Tempo Player - The music player that fits YOUR tempo

Ivan: Metaphor - Think about more suitable title.

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 - The music player that fits YOUR tempo

Subject:
Mobile Systems

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

Project group:
SW802F15

Participants:
Alexander Drægert
Christoffer Nduru
Dan Petersen
Kristian Thomsen

Supervisor:
Ivan Aaen

Printings: ?

Pages: ??

Appendices: ?

Total pages: ??

Source code:
`https://github.com/SW802F15/SourceCode`

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 tempo. The project is based on the research of Edworthy and Waring, which concluded that music tempo affects a runner's pace and enjoyment.

Preface

Aalborg, May 21, 2015

Alexander Drægert

Christoffer Nduru

Dan Petersen

Kristian M. Thomsen

Contents

Introduction 1

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. [11] which suggests, that *"... music had no impact on mean power output."*. Yamamoto et al. [11] 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. [11] 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 properly, the runner would have to stop running, or focus more than normally, which 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. [9] *"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 [6] *"... is one of the critical success factors in a systems development project."*. Based on this knowledge the third problem can be stated:

How do we select and implement a structured systems development methodology?

Alexander: Read last problem and compare current version vs previous.

Methodology 2

In this chapter, the agile software development methodology, extreme programming is explained. Extreme programming is then compared to the SCRUM. Extreme programming was used in this project and an explanation of how it was applied is also contained in this chapter. Lastly, the modifications made to extreme programming to fit this project are described.

2.1 Extreme Programming explained

Extreme Programming (XP) is a software development methodology created by Kent Beck, and this section is based on his book *Extreme Programming Explained (1999)* [5]. Extreme Programming is supposed to be a lightweight, efficient, and fun approach to developing software.

The methodology consists of the 12 practices:

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

Some of these practices are more engrained than others. While some supports each other, others may be unrelated. (i.e. coding standards supports collective ownership, but on-site customer and pair programming are unrelated.) The complete map can be seen here:

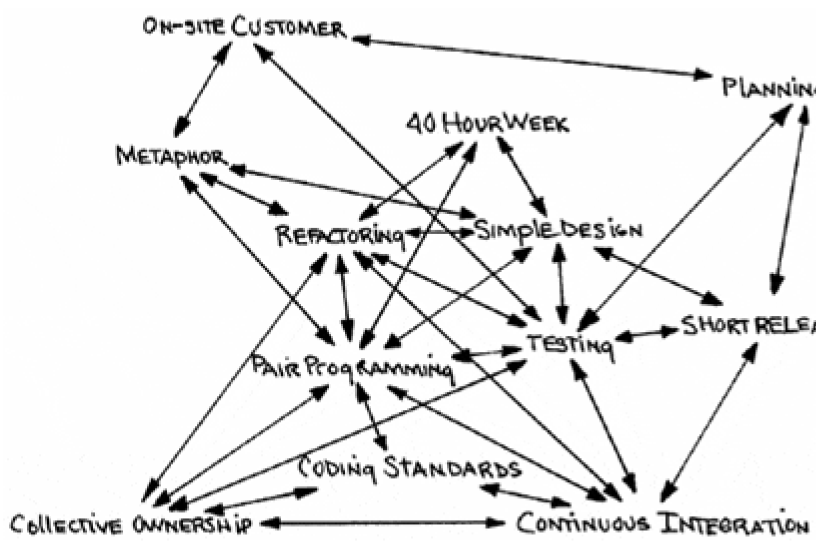

Figure 2.1: Map, from page 70 in *Extreme Programming Explained (1999)* [5], of the practices and how they support each other.

2.1.1 40-hour Week

This practice is created to ensure the developers are well rested and fresh when they come in to work.

As the name suggests, the practice encourages developers not to work more than 40 hours per week. Although this practice is called 40-hour Week, it should not necessarily be specified to 40 hours per week. The correct week length will depend on the specific team.

This practice is basis for many of the other practices, as one needs to be well rested to think straight and be creative. On the other hand, for this practice to work it is required that there are enough tasks to solve. These tasks are created through the Planning Game practice, hence making it a prerequisite for the 40-hour Week practice to work.

2.1.2 Coding Standards

When a project is written by many different developers, it is important that the code are easily understood by other developers. Otherwise it would waste a lot of time every time a developer has to extend existing code.

The practice is all about writing code in a agreed upon format. This ensure the code are consistent and easy for all developers to understand.

This practice encourages the Collective Ownership, Pair Programming, and Refactoring practice. By making it impossible to distinguish between who wrote what code, it eliminates the tendency to say “I don’t understand this, it is not my code”. Instead the developer instantly understands the purpose, as it is written in a format he understands. Pair Programming in and of itself is challenging enough. If you additionally do not understand each others way of coding, it becomes almost impossible. A mutual Coding Standard helps by eliminating this problem.
It is much faster to refactor a code snippet when you understand its purpose right away.

2.1.3 Collective Ownership

Extreme Programming strives to be efficient. Therefore it has adopted a collective approach to ownership, instead of the individual approach. Where the individual approach enforces stable code, it also leads to slow changes and less code understanding. In contrast the collective approach enforces rapid changes and high quality, given it is used correctly and does not become “when everybody is responsible, no one is responsible”.

The Collective Ownership works by allowing everybody the rights to change and extend everything. However, before any changes are released it has to pass 100% of the automated tests. If it does not pass, the changes will be discarded, and you try again or let someone else try.

For this to work, it is highly dependent upon the Testing practice, as it is crucial to have very good tests. Otherwise the changes might break the system and this risk increases every time a change is introduced without being tested.

2.1.4 Continuous Integration

To avoid wasting time on developing fragmented versions of the same functionality, it is important to integrate at least once a day, often more frequent.

When a pair is done with a task, they sit down and integrate it. They then make sure all unit tests run at 100%. If the unit tests does not reach a 100%, they know it is their task that caused this. It is therefore up to them to fix it or discard their solution and try again.

It is evident that this practice requires the Testing practice to work.

Alexander: Spørgsmål til Ivan: Hvilken struktur er bedst, den lange eller den korte?

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 to intuitively understand the purpose of the variable, method, or class.

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

Pair Programming is the way to code in a 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. The 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 estimates them according to cost.

Refactoring is the practice used to eradicate bad smells in the code, such as duplicate code.

Simple Design is the best design, and is defined as the easiest that work. For it to work, it must pass all unit tests and acceptance tests. Further the best design should be so simple it never has any duplicate code.

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

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

2.2 Extreme Programming versus Scrum

The two are quite similar, but there exists some significant differences.

The most significant difference is that Scrum does not have any development practices. XP, on the other hand, have plenty of development oriented practices, such as: pair programming, test-driven development, refactoring, automated testing, simple design, and so forth.

Another difference is the flexibility of the iteration or sprints, as they are called in Scrum. In Scrum a sprint is between two week to one month in length. When a sprint is planned and started, the sprint backlog is not susceptible to change. In XP the iterations are between one to two weeks in length. 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 Modifications of eXtreme Programming

Our modifications of XP. Practical applications.

e.g. Code Standards are not as important in our project because we are familiar with each other.

p.s. remember to mention how we have used `Essence` retrospective methods.

p.p.s. remember to mention that we use TDD, because it is referred to from Test - Unit Test.

Tempo Player - The Tour 3

3.1 The Tour

Ivan: Hvis vi vil starte med et overblik, bør vi medtage en brugskontekst - hvad er det egentlig man kan med det? Det kunne også give mening at give et overblik over programmet.

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. It uses the accelerometer for calculating the user's running pace. The application's main screen can be seen in Figure 3.1.

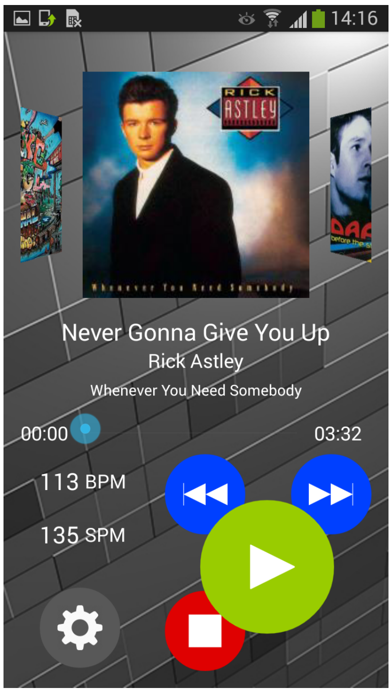

Figure 3.1: A screenshot showing Tempo Player's main screen.

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 steps per minute.

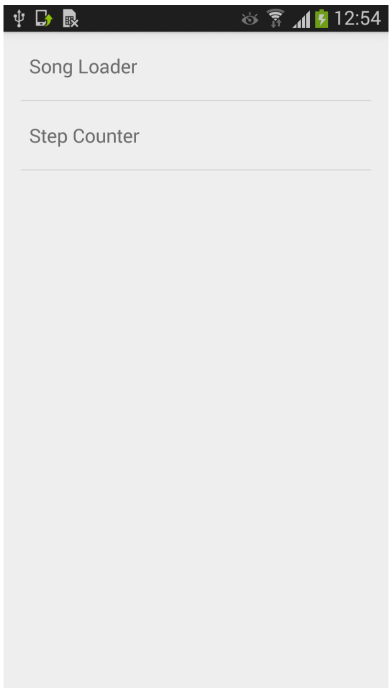

Figure 3.2: Screenshot showing the settings screen.

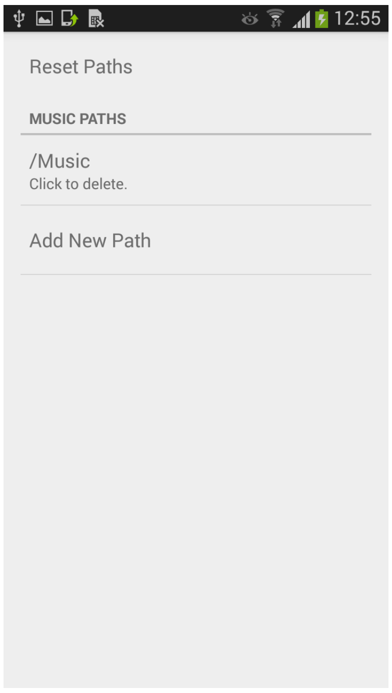

Figure 3.3: Screenshot showing the "Reset paths" screen of the settings screen.

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 with an allowed deviation of 45 BPM.

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.

We have chosen that 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.

Iterations and Releases 4

Over the course of this project we been working in iterations of two weeks, which is approximately one week worth of work on the project. In this chapter we will cover the goals and findings of each iteration. Doing the project we had three releases each covering two iterations.

Kristian: should be expanded

4.1 1st Release

Dan: General stuff about the release here

4.1.1 1st Iteration

The goal for the 1st release was to create a music player on the android smartphone platform that should be usable while running.

Kristian: consider if we should have use cases here.

The tasks of the 1st iteration involved:

- Playing MP3 files located on the device.

  Stop/Pause functionality.

  Playing should continue when screen is turned off.

- Selected MP3 files to play based on beat per minute.

  Replay previously played songs and skip a song and select a new.

4.1.2 2nd Iteration

4.1.3 Methodology Observations

Aside from developing the product, Tempo Player, a lot of energy was spent on considering how XP could be used to make the development process as efficient and controlled as possible. We had a clear goal to create high quality code while also keeping track of the development in a way that would allow us to estimate how long a given addition to the project would take, and decide if it is feasibly to include it in the project based on that estimation.

Dan: Should we mention this “clear goal” before? (We had a clear goal....)

In this and the following Methodology Observations sections, we will present some of the more interesting observations we made while working on each release. As stated in LABEL TO RELEVANT XP SECTION all the XP practices are important to gain as much as possible from XP, and all of them were considered (as apparent in the unedited meeting summaries in Appendix A), but three of them are especially interesting when it comes to reaching our goal: pair programming, planning, and refactoring.

Dan: Find LABEL TO RELEVANT XP SECTION

Initially, during the first few iterations, we had to take the first steps towards using XP. This did not only include reading Beck’s book

Dan: Beck’s book

, but also adapting and getting used to the XP practices, so we sat down and pretended to be the customer, talked about some user stories, wrote them down the best we could. We looked at the user stories, turned them into issues, and started a game of planning poker. The issues got an estimation and a priority, and we started working - we did not know how many hours we could actually expect to get done in an iteration yet, but we knew we would not get in trouble with our customer, as it was, in fact, ourselves.

Dan: Did we explain planning poker?

Ivan: Overvej hvor det er bedst at placere metodologi-observationer. Hvis vi bliver som vi er nu, så skriv observationer der passer til perioden og byg løbende en forståelse op, som bliver analyseret til sidst. Fortæl det gerne som “historier”, så det er lidt spændende at læse.

4.2 2nd Release

With the basic music plater application completed in the 2nd iteration we outlined the configuration for the 3rd and 4th iterations in the configuration table seen in 4.1 taken from **?**].

- Paradigm Focus: The challenge is to improve running experience by matching music beat to pace. The application is supposed to be used while running.
- Paradigm Overview: Stakeholder is the user who is using the application.
- Paradigm Details: For this release we have a two main scenarios to fulfil. First we want the application to automatically select appropriate (songs with a matching beat) songs to play as the user runs at varying paces. The song pool to select from should be customisable for each device.
- Product Focus: The application is first and foremost a running pace, meaning the purpose is to match music beat with steps per minute (pace). It is possible to add an option for interval training so the application no longer adapt music beat based on pace, but encourage the user to adapts his or her pace to predetermined intervals of

	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 tempo.	*Facilitation* Focus on immediate benefits to user.
Overview	*Stakeholders* Runner.	*Design* Running Pacer with music player-like design and functionalities.	*Elements* Grounds: Paced music gives better running experience. Warrant: When running, it is human nature to match pace with the music playing. Qualifier: Detailed music information (bpm) required. Precise SPM measurement required. Rebuttal: Use context: To match songs with pace. Limitations: Step counter measures step frequency.	*Evaluation* Procedure: Iteration retrospective by surrogate customer Criteria: Evaluate immediate functionality based on acceptance tests.
Details	*Scenarios* Automatically fades into songs, which fit running pace. Use private collection of MP3 files as a basis for exercise/running.	*Components* Music player. Music library. Step Counter.	*Features* Running pacer. Music player. Step counting.	*Findings* Extracting beat pr minute information from MP3 files is not practical. Controlling the device while running can be difficult.

Table 4.1: Table taken from

low and high beat music. Additional training programs could also be added. Further more the application can also server as a step counter not playing any music just recording steps taken and displays steps per minute, and the other way around just playing music without taking steps per second into account.

- Product Overview: The application design should be based on a "familiar music player design", it should have the expected functionally from a music player and new elements (such as steps per minute) should expand on the familiar design.

Kristian: consider "familiar music player design"

- Product Details: This configuration consists of 3 components the music player, a 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 MP3 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 the beat of songs to played with running pace.
- Project Overview: Grounds are described in chapter 1.
- Project Details:

  Kristian: this seems similar to *Affordance*

- Process Focus: The evaluation is based on benefits to the user(runner).
- Process Overview: We evaluate the application as a surrogate customer at iteration end/review. The evaluation is based on the fulfilment of acceptance tests.
- Process Details:

4.2.1 3rd Iteration

4.2.2 4th Iteration

The 4th iteration is where the 2nd release where completed

4.2.3 Methodology Observations

4.3 3rd Release

Dan: General stuff about the release here

4.3.1 5th Iteration

4.3.2 6th Iteration

4.3.3 Methodology Observations

The Music Player Module 5

Behind the scenes, Tempo Player consists of a number of modules, each playing an important role in the overall functionality of the application.

In this chapter the Music Player module is described in terms of user stories and implementation, as well as observations made in regards to XP during development of the module. In Chapter 10 reflections are made on these observations.

This chapter also contains a brief explanation of Android terms which are necessary to understand this chapter.

5.1 Android Terminology

When explaining the implementation of the modules, some Android specific terms will be used, so a very short description of the terms needed to understand the explanations are listed here. These explanations are based on the Android API Guide Glossary [3], the Android Fragments guide [1] and the Sensor Overview [2].

Activity

A screen in the application implemented as a Java class with layout defined as XML. It can handle UI events and call methods.

Fragment

A fragment is intended to handle a part of the behaviour and UI of an Activity, especially useful for creating multi-pane UI's and adjusting the application to run on devices with different screen sizes.

Service

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

Sensors and Listeners

The phone has 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 some of 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 .mp3 files" without the Song Scanner, which is described in Chapter 6.

Title:	Play .mp3 files.
Priority:	High
Story:	As a user, I, want to listen to the .mp3 files stored on my device.
Acceptance Criteria:	• I am able to play a .mp3 file stored on the device. • I am able to regulate the volume when listening to the song. • I am not able to open other file types than .mp3.

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 speed.
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 to most notable are `DynamicQueue`, `MusicPlayerActivity`, and `MusicPlayerService`. `DynamicQueue` handles navigation, and it makes sure the queue of songs matches the user's running speed. `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 runnin speed of the user.

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. Is starts the `MusicPlayerService`, calls the `DynamicQueue` for songs to play, attaches listeners to the buttons, making them clickable, and it checks that there are enough songs in the specified folder, sending the user to the settings menu if this is not the case.

Because this class is responsible for managing so many things, that it was necessary to move some of its functionality to another class, `Initializers`, to prevent `MusicPlayerActivity` from becoming too confusing. `Initializers` is responsible for all the methods to set click listeners, and initialize the dynamic queue and the cover flow, which shows the album cover of each song in the queue.

Dan: Please rewrite this to make more sense.

5.3.3 MusicPlayerService

Dan: Fit this stuff in:

We found that trying to stop the music player while not loaded caused it to skip to next song. It was due to the fact this was an illegal action in that state, as seen in Figure (Android graph over mediaplayer states). We fixed this issue by implementing something, as seen in Listing (Code snipped of interesting code).

Song Scanner 6

In this chapter, the development, implementation and relevant observations in regard to XP made while developing the song scanner are described.

The chapter contains a description of the user stories created, which acted as a basis for the development of the song scanner in regards to the requirements it should fulfill. The interesting aspects of its implementation are also described.

The song scanner's main responsibilities are to find songs in a directory, look up their 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

The following are the user stores used for the Song Scanner.

Title:	Locate .mp3 files on device.
Priority:	High
Story:	As a user, I, want the application to search the directories I specify for .mp3 files.
Acceptance Criteria:	• When I start the application, it finds the .mp3 files stored in /sdcard/Music.

Title:	Obtain beats-per-minute for song.
Priority:	High
Story:	As a user, I, want the application to automatically obtain the beats-per-minute (BPM) of the .mp3 files located.
Acceptance Criteria:	• Obtain 113 as BPM for Rick Astley's "Never Gonna Give You Up", when locating the 12 .mp3 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` hold information regarding songs. `SongDatabase` is an abstraction to our sqlite database. `SongScanner` is responsible for keeping the database and MP3 file in selected folders synchronised.

6.2.1 Song

An instance of `Song` represents a MP3 file loaded into memory. The `Song` class consists of relevant data fields such as `file path` and `BPM`, as well as constructors for the database and song scanner.

Most information used to populate the fields of `Song` is contained within the MP3 file itself in the form of `ID3 Tags`, an informal standard is specified by Nilsson [10]. The constructor used by `SongScanner` takes a MP3 file and extracts data (title 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 beat per minute.

`DynamicQueue`, see section 5.3.1, relies on `SongDatabase` to handle queries for songs with a beat per minute in a specific range. In the event a row in the database does not have a corresponding MP3 file the row is deleted, this can happen if a user no longer wish 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 beat per minute online.

The first task involve a recursive search though selected folders for MP3 files, construct `Song` objects, then insert them into the database. After the `Song` object is constructed the `SongScanner` attempts to extract a cover image from the file and safes the cover in a separate file then adds the cover path to the object. This behaviour reflects the idea that the cover image where supposed to be downloaded online, but that feature failed to emerge as it is common to store album cover in the file.

Kristian: not quite simple design.

Kristian: todo: skriv noget om hvordan vi finder BPM for songs

Step Counter 7

In this chapter, the development of the step counter module is explained. The chapter separated into user stories which were the basis for developing the step counter, technology explorations made, implementation of the counter and lastly methodology observations made in regards to XP during the development of the step counter.

In our implementation of the step counter we decided to split the task into a data collection task, 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

The following are the user stories used as a requirement basis to develop the step counter

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:	• Criteria 1

- Precondition:
  - The application must be launched in the MusicPlayer Activity.
  - The user must have the device on his arm.
- Procedure:
  - Start running.
  - Increase running pace.
- Postcondition:
  - Verify pace is displayed on the screen and increases.

7.2 Technology Exploration

Ivan: Is it possible to get hardware access or is the listener the only way? Potentially no choice is taken.

Ivan: What kind of noise is removed?

Ivan: Is there any alternate way to get data from the chip? Is a decision even made?

Ivan: Was it a good decision, why?

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, see Android [2] for a 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 carrying the smartphone in a specific position (in hand, in pocket or strapped to 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.*
2. Jog 100 steps. *Jog: to run at a leisurely, slow pace.*
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.*
4. Sprint 100 steps. *Sprint: to run at full speed.*
5. Alternating. Walk 20 steps followed by jogging 20 steps, running 20 steps, sprinting 20 steps, and walking 20 steps.2
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 practice 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 in Zhao [12].

Algorithm Modification The algorithm in Zhao [12] is for a pedometer working directly with the ADXL345 chip. We, however, rely on a sensor listener on 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

7.3.1 Step Detection

To determine whether a step is taken, the approach taken by Zhao [12] 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 [12, 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 [12, 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 an 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 number of steps taken 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

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

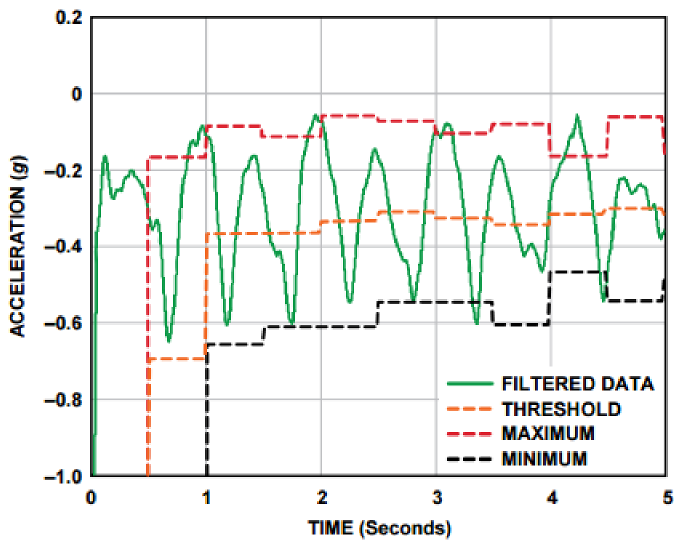

Figure 7.1: A acceleration plot from Zhao [12, p. 2] showing filtered data from a pedometer worn by a person walking

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.

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

In this chapter, the development, implementation and relevant observations in regard to XP made while developing the non-graphical user interface (NGUI), are described. The NGUI makes it possible to control the play, pause, stop, previous and next functionality of the music player by tapping anywhere on the screen.

The chapter contains a description of the user stories created, which acted as a basis for the development of the NGUI in regards to the requirements it should fulfill. Points of interest from its implementation are also described.

8.1 User Stories

The user stories on which the implementation of the non-graphical user interface was based on were

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:	• The application can still be controlled after the `Power` button is pressed.

- Precondition:
  - The application must be launched in the MusicPlayer Activity.
- Procedure:
  - Press `Power` button.
  - Verify screen turns off.
  - Perform single tap.
  - Verify application starts playing.
  - Perform double tap.
  - Verify playing changed to next song in queue.

    - Perform triple tap.
    - Verify playing song changed back to previous played song.

- Postcondition:
    - Verify all verifications have passed.

8.2 Implementation

8.2.1 Turning off the Screen

The implementation of when the screen is turned of, 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 by for example 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 8.2.2.

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

8.2.2 Handling of Taps

The detection of taps is done by listeners provided by the Android platform. Two types of taps are handled, others are ignored. The two types are: when a finger taps once and not a fling for example, and when a long tap is performed, for example holding one's finger on the screen for 200 ms. When a finger taps once, a method called `increment()` is called.

The 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 occur. Every time within 500 ms 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 appropriate action. The 500 ms were determined experimentally since we could not find any standard regarding wait times on NGUIs.

After an action has been performed, the global tap counter is reset and everything goes back to the state it was in at first. The application is ready for a new action.

The exception is the long press. When the long press is performed, the music is stopped immediately since there is no need to detect more than one successive long press.

Test 9

Testing is important in software development. It is critical since 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).

In this chapter the types of tests performed during the development application are described. The types of tests used were acceptance tests and unit tests. The advantages and disadvantages of unit testing, the group experienced during development are also discussed. Lastly the group's reflections about testing

9.1 Acceptance Test

As a way of making sure that a user story is fulfilled an acceptance test should be written. An acceptance test describes how the application should perform in a given situation for the 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 program 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 them automated is to make it possible to use them as regression tests, making sure that refactoring and/or changes to other parts of the code do not break the functionality of features that have already been implemented.

Dan: Should maybe have some sources?

The important thing when writing a test is to make sure it is completely unambiguous and reproducable. 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.

Dan: Should we write more about how we wrote this test?

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 Android [4] 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 practice 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

This section is where the reflections about the programmatic part of the test should be. Hence NO reflections about methodology here.

Acceptance Tests

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.

Christoffer: should the stuff below be moved to methodology reflection?

Testing also helped enforced other XP practices than refactoring. Namely collective ownership, pair programming, and continuous integration.

The XP practice 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 practice. 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 solution 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.

Methodology - Reflection 10

Ivan: Overvej hvor det er bedst at placere metodologi-observationer. Hvis vi bliver som vi er nu, så skriv observationer der passer til perioden og byg løbende en forståelse op, som bliver analyseret til sidst. Fortæl det gerne som "historier", så det er lidt spændende at læse.

This chapter contains the reflections and discussions about our use of XP in this project. The discussion is centered around the twelve main practices of XP, but with extra focus on three practices that were extra important for the project, namely pair programming, planning, and refactoring. These three practices are especially interesting because they required us to change our usual way of working considerably, and extra effort was put into following these particular practices. In Sections **??**, **??**, **??**, and **??** we presented some observations made during the project, and the following discussion is based on those observations.

Dan: Find the right methodology observations refs

10.1 Pair Programming

Initially, we used Teamviewer for pair programming instead of one shared physical monitor as XP prescribes. We found that this did not work as well as with a single monitor. The main problem was that it did not encourage frequent driver change. This meant that often one person coded and the other person was just looking at it. We experienced a learning curve adopting pair programming, and initially it was difficult for the observer to contribute with input and most work was done by the driver. The driver had to carry the work load.

Another issue was that if the task being worked on was not well understood by the pair, two people ended up not advancing towards a solution. We addressed the issue by temporarily splitting the pair to perform research and experimentation, which could help reach a solution.

We found that the forced timed pair programming switch did not sit right with us. Multiple times we experienced that we should switch just before the current issue was done. This created overhead and we sometimes forgot to change partners. We therefore decided to use a task-based approach where switches were only made between issues or between issues estimated to take more than 3 hours. We suspect that this may have hurt the collective ownership, but we found it more productive. We later abandoned the 3 hour switch.

We attempted to reduce the overhead by solving trivial issues alone rather than in pairs.

10.2 Planning

Planning poker was used for planning. We found planning poker made sure that everybody was heard in regards to planning. It also made sure that we had a common understanding of the task to be planned, e.g. if there was a big difference in the estimates, the group might not agree on what the task entails.

At first, we did planning but we did not prioritise the tasks. This was a mistake because the most important task was not always the first to be finished. We tried different ways of timing the solving of tasks. First, we just looked back in the calendar for the iteration and looked at which days we worked on what. Then we made a rough estimated of how long it took to solve the task.

We experienced that it was difficult to get meaningful timing data this way. Later, a tool was found to help us track time. Based on this, we could determine how good we were at estimating tasks in relation to how long it took to solve them. Additionally, we could keep track of how many development hours were used each iteration.

A disadvantage of using the tool was that since much weight was put on detailed timing, the tool itself generated overhead. We later moved on to just estimating units.

As a result of our estimate reviews, we found a recurring pattern in our estimates. Trivial tasks were generally easy to estimate accurately. Smaller, one unit, tasks were more likely to exceed their estimate. Lastly, the big tasks, four plus units, were likely to be over-estimated with the exception of a few which grew bigger. We suspect this was because we should have decomposed the tasks further.

Having big tasks with no specific goals, such as ongoing refactoring, did not work well, because they made us unproductive. Refactoring for example was later decomposed into concrete tasks which worked better.

10.3 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.

10.4 Other practices

10.4.1 40-hour Work Week

10.4.2 Coding Standard

In the first two thirds of the project our only coding standards were how variables were named, and where brackets should be. We decided to only add to the coding standard if we were in disagreement about how to write code. Later we found that this was insufficient. The conflict never arose, even though group members wrote code differently.

We ended up creating a more detailed and formal coding standard which can be seen in

Christoffer: Make reference to coding standard section

. Code written after the improvement of the original coding standard improved the quality of the code.

10.4.3 Collective Ownership

10.4.4 Continuous Integration

10.4.5 Metaphor

10.4.6 On-Site Customer

10.4.7 Simple Design

10.4.8 Small Releases

10.4.9 Testing

Conclusion 11

11.1 Discussion

Ivan: Vedr. GUI: Vi kan diskutere hvor godt det er i forhold til brugbar/ikke brugbar information og feedback. Specielt i forhold til det ikke grafiske ui kan vi overveje en mere klar form for feedback ved f.eks. nummerskift (vibration?).

Ivan: Hvorfor er der settings på main screen

11.2 Future Work

11.3 Conclusion

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.

Ivan: brug af reactoring?

Retrospective 1

12. feb - 20. feb

A.1.0.1 Coding Standard

We have tried to make the code readable by using field-standards

Ivan: def. 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.

Ivan: Nesting?, switches?, complexity?

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 practice, 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 practice 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.

  Ivan: Explain what Everhour is.

- 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.

  Ivan: Consequence of a review

What Did We Accomplish?

Ivan: Create some kind of reference/correlation between this and 40-hour work week.

- 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.

  Ivan: What is done? Scrum done? Completed? Started?

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.

Ivan: Illustrate this "de facto" standard, maybe by examples.

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.

"pace" might be ambiguous, as it can represent both velocity and SPM.

Ivan: def. SPM

Ivan: For whom will "pace" be ambiguous? Us - hence the Metaphor is an inward practice.
This however does not mean the user will see it as ambiguous.

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 Small Releases

We use it, so no comments..

Ivan: virker det?

A.1.0.13 On-site Customer

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

Ivan: surrocate costumer

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.

Christoffer: REMEMBER TO CHECK SOURCES

A.1.0.15 Refactoring

We found that our quality is not good enough, so we will in the future (against XP recommendations) have explicit tasks for refactoring.

Ivan: see fowler

Kristian: refactoring bad code is ok

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.

Ivan: One of the goals for ‘Code standard’ is to create an environment where the code is similar and easy to “learn”.

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.

Ivan: no tools?

Kristian: we got tools now

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 practice 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.

  Ivan: omsk. velocity begreb
- 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 være gennemtestet!

  Ivan: stort krav

- 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.

Ivan: ??

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:

```
if (statement) {
    //Do stuff
}
else {
    //Do stuff
}
```

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 practice 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.

Ivan: Good reflections and good section.

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.

Ivan: Title is a type of metaphor - reflect upon this. Maybe even change the title.

Ivan: Icons are types of metaphors - reflect upon this. Maybe even change the icons.

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

Ivan: Haven't we used a surrogate 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.

Ivan: Explain the difference between the issues we use and the use cases XP uses.

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

Ivan: Present arguments

Ivan: Define/Refer to definition of 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.

Ivan: Come up with better argument, since user is not always about cost/value.

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.

Ivan: Search for Laurie Williams sources.

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 - e.g. Kristian writes settings GUI, it is his GUI. 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.

Ivan: One of the goals for 'Code standard' is to create an environment where the code is similar and easy to "learn". It sound like we have trouble learning the code, suggesting we have very bad code standard. Explain.

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.

Ivan: It is really that time consuming to setup?

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?

Ivan: Elaborate on symptoms and diagnosis. What went wrong? Why did it go wrong?

- 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.

  Ivan: Ugly, but okay that we write it. Either describe symptoms and diagnosis or just say “Okay, we are busy, so in this version, we have this bug with a todo, although it is quick and dirty”.

- We discovered that we have not quite tested well enough.

  Ivan: Elaborate

- All in all we have been VERY VERY 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

Ivan: Evaluate how the fragmentation of timeslots have affected the project.

The 40-hour work week practice 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.

Ivan: What is the solution to counter this?

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.

Ivan: Define novelty!

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.

Ivan: It is sad that we have not gotten this to work.

Ivan: Would have been a good opportunity to use the surrogate costumer role to look at the program with a different set of eyes. This could lead to other ideas and innovation.

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.

Ivan: Maybe mention something about how, we have used this practice in a other way, did that work?

Ivan: Must a surrogate customer only be used in review and acceptance test?

Ivan: Can we integrate the surrogate customer role in the daily work? It would be difficult, since the idea is to forget oneself and be a customer instead. Discuss. One upside could be how we, while developing, always can be in the user/runners stead, since it is so close to us.

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.

Ivan: Why did we have motivational problems? Was it due to problems in our use of XP? Was it due to problems in our project?

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.

```
////////////////////////////////////////////////////////////
//                    Private Shared Resources             //
////////////////////////////////////////////////////////////
//region
  Code...
//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] Google Android. Fragments. http://developer.android.com/guide/components/fragments.html, 2015. [Online; Accessed 2015-05-18].

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

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

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

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

[6] 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.

[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] 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.].

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

[11] 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.

[12] 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].

Project CD B

The CD found on this page contains the following:

- The source code for
- A compiled version of
- A digital version of the report in PDF format.

Examples & ToDo C

Alexander: Example of comment/ToDo made by Alexander

Christoffer: Example of comment/ToDo made by Christoffer

Dan: Example of comment/ToDo made by Dan

Kristian: Example of comment/ToDo made by Kristian

Ivan: Example of comment/ToDo made by Ivan

```java
public class HelloWorld {

    public static void main(String[] args) {
        System.out.println("Hello, World");
    }

}
```

Listing C.1: Caption of code snippet

This is how you refer to a source written by Edworthy and Waring [7].

List of Todos

i, 🟥 **Ivan:** Metaphor - Think about more suitable title.

2, 🟩 **Alexander:** Read last problem and compare current version vs previous.

5, 🟩 **Alexander:** Spørgsmål til Ivan: Hvilken struktur er bedst, den lange eller den korte?

7, 🟥 **Ivan:** Hvis vi vil starte med et overblik, bør vi medtage en brugskontekst - hvad er det egentlig man kan med det? Det kunne også give mening at give et overblik over programmet.

9, 🟨 **Kristian:** should be expanded

9, 🟦 **Dan:** General stuff about the release here

9, 🟨 **Kristian:** consider if we should have use cases here.

9, 🟦 **Dan:** Should we mention this “clear goal” before? (We had a clear goal....)

10, 🟦 **Dan:** Find LABEL TO RELEVANT XP SECTION

10, 🟦 **Dan:** Beck’s book

10, 🟦 **Dan:** Did we explain planning poker?

10, 🟥 **Ivan:** Overvej hvor det er bedst at placere metodologi-observationer. Hvis vi bliver som vi er nu, så skriv observationer der passer til perioden og byg løbende en forståelse op, som bliver analyseret til sidst. Fortæl det gerne som “historier”, så det er lidt spændende at læse.

11, 🟨 **Kristian:** consider “familiar music player design”

12, 🟨 **Kristian:** this seems similar to *Affordance*

12, 🟦 **Dan:** General stuff about the release here

15, 🟦 **Dan:** Please rewrite this to make more sense.

15, 🟦 **Dan:** Fit this stuff in:

18, 🟨 **Kristian:** not quite simple design.

18, 🟨 **Kristian:** todo: skriv noget om hvordan vi finder BPM for songs

20, 🟥 **Ivan:** Is it possible to get hardware access or is the listener the only way? Potentially no choice is taken.

20, 🟥 **Ivan:** What kind of noise is removed?

20, 🟥 **Ivan:** Is there any alternate way to get data from the chip? Is a decision even made?

20, 🟥 **Ivan:** Was it a good decision, why?

25, 🟦 **Dan:** Should maybe have some sources?

26, 🟦 **Dan:** Should we write more about how we wrote this test?

27, 🟧 **Christoffer:** should the stuff below be moved to methodology reflection?

29, 🟥 **Ivan:** Overvej hvor det er bedst at placere metodologi-observationer. Hvis vi bliver som vi er nu, så skriv observationer der passer til perioden og byg løbende en forståelse op, som bliver analyseret til sidst. Fortæl det gerne som "historier", så det er lidt spændende at læse.

29, 🟦 **Dan:** Find the right methodology observations refs

31, 🟧 **Christoffer:** Make reference to coding standard section

33, 🟥 **Ivan:** Vedr. GUI: Vi kan diskutere hvor godt det er i forhold til brugbar/ikke brugbar information og feedback. Specielt i forhold til det ikke grafiske ui kan vi overveje en mere klar form for feedback ved f.eks. nummerskift (vibration?).

33, 🟥 **Ivan:** Hvorfor er der settings på main screen

37, 🟥 **Ivan:** brug af reactoring?

37, 🟥 **Ivan:** def. field-standards?

37, 🟥 **Ivan:** Nesting?, switches?, complexity?

40, 🟥 **Ivan:** Explain what Everhour is.

40, 🟥 **Ivan:** Consequence of a review

40, 🟥 **Ivan:** Create some kind of reference/correlation between this and 40-hour work week.

40, 🟥 **Ivan:** What is done? Scrum done? Completed? Started?

41, 🟥 **Ivan:** Illustrate this "de facto" standard, maybe by examples.

41, 🟥 **Ivan:** def. SPM

41, 🟥 **Ivan:** For whom will "pace" be ambiguous? Us - hence the Metaphor is an inward practice.
This however does not mean the user will see it as ambiguous.

41, 🟥 **Ivan:** virker det?

41, 🟥 **Ivan:** surrocate costumer

42, 🟧 **Christoffer:** REMEMBER TO CHECK SOURCES

42, 🟥 **Ivan:** see fowler

42, 🟨 **Kristian:** refactoring bad code is ok

43, 🟥 **Ivan:** One of the goals for 'Code standard' is to create an environment where the code is similar and easy to "learn".

43, 🟥 **Ivan:** no tools?

43, 🟨 **Kristian:** we got tools now

44, 🟥 **Ivan:** omsk. velocity begreb

45, 🟥 **Ivan:** stort krav

45, 🟥 **Ivan:** ??

49, 🟥 **Ivan:** Good reflections and good section.

50, 🟥 **Ivan:** Title is a type of metaphor - reflect upon this. Maybe even change the title.

50, 🟥 **Ivan:** Icons are types of metaphors - reflect upon this. Maybe even change the icons.

51, 🟥 **Ivan:** Haven't we used a surrogate customer?

52, 🟥 **Ivan:** Explain the difference between the issues we use and the use cases XP uses.

52, 🟥 **Ivan:** Present arguments

52, 🟥 **Ivan:** Define/Refer to definition of novelty.

52, 🟥 **Ivan:** Come up with better argument, since user is not always about cost/value.

53, 🟥 **Ivan:** Search for Laurie Williams sources.

53, 🟥 **Ivan:** One of the goals for 'Code standard' is to create an environment where the code is similar and easy to "learn". It sound like we have trouble learning the code, suggesting we have very bad code standard. Explain.

54, 🟥 **Ivan:** It is really that time consuming to setup?

55, 🟥 **Ivan:** Elaborate on symptoms and diagnosis. What went wrong? Why did it go wrong?

55, 🟥 **Ivan:** Ugly, but okay that we write it. Either describe symptoms and diagnosis or just say "Okay, we are busy, so in this version, we have this bug with a todo, although it is quick and dirty".

56, 🟥 **Ivan:** Elaborate

57, 🟥 **Ivan:** Evaluate how the fragmentation of timeslots have affected the project.

57, 🟥 **Ivan:** What is the solution to counter this?

57, 🟥 **Ivan:** Define novelty!

57, 🟥 **Ivan:** It is sad that we have not gotten this to work.

57, 🟥 **Ivan:** Would have been a good opportunity to use the surrogate costumer role to look at the program with a different set of eyes. This could lead to other ideas and innovation.

58, 🟥 **Ivan:** Maybe mention something about how, we have used this practice in a other way, did that work?

58, 🟥 **Ivan:** Must a surrogate customer only be used in review and acceptance test?

58, 🟥 **Ivan:** Can we integrate the surrogate customer role in the daily work? It would be difficult, since the idea is to forget oneself and be a customer instead. Discuss. One upside could be how we, while developing, always can be in the user/runners stead, since it is so close to us.

60, 🟥 **Ivan:** Why did we have motivational problems? Was it due to problems in our use of XP? Was it due to problems in our project?

67, 🟩 **Alexander:** Example of comment/ToDo made by Alexander

67, 🟧 **Christoffer:** Example of comment/ToDo made by Christoffer

67, 🟦 **Dan:** Example of comment/ToDo made by Dan

67, 🟨 **Kristian:** Example of comment/ToDo made by Kristian

67, 🟥 **Ivan:** Example of comment/ToDo made by Ivan