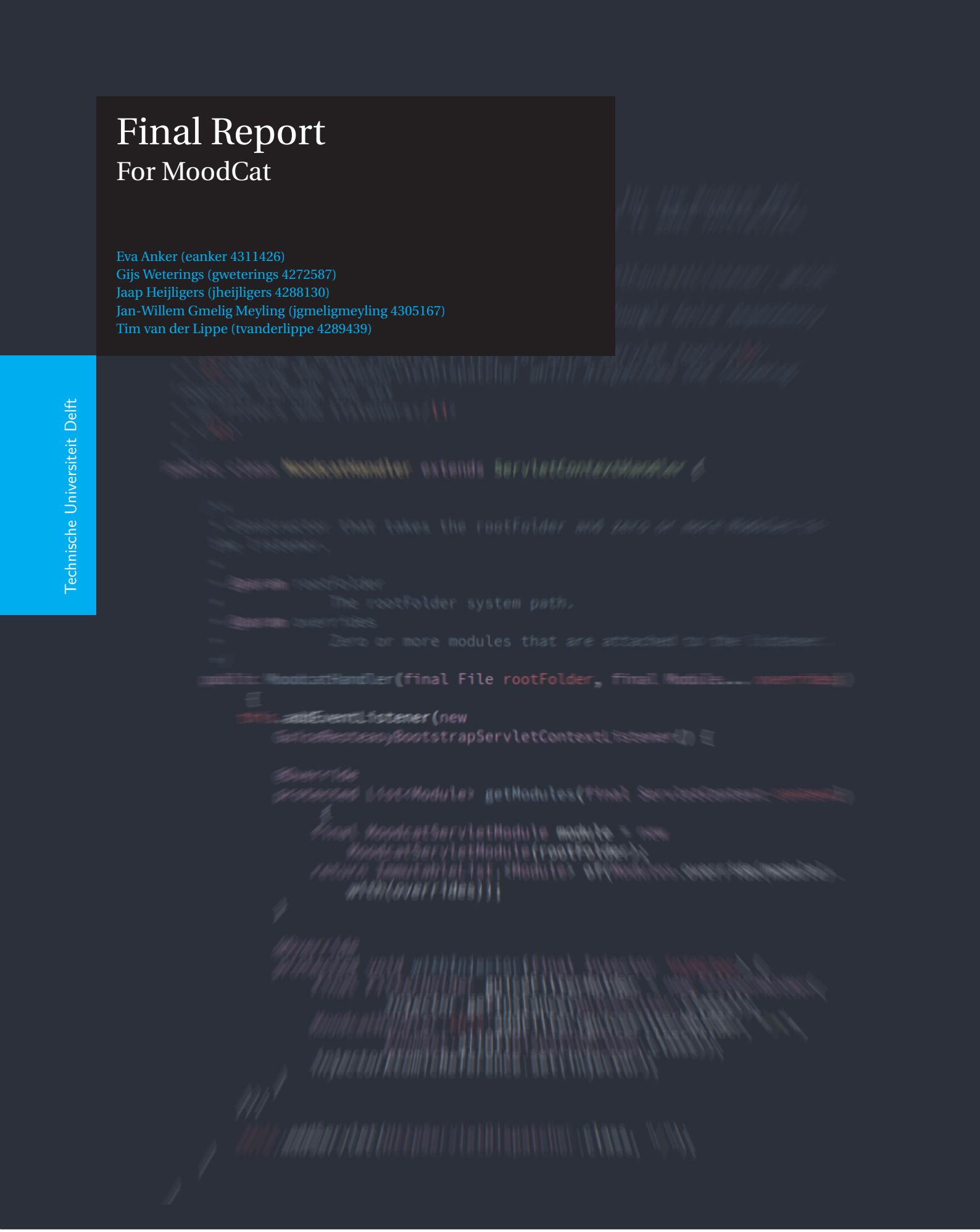
Final Report
For MoodCat

Eva Anker (eanker 4311426)
Gijs Weterings (gweterings 4272587)
Jaap Heijligers (jheijligers 4288130)
Jan-Willem Gmelig Meyling (jgmeligmeyling 4305167)
Tim van der Lippe (tvanderlippe 4289439)

Technische Universiteit Delft



TU Delft Delft University of Technology

Challenge the future

FINAL REPORT

FOR MOODCAT

by

Eva Anker (eanker 4311426)
Gijs Weterings (gweterings 4272587)
Jaap Heijligers (jheijligers 4288130)
Jan-Willem Gmelig Meyling (jgmeligmeyling 4305167)
Tim van der Lippe (tvanderlippe 4289439)

June 25, 2015

Supervisor:	Cynthia Liem	TU Delft
Teaching Assistants:	Friso Abcouwer,	TU Delft
	Abhishek Sen,	TU Delft

An electronic version of this document is available at https://github.com/MoodCat/MoodCat.me/.

TU Delft Delft University of Technology

Contents

1

INTRODUCTION

The context project is a project in which bachelor students at the TU Delft should develop a product in 10 weeks.

The task we were assigned to was to design the music services of tomorrow. Our group had to design an application that would allow users to listen to music of a certain mood and interact with people in the same mood.

The system should allow users select a mood and a room, where after they join a room according to the selected mood. When a user is logged in, he should be able to send messages to other users in the room. Additionally a user can classify a song according to mood of the song. Lastly there should be a ranking game where users can classify unranked songs. The user shall be rewarded with points for contributing to the system.

In this report we will explain MoodCat and how we have developed this system. First we will give an overview of the overall system and secondly reflect on the product and process. Thirdly we will explain the different functionalities of MoodCat. After that we will elaborate the interaction between the user and our system. Then we will evaluate the functionalities and provide a list of implemented features. Lastly we will give an outlook of possible improvements for the system and how they could be implemented.

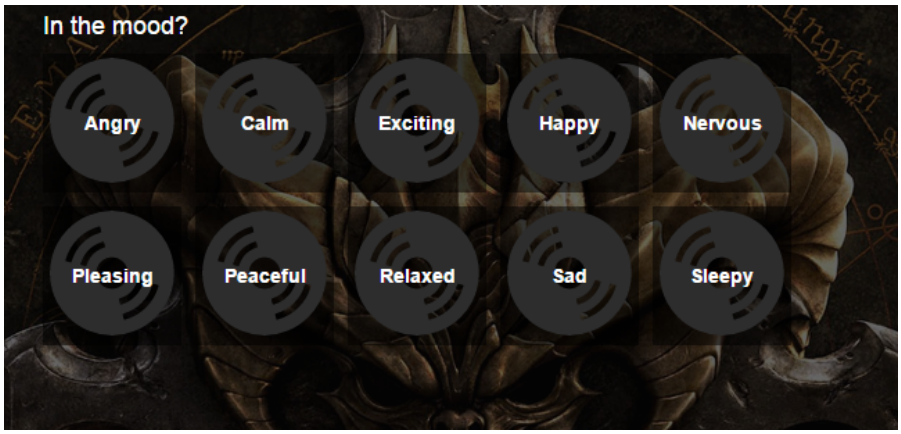


Figure 1.1: Mood selection page

2

Overview of the developed and implemented software product

MoodCat is a website where people can listen to music together. It combines enjoying your mood with social interaction. When entering MoodCat, the user can choose one or more moods from a given list. Based on the selected moods, a list of rooms sorted on relevance is shown, displaying the names and the song currently playing in the rooms. When entering a room the user can chat with other users that have joined the room.

2.1. Components

The software of MoodCat consists of two main components: the Java based backend and the frontend which is a website, hosted at `http://moodcat.me`.

2.1.1. Frontend

The frontend uses AngularJS for managing its objects and connection with the backend and bootstrap for the layout and design. AngularJS is an MVVM framework written in Javascript. It provides a mechanism to separate concerns between models, views and controllers. It also simplifies managing and updating displayed data through two-way data binding. AngularJS parses HTML elements that contain custom tag attributes. The attributes are interpreted by AngularJS and replaced with actual data from the model. The model is updated through constant communication with the backend. The current song and the time of the music is synced with the backends in order to make sure the users in a room listen to the same song at the same time.

2.1.2. Backend

The backend is responsible for managing the underlying structure of the project, to connect loosely coupled with the frontend and to persist and retrieve data from our database. The backend connects through Hibernate to a PostgreSQL database running on a server which is also hosted on the same server. The database contains the songs, artists, rooms and users. When the backend is started, it connects with the database and retrieves the current rooms. The REST API is then initialized, which can handle calls at `http://moodcat.me/api/`. The backend provides models, of which the fields are sent through the REST API as generated JSON. The models represent the inner content of the entities in the database.

2.2. MAPPING MOODS

In order to compare rooms, moods and songs we use valence arousal **vectors**. Every mood is mapped by a 2-dimensional vector of valence arousal, both of which have values between -1 and 1. Each room has its own **vector** which is used to determine which room to suggest based on the selection of moods. In order to queue up new songs for a room, the backend will fetch new songs that have vectors close to the room's vector. The database uses a R-tree spatial index to quickly calculate distances and order the songs accordingly. The indexing method is further described in the appendix A. The songs are mapped to vectors through a user feedback system. When a user likes a song that plays in the room, the song can be upvoted. The system then knows this song fits in the room. When a user downvotes a song, the song apparently does not fit the room. The user will now be asked to provide a classification for the song. The user can choose a degree of both valence and arousal and the valence arousal vector is slightly tweaked accordingly.

3

REFLECTION ON THE PRODUCT AND PROCESS FROM A SOFTWARE ENGINEERING PERSPECTIVE

In this chapter we discuss the software engineering perspectives of MoodCat. First we discuss our developing process. Secondly, the general structure and architecture of the product is explained. Then we will take a closer look at the used design patterns. After that we will elaborate on the choices to increase code readability and maintainability. Finally, the most important libraries and languages we used will be listed.

3.1. PRODUCT ARCHITECTURE

The architecture of MoodCat is very similar compared to existing websites, we use a client-server architecture. The frontend (client) is a AngularJS web application, whereas the backend (server) is a Java REST API based on JAX-RS[1]. The backend has a persistence layer that interactes with a PostgreSQL database.

3.2. DESIGN PATTERNS

Due to the usage of various libraries and frameworks, we use quite some design patterns. Additionally we did design our backend system very methodically, using the model-view-controller (MVC) pattern. The API classes define the view of the application (in the form of JSON responses) which is fetched by the frontend to be used by the client. The backend package contains the controllers of the system. The entities in the database package define our models.

In the frontend we use a slightly different structure, most commonly referred to MVVM (Model, View, ViewModel). The API calls are the models obtaining the data. The controllers and services are the viewmodels, which not only show but also alter the content. And lastly the HTML templates are the views showing the website to the user.

A few of the design patterns we used are: Facades, Proxies, Singletons, Factories, dependency injection, MVC, MVVM, and resource pooling .

3.3. CODE READABILITY / MAINTAINABILITY

In our efforts to create a codebase that not only fit our needs, but is readable and maintainable by others as well, we valued clean, short code greatly. Methods are kept short and simple, fill only one purpose, and have descriptive names.

We used the new technologies in Java 8 to it's full potential, substituting for loops and (nested) if statements with lambdas and anonymous classes wherever applicable.

In order to ease the communication with the database, we build the SQL commands using QueryDSL, which implements the builder pattern. Added benefit was the type checking capability of this pattern, de-

[1] The implementation we use is RestEASY, which can be found on `http://resteasy.jboss.org/`

creasing the risk of writing incorrect queries and making sure when making updates to entities, queries remain correct.

Google Guice allowed us to build our application like LEGO™: composition at its best. **Guice** is a Dependency injection framework which allows looser coupling between classes and packages. Responsibilities are split better and rapid prototyping is possible without having to worry how to pass dependencies around in the project.

Lastly the MVC-structure enabled us to change inner workings of the backend without breaking the API contract with the frontend and/or without breaking the database schema.

3.4. SCRUM

We worked with the **SCRUM-methodology**. Agile methodologies were known to most of us before this project, but no one had really used it consistently, aside from the basics of SCRUM during the SEM course. We did some research and found a really nice guide from Atlassian[2]. We learned about the different roles of SCRUM: the SCRUM Master, the Product Owner and the development team, as well as how to manage meetings in a proper manner. We decided to have daily SCRUM meetings at 09.00 o'clock every morning. During this meeting, we would use our Waffle[3] board to keep track of the groups progress. Each member of the group had to answer 4 questions:

1. What have you done yesterday?
2. What are you going to do today?
3. Are you on schedule?
4. Do you need help on anything?

This went surprisingly well! In our first sprint we had a little trouble guessing how long an issue would take. We discovered quickly in the following sprints that story points in Waffle helped us managing tickets and sprints. We decided that 1 story point equals 2 hours of work.

SCRUM meetings were useful to keep an overview of the system, as well as help others out when they were stuck. We did continue past 15 minutes a few times, but usually with a smaller group for a specific issue.

Overall, we are all convinced **SCRUM** is a great way to develop software, however we think that a sprint could better last two or three weeks instead of one. The sprint planning and sprint reflection took up a considerable amount of time in the sprint and the weekly demo forced us to put a difficult issue aside more than once.

3.5. PULL-BASED DEVELOPMENT AND CODE REVIEWS

We extensively used this methodology to control our Git usage. We used the following procedure:

- After we have discussed and defined a new feature, bug or enhancement, a ticked will be created and the assigned team member will write the code, possibly in pair with another team member (pair programming). This code will then be pushed to a branch on the remote repository.
- The team member opens a pull request (PR). This is a formal request to merge the branch into another different branch, usually the master branch.
- Other team members review the code in a static and a dynamic fashion. They look for errors, improvements or discussion points. Additionally, the Continuous Integration (CI) server builds the branch to make sure the code compiles and all tests pass.
- The assigned team member responds to the feedback. The team member should explain the reasoning behind the code contributions and process the feedback.
- The PR is accepted once all feedback has been processed and the code changes are approved.

[2] https://www.atlassian.com/agile
[3] http://waffle.io

3.5.1. REFLECTION ON THE PULL-BASED DEVELOPMENT

Overall we are pleased with how we were able to use pull requests. One very trivial advantage of reviewing changes to the code, is that you ensure that the code is readable. If another teammember is unable to read the code, the code has to be refactored. This way the system becomes more maintainable.

Another benefit came a bit more to a surprise to us. Throughout the project we used many techniques and libaries that were new to us (see 3.6). The pull-based development enabled us to discuss the usage of the framework. Therefore we really learned a lot from reading and reviewing each others code, prevented *wrong*[4] usage of the used frameworks, and ensured that we were still holding to our software architecture.

In the beginning we did not schedule time for the reviewing process of the pull-requests. The pull-requests piled up till we hastily reviewed and merged them at the end of the iteration. This caused unnecessary problems with merging, as there were quite some conflicting line changes. Therefore we decided to integrate the reviewing and merging more into the sprints, which worked out really well.

3.6. USE OF LIBARIES AND LANGUAGES

We have used quite a lot of languages and libraries. To get a clear view of what belongs where, we split the system in our three components.

3.6.1. FRONTEND

The frontend for Moodcat is the website loaded into the user's browser.

LANGUAGES

To create a website, we use the standard web languages: HTML for markup and Javascript for client-side logic, DOM manipulation and communication with the backend API, and CSS for styling.

LIBRARIES

We decided to use AngularJS to increase the productivity of developing features on the frontend. This is a framework built upon Javascript and it deals exceptionally well with DOM updates (with 2-way data binding) and HTTP calls. It also enables re-usable code with directives, a mechanism to create an on the fly webcomponent. We actually extend AngularJS further with packages to suit our specific needs.

In order to generate CSS, we use preprocessor called **SASS** that eases the re-use, inheritance and nesting of rules.

3.6.2. BACKEND

LANGUAGES

The backend is written in Java. We have chosen Java because everyone in the team is the most experienced in this language.

LIBRARIES

To turn the application into a servlet and process HTTP requests, we use Jetty. The RESTful API is implemented using RestEASY, a library that implements the interfaces defined by JAX-RS. JAX-RS has a dependency injection module to ease the process of defining end-points. However, we also wanted a more complete dependency injection framework. Therefore we decided to use **Guice** for constructor injection, together with **Guice-JPA** for transaction support and **Guice assisted-inject** for factories.

We're using Hibernate for ORM (Object-relational mapping). This allows us to transparently store Java objects in the database. Tables are generated according to the fields defined in the entities.

In order to quickly select songs or rooms from the database, based on their vector, we use a R-tree spatial index. In order to enable R-trees in our database and backend, we use Hibernate Spatial.

In order to ease the development process, we use the H2 database with the GeoDB spatial query extension for testing and development, so we do not need additional PostgreSQL instances. Besides that, we're using Project Lombok for automatic generation of general methods.

[4] With *wrong* usage of the framework we mean not adhering to the public API of the framework, introducing a libary very similar to an existing one, or writing something manually that is already part of the framework.

4

DESCRIPTION OF THE DEVELOPED FUNCTIONALITIES

In this section we will discuss the main functionalities of our system. MoodCat can be split up into three main features. These functionalities are the ability to play music, chatting in the rooms and classifying a song. They will be described in the sections below. But first we will give a small overview of the structure of the system.

4.1. INTRODUCTION TO MOODCAT

When a user enters the website he selects one or more moods (for example: happy, exciting or relaxed). Then a list of rooms fitting that selection of moods will appear. When clicking on a room a user joins, and he can listen and chat with other people in the room. A user can vote if the song is wrongly classified for a room. If this is the case, a small pop-up will appear and asks how the song should be classified instead. MoodCat also has a gamification aspect where a user is served with songs not yet classified. The user then gets a snippet of the song (30 seconds at most) and is then asked for a rating.

4.2. LISTENING TO MUSIC

MoodCat is developed as a music application and thus the ability to play music is very important. Although, our music player is a bit different from a normal music player. MoodCat works as a streaming service, and because of that the music is synchronized with the room the player is in. So you can only mute the sound, not alter its progression. When the internet connection of the user goes down for a few seconds or the connection is slow, the music synchronises with the room as soon as possible. In fact, a room in MoodCat can be compared to a radio station.

4.3. CHATTING

In Moodcat you can chat with other people in the room. In order to send a message the user needs to be logged in using SoundCloud. At SoundCloud you can register with one click using your existing Facebook or Google account.

When a user is logged in, he can chat by typing a message in the "Type a message" bar. The user sees the messages from other users, accompanied by their usernames. The name of a user is the name a user has set in SoundCloud as their real name.

4.4. CLASSIFYING

When a user disagrees with the classification of a song, the user can give his own classification. This is done with valence and arousal, which are terms to express the mood of a user. To classify a user gets different buttons with an image of state. These images are designed in such a way that it only can be seen as one state. When a user plays the classifying game the user classifies the snippets with the same buttons as the normal classification.

The classification effects how the song is classified as a mood. When a song is downvoted by a lot of people in the room, the song will not be playied in that room anymore. It can be scheduld in another room though, according to its ajusted classification. This way the system improves over time. When a user can earn points by classifying songs and is therefore encouraged to train the system. The scores can be found on the leaderboard. For classifying an unrated song a user gets more points than a normal classification. This is done to stimulate the user to classify more songs, and thus grow MoodCat's collection.

5

HUMAN-COMPUTER INTERACTION

5.1. INTRODUCTION

In order to evaluate the usability of the system, we have organised various testing sessions with potential users. In these testing sessions we used a think-aloud approach in order to gather user feedback. First we will elaborate on the claim analysis and what knowledge we expect the user to have. Secondly we will explain the setup of the testing sessions and what the outcomes were.

5.2. PRODUCT DESIGN AND CLAIM ANALYSIS

We have various claims on what we expect of the user. In this section we will elaborate the two most important claims: user navigation and social interaction.

5.2.1. USER NAVIGATION

The interface was designed to be minimalistic and intuitive. The user should be able to navigate throughout the system without external guidelines or instructions.

We do assume the user is able to understand the ways how to navigate from the mood selection to a room. We had several iterations of the state transitions to smooth out the user experience. After selecting the moods, rooms appeared on the screen. In the evalution some users did not fully understand this and we realized the transition was not clear enough. Therefore we changed the behaviour to hide the moods and only show the list of rooms after a mood was selected. This resulted in the users understanding the different steps in process of entering a room from a mood.

After changing this, we received additional feedback about the instructions. The instructions at the various steps were unclear and could be worded differently. We agreed that the instructions could be more fine-grained and decided to place little textboxes on the page. These textboxes guide the user, if they require more information. Therefore it does not bother users whom already know how to use the system, but will help beginners for which the instructions are aimed.

5.2.2. SOCIAL INTERACTION

Users meet in rooms, where music is playing. Additionally we created a chat message system that allows users to interact.

We claim that users enjoy or prefer to have social interaction with other users. For a part of the potential userbase, listening to music is a social activity. Good examples of these activities where this social element shines are festivals and pubs. In these places music is played and groups of people enjoy the music together. It is not necessary that the persons in the group know each other, as experiencing music is a universal activity.

Our chat system provides the above group the possibility to interact with each other, in order to express their feelings. However, the chat system does not bother the user group that does not require social interaction. This group usually consists of people who listen to music passively; when they are doing other activities while enjoying music. If the user uses MoodCat passively, there is the possibility to hide the browser from the desktop. Therefore the user does not view the page and does not see the chat system either.

We have asked our testers explicitly if they appreciate the placement chatbox. In general it did not bother them. The purpose of the system was clear and the usage was intuitive. So far we have not received a single negative response regarding it and therefore we assume the usage is easy and the results succesful. Additionally we saw various users immediately post messages to the rooms they were in, by checking our logs.

Concluding we think the general intention is clear and it is a successful addition to our product.

5.3. CONTEXT INQUIRY

Moodcat has been used by a group of testers. Additionally we supervised 6 testers who tested the system extensively. In this section we will explain what the outcome was of the 6 think-out-loud testing sessions.

The first impressions were positive. Everyone liked the user interface and the initial look-and-feel. After everyone examined the homepage, the next goal was to go in a room. Not all the testers were certain what to do at this point and required additional guiding. Therefore we decided to add extra text on the various pages to explain how to reach the next step. Secondly, the login system was not really noticeable at the beginning. We had to explicitly point out that users have the ability to log in to our system. In order to stimulate logging in and using the system by its full potential, we added a guiding arrow to point out the login button.

Besides the initial navigation problems, the testers did not have more trouble using and clicking through the system. The purpose of the chatbox was clear, even though we did not provide explicit explanations. Together with the song information, the general consensus was that the room was nicely and intuitively designed.

Two of the testers mentioned that it was not clear which moods they selected and which they did not. This was mostly related to the fact that the next button was placed as it were a mood. We changed the behaviour of the next button to only show when moods are selected. This animation implicitly guides users to click the next button when they are satisfied with their mood choices.

Most users also expected visual confirmation that the vote they casted had been received. Initially this was only a disabled button. We therefore added color changes based on the type of vote, to not only emphasize the succesful vote, but also to remind the user which vote they casted.

Apart from the above points, some small issues were discovered and have been fixed. They were mostly related to sizing issues on the various platforms we tested on.

Concluding the developed system is well-received. The only ambiguities that became apparent were related to navigation and guidance of the user. Additions of small texts and changes in transitions have solved these issues.

6

EVALUATION OF THE FUNCTIONAL MODULES AND PRODUCT

As described in section 2, Moodcat is roughly developed in four modules. For the backend we can distinguish the persistence layer, the backend layer and the API presentation layer. The frontend can be considered a module on itself. We will now describe how we reflect on the implementation of these modules.

6.1. SOFTWARE DEPENDENCIES

One of the more controversial choices in our project is undoubtedly the relatively heavy usage of software dependencies and generated code (see section 3). The downside of this approach is that we all had to learn and adapt to the various libraries and frameworks. On the other hand, however, the used libraries and frameworks really set the bar on code quality and kickstarted our project. We would definitely use JAX-RS, AngularJS and Hibernate again if we had to write a web system for future projects and we are glad that we took this project as the opportunity to work with these tools.

6.2. PROJECT PROBLEMS

If we would have to pinpoint two failures in the development process, we can list two design problems. One failure was that we did not know our ORM framework (Hibernate) well enough. This led to quite some bugs which involved low level debugging in the framework. Eventually we got it fixed but it took relatively much time. Most of the problems were caused because we wanted to synchronize chat messages in the database so that another server would be able to take over in case of a failure or when scaling out. But in the end we can debate if this was such a good idea to implement something so complex for something that is "nice to have".

For the web interface there is still some room for improvement. Most importantly the unclear positioning of various elements and the lack of descriptive headers made some parts of the website not so intuitive. The only real failure we encountered in the UI layer was that the two ways of listening to music (the classification game and the room listening) had quite some conflicting logic which had to be dealt with. Because we designed these views / features separately, we did not plan out how transitions between the two modes would be handled well enough. This caused some bugs that had to resolved later on and the implementation of these fixes can still be improved using, for example, a state pattern.

6.3. REFLECTION

Aside from these two failures, we are really happy overall about the product we have built in the past few weeks. First of all, the product we have built is in a deployable state and almost all feedback from our current user group was positive. Secondly, we have managed to realize almost all must have requirements, and missed just one should have requirement. This of course besides the neural network, described in appendix B, which we unfortunately had to drop in the development process. We however do think that the neural network that we designed would work in practice and that it would be a huge improvement both to the functioning and user experience of the system. We will further elaborate on this in the section Song and room suggestion.

7

OUTLOOK

During the project, we developed several improvements to the system. These improvements could not have been implemented due to time constraints, but the system designed in such a way that support for these features can be easily achieved.

7.1. SONG AND ROOM SUGGESTION

Our initial selling point was a neural-network based suggestion system (see also appendix B). A neural-network is a system that can classify items based on features and output in one (or few) dimensions. In order to train a neural-network, you also need pre-determined labels/expected values for the dimensions. In our case, the neural-network should output valence and arousal. However, existing datasets did not contain those values. How to obtain the valence and arousal of song-features is currently researched by various universities, but no method has been succesful yet. Therefore we were unable to train and use our network.

At the moment users can classify songs and provide the system the expected valence and arousal. The weights of the nodes in a network can be trained using the user-data gathered over time, which makes it possible to integrate a network in the future.

7.2. MESSAGE POLLING VERSUS BROADCASTED EVENTS

Currently all room interactions are based on a polling structure between the frontend and backend. The backend provides the song that is currently playing and the latest chat messages which the frontend uses to quickly check their status. This unnecessarily stresses the network connection, as most calls do not contain additional information the frontend did not have yet.

To reduce the network usage, the polling-structure can be replaced with socket events. Therefore only when there are updates, packets are sent between the frontend and backend. This also makes changes more instant as the clients will immediately get notified, instead of on regular intervals.

7.3. USER INTERFACE

There are various possible improvements to the user interface.

1. At the moment of writing, the room selection page contains a list of rooms. This list of rooms is sorted by relevance, but there is no indication on how relevant a certain room is.

   Our clients indicated a more suitable representation would be to display the select rooms in a 2D space. This space would implicitly represent the inner vector structure used in the backend. To indicate the selected moods and the corresponding vector, a point is marked in the space. Using this represenation, relevant rooms are placed more closely to the point, whereas less relevant rooms are placed further away. The room selection page can be easily changed to suit this new representation.

2. The up-vote and down-vote buttons are at the moment placed in the top bar. The initial philosophy was that while a song is playing, the user should be able to express his/her feelings. Since the currently played song is in the top bar, the natural decision was to place the ote buttons next to it.

   After gathering user feedback, one of the outcomes of the tests was that the purposes of the buttons was unknown. Some prototypes can be developed with different placements of buttons which can be tested on a set of users. Then the users can vote on the most intuitive solution.

A

R-Trees as a database level Nearest Neighbours Algorithm

As elaborated in Referencesmapping-moods, our approach to find songs and rooms with similar moods, we find the valence and arousal values for songs and rooms, and map these as coordinates on a plane. Finding these songs should be autonomous, accurate, performant and scale to large datasets.

In our development we initially opted to achieve this using a K-Nearest Neighbour (**KNN**) algorithm. Such an algorithm finds the K-nearest elements from a given point. Consider the room to be a point in 2D space (x-axis is the valence value, the y-axis is the arousal value). For example, see figure A.1. The red dot is the room we are seeking a song for, the green and blue dots are songs defined with a valence arousal vector. The purple circle describes a **K-NN** with $K = 3$, while the blue circle describes a KNN with $K = 7$.

Our initial, naive implementation slowed down drastically with a high amount of songs and rooms; we had to think of a different solution. Initially, we created an Approximate Nearest Neighbours (**ANN**) implementation using a KD-tree. This remedies the slow runtime by calculating an approximate subset of points close to the room. On this subset our original NN algorithm could run. Having solved the runtime complexity issues, we ran into memory issues instead. Indexing all songs in a KD-tree still requires every song object to be loaded into memory. We needed to find a sure-fire, effecient way of finding appropriate songs for a room.

We found the answer in the most unexpected part of the application: The database. Specifically an R-tree index (proposed by Guttman in 1984). They provide reasonably fast and effecient storage and retrieval, and have the option to to a cheap **K-NN** like selection of the closest elements to a given point. A good example of an application that uses this is OpenStreetMap[1]. They use Postgres spatial indexing (using R-trees) on all

[1] http://openstreetmap.org

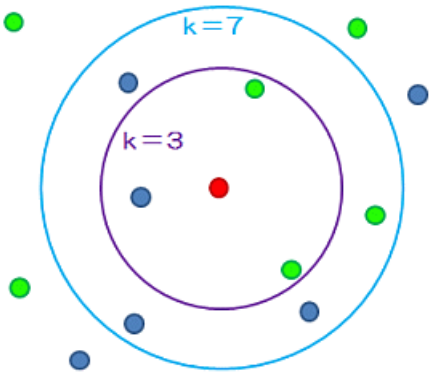


Figure A.1: K-Nearest Neighbour results

elements on their map. With this, they decide what they should show at the current zoom level, as well as use it for route planning.

With R-trees we found the solution for our problem to find songs efficiently. Every room has a vector, and every song has a vector. Both those vectors are indexed with R-trees. To find fitting rooms to a given set of moods, we construct a vector from those moods and find the closest rooms in order of closeness. In a room, we need to find fitting songs.

The reason R-trees are suitable for us to fix our scaling problems, is that the database does all the work. This has the added benefit of the database being able to load only the tree to find the correct nodes, not the entire dataset. This way we keep what we do not need in memory on disk, and retrieve what we do need as fast as possible.

B

NEURAL NETWORKS

One of the bigger issues we had to deal with in MoodCat was to classify the mood of a song. The initial plan was to create a neural network that could do this.

Some of the MoodCat team members have followed a course Computational Intelligence, where they learned different learning algorithms to solve different kinds of problems. One widely used technique to classify objects by using the objects' features as inputs for the neural network.

B.1. STRUCTURE OF THE NEURAL NETWORK

Neural networks consist of an input layer, an output layer and a number of hidden layers. Each layer consists of a number of nodes (See also figure B.1). The nodes in the input layer are fed with *features*. For the classification of songs these would be all sorts of information of the song that can be represented as some form of numeric metrics, for example: the amount of beats per minute (**BPM**), the average **loudness** or **tonality** of the song. The key of a song is not an obvious metric as it cannot be represented on a linear scale: music in A has not more in common with *B* than *C* or even *G*, while on a linear scale *A* would deviantly have more in common with *B* than *G*. The network could however map the input for each key: if a song is written in *A*, the *A* input is 1 and the other key inputs are 0.

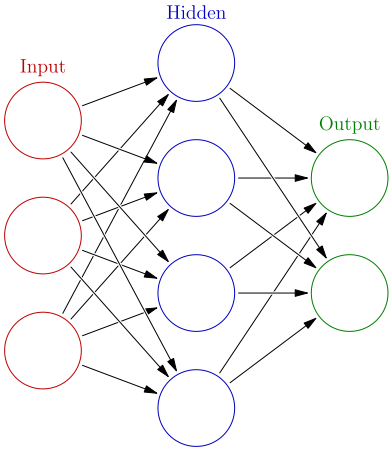
The nodes in the hidden layers have nodes with weights for each node in the next layer, and the output layers would have a node for each mood, the output nodes output a number representing how likely the song is to correspond with that mood.

The weights in the network are initialized randomly and trained by feeding training data to the network. The training data consists of a list of songs along with their respective moods. When the classification is correct, the network does not do anything. However if the classification is incorrect, the weights are updated based on error propagation. This way, the neural network *learns* how features relate, how they affect moods, and eventually how to classify songs to moods.

B.2. DATASET FOR THE NEURAL NETWORK

Sadly there was no dataset that contained either the song or features of songs and their corresponding moods. Therefore we were unable to train the network with the provided data. If we would have been able to train the network, we could have avoided the cold start problem the system currently suffers. To compensate for this problem, we have designed the rating game. The rating game let our users classify the songs. In order to stimulate this behaviour, we let them award them with points.

Figure B.1: Neural network



GLOSSARY

AngularJS AngularJS is an open-source web application framework maintained by Google and by a community of individual developers and corporations to address many of the challenges encountered in developing single-page applications. It aims to simplify both the development and the testing of such applications by providing a framework for client-side model–view–controller (MVC) and model-view-viewmodel (MVVM) architectures, along with components commonly used in rich Internet applications.. 2, 4, 6, 11
arousal How calm or excited you are. 3, 12

CSS Cascading Style Sheets is a style sheet language used for describing the look and formatting of a document written in a markup language. 6

data binding Data binding is the process that establishes a connection between the application UI (User Interface) and business logic.. 2, 6
dependency injection Dependency injection is a software design pattern that implements inversion of control. The responsibility for locating or constructing dependencies is expressly separated from code that would take responsibility for using those dependencies.. 4–6

GeoDB GeoDB is a spatial extension of H2, the Java SQL database.. 6

H2 H2 is a relational database management system written in Java. It can be embedded in Java applications or run in the client-server mode.. 6
Hibernate Hibernate ORM (Hibernate in short) is an object-relational mapping framework for the Java language, providing a framework for mapping an object-oriented domain model to a traditional relational database.. 2, 6, 11
Hibernate Spatial Hibernate Spatial is a generic extension to Hibernate for handling geographic data.. 6
HTML HyperText Markup Language is the standard markup language used to create web pages. 2, 4, 6
HTTP The Hypertext Transfer Protocol is an application protocol for distributed, collaborative, hypermedia information systems. 6

Javascript A dynamic programming language. It is most commonly used as part of web browsers, whose implementations allow client-side scripts to interact with the user, control the browser, communicate asynchronously, and alter the document content that is displayed. 6
JAX-RS JAX-RS: Java API for RESTful Web Services (JAX-RS) is a Java programming language API that provides support in creating web services according to the Representational State Transfer (REST) architectural pattern.. 4, 6, 11
JSON JavaScript Object Notation, is an open standard format that uses human-readable text to transmit data objects consisting of attribute–value pairs. It is used primarily to transmit data between a server and web application, as an alternative to XML. 2, 4

KD-tree Data structure used for spatial querying, optimized by a recursive structure using buckets defined by bounding boxes. 14

MVC Model–view–controller (MVC) is a software architectural pattern for implementing user interfaces.. 4
MVVM Model View ViewModel (MVVM) is an architectural pattern for software development.. 2, 4

ORM Object-relational mapping (ORM, O/RM, and O/R mapping) in computer science is a programming technique for converting data between incompatible type systems in object-oriented programming languages.. 6, 11

polling Polling, or polled operation, in computer science, refers to actively sampling the status of an external device by a client program as a synchronous activity.. 12
PostgreSQL PostgreSQL, often simply Postgres, is an object-relational database management system (ORDBMS) with an emphasis on extensibility and on standards-compliance.. 2, 4, 6

R-tree Data structure explained in appendix A. 3, 6, 14, 15
REST Representational State Transfer (REST) is a software architecture style consisting of guidelines and best practices for creating scalable web services. REST was introduced by Fielding. 2, 4

valence The amount of attraction you have towards a certain event. 3, 12

BIBLIOGRAPHY

[1] A. Guttman, *R-trees: A dynamic index structure for spatial searching,* in *INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA* (ACM, 1984) pp. 47–57.

[2] R. T. Fielding, *REST: Architectural Styles and the Design of Network-based Software Architectures*, Doctoral dissertation, University of California, Irvine (2000).