead (actual "real time" and mental), and more importantly in having a work environment where Big Brother "Everhour" watches your every step if we do not set a line.

The problem with logging everything is how conclusions are drawn. If one concludes that the best programmer is based on "Lines of code per minute" or similar, is that it can be very misleading. In this case there is no thought about quality, hence the real best programmer might be assessed to be a lousy programmer.

Some members forget or didn't bother to start the time tracking tool. This could be caused by laziness or because they felt it was not really necessary. This could also be due to the fact we are not used to tracking our time with a tool.

The bottom line is that time tracking should improve our process.

Planning: We decided not to estimate or create issues for the refactoring and report process, due to not knowing what was to be done.
As explained above, this decision was cause for some problems with concentration and energy. In the future we will therefore create issues for all processes and estimate all code issues, to counter this problem.

In summation: We should actually plan an iteration before doing it.

A.1.3.7 Refactoring

We refactored as an activity this iteration in spite of what XP tells us.

We created a refactoring plan to ensure consistency across all classes and coders.

We did this because of the reasons stated in retrospective 4.

We did not refactor as much as planned, but we got a clear idea how to increase the quality of all our code.

We ended up doing some things that could be considered functionality while refactoring, e.g. we removed stubs and connected some unconnected parts of the code (that should be connected). This made sense for us, however, as it clears up almost finished functionality without making big changes to functionality.

A.1.3.8 Simple Design

Since we have not really developed anything new, we have not used this a lot.

We have removed a few things that were not "simple design" when refactoring, but it seems like we have not been fully aware of what "simple design" actually means.

"Put in what you need when you need it."

A.1.3.9 Pair Programming

We are not sure when we gain something from pair programming - some trivial code could be refactored by one person, but maybe doing it in pairs will give a better end result. We will experiment with different settings in the last few iterations.

note: only relevant due to we refactor as an activity.

A.1.3.10 Collective Ownership

Refactoring as an activity has improved our collective ownership, as more people get to go over the code that has been written by others.

Refactoring, while implementing, code written by other pairs would have the same effect, but we noticed that people often steer away from changing code that was not written by themselves.

In conclusion: Collective ownership is hard, but gets better over time.

A.1.3.11 Testing

Adding test case after finding a bug, why is this not done?We did this to a small extent but we are not consistent with it and at this point we might as well not. Since the extra quality we get from writing a test for this does not justify the time it takes. We would gain experience with testing against bugs, but we do not learn much about process, increase the quality of the rest of our code nor benefit from it at the exam, since very few supervisors and censors read it.

We should evaluate the coverage of our test: We should remember to compare our coverage with how many bugs we actually find. Find appropriate methods from test and verification course.

Needed tests were not made, discuss! We found bugs that should have been caught by tests (such as does this feature even do anything)

Because we have problems writing a sufficient amount of tests, we should figure out a solution that could solve this. Because of time we will most likely not implement the solution. We could supply some test templates/specifications, telling us how to test certain types of methods, i.e., all methods with parameters should at least have a boundary test.

We will not not specify test templates at this stage of the project, but we will review our current tests and discern if any is missing.

The specific cases of missing test cases will be examined to determine if they actually are missing or if there is a test, which does not test correctly.

A.1.3.12 Continuous Integration

We still have the same issue of tests taking a long time, but after working around some trouble with the UI it is now significantly faster, and the lack of integration server is not as big an issue as before.

A.2 Refactoring

Song - Class

We refactored the song class and we found these bad smells.

1. Constants was written without '_' as whitespace. Was fixed by inserting '_' as whitespace.
   e.g. MS_PER_SEC

2. We have restructured the class into five sections. This is done because we had much trouble determining which methods were public accessible and which was wrongly made public. Due to this we named the five sections:

   a) Private Shared Resources

   b) Accessors

   c) Constructors

   d) Private Functionality

   e) Public Functionality - Interface

   Other classes may still have a section called 'Stubs and Drivers' in the top of the class. Each section is encapsulated with a '//region' and '//endregion' making it collapsible. e.g.

```
1  ////////////////////////////////////////////////////////////
2  //                  Private Shared Resources              //
3  ////////////////////////////////////////////////////////////
4  //region
5    Code...
6  //endregion
```

   Listing A.2: Example of code section.

3. Refactored the method `getDurationInMinAndSec` to use ternary operators. This reduced 6 lines of nearly identical if-statements to to lines using ternary operators.

4. In `getDurationInMinAndSec` we found that the method used the class' own getter to access the variable `_durationInSec`. This has been fixed by accessing the variable directly.

Bibliography

[1] Dictionary.com. http://dictionary.reference.com, 2015. [Online; Accessed 2015-06-22].

[2] Ivan Aaen. Essence - pragmatic software innovation, 2015. Unpublished Book.

[3] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Longman Publishing Co., Inc., 1st edition, 1999. ISBN 0-201-61641-6.

[4] Kent Beck and Martin Fowler. *Planning Extreme Programming*. Addison-Wesley Longman Publishing Co., Inc., 1st edition, 2000. ISBN 0201710919.

[5] Dr. Paul Dorsey. Top 10 reasons why systems projects fail. 2005. URL http://www.ksg.harvard.edu/m-rcbg/ethiopia/Publications/Top%2010%20Reasons%20Why%20Systems%20Projects%20Fail.pdf.

[6] EchoNest. echonest. http://developer.echonest.com/index.html, 2015. [Online; Accessed 2015-05-22.].

[7] Judy Edworthy and Hannah Waring. The effects of music tempo and loudness level on treadmill exercise. *Ergonomics*, 49(15):1597–1610, 2006. doi: 10.1080/00140130600899104. URL http://dx.doi.org/10.1080/00140130600899104. PMID: 17090506.

[8] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., 1st edition, 1999. ISBN 0-201-48567-2.

[9] Martin Fowler. Codesmell. http://martinfowler.com/bliki/CodeSmell.html, 2006. [Online; Accessed 2015-05-22.].

[10] Gartner Inc. Why critical program management is the right decision. http://web.archive.org/web/20070308151701/http://www.gartner.com/it/products/consulting/critical_program_mgmt.jsp, 2007. [Online; Accessed 2015-05-13 but originally showed on 2007-03-08.].

[11] Google Android. Mediaplayer. http://developer.android.com/reference/android/media/MediaPlayer.html, 2015. [Online; Accessed 2015-05-21.].

[12] Google Android. Sensors overview. http://developer.android.com/guide/topics/sensors/sensors_overview.html, 2015. [Online; Accessed 2015-04-21].

[13] Google Android. Api guide glossary. https://developer.android.com/guide/appendix/glossary.html, 2015. [Online; Accessed 2015-05-18].

[14] Google Android. Testing fundamentals. http://developer.android.com/tools/testing/testing_android.html, 2015. [Online; Accessed 2015-05-19.].

[15] Eva Johnson. Scrum, scrumban and extreme programming. http://intland.com/blog/sdlc/extreme-programming-scrum-scrumban/, 2014. [Online; Accessed 2015-05-27.].

[16] M. Nilsson. Id3 tag version 2.3.0. http://id3.org/id3v2.3.0, 1999. [Online; Accessed 2015-05-19].

[17] William Wake. *Extreme Programming Explored*. Addison-Wesley Professional, 1st edition, 2001.

[18] Don Wells. Extreme programming: A gentle introduction. http://www.extremeprogramming.org, 2013. [Online; Accessed 2015-05-26.].

[19] T. Yamamoto, T. Ohkuwa, H. Itoh, M. Kitoh, J. Terasawa, T. Tsuda, S. Kitagawa, and Y. Sato. Effects of pre-exercise listening to slow and fast rhythm music on supramaximal cycle performance and selected metabolic variables. *Archives Of Physiology And Biochemistry*, 111(3):211–214, 2003. doi: 10.1076/apab.111.3.211.23464. URL http://dx.doi.org/10.1076/apab.111.3.211.23464. PMID: 14972741.

[20] Neil Zhao. Full-featured pedometer design realized with 3-axis digital accelerometer. http://www.analog.com/media/en/technical-documentation/analog-dialogue/pedometer.pdf, 2010. [Online; Accessed 2015-04-21].