Adaptive Music Recommendation System

by

Fan Yang

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Masters of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2010

© Fan Yang, MMX. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part.
Adaptive Music Recommendation System

by

Fan Yang

Submitted to the Department of Electrical Engineering and Computer Science on May 21, 2010, in partial fulfillment of the requirements for the degree of Masters of Engineering in Computer Science and Engineering

Abstract

While sources of digital music are getting more abundant and music players are becoming increasingly feature-rich, we still struggle to find new music that we may like. This thesis explores the design and implementation of the MusicPlanner — a music recommendation application that utilizes a goal-oriented framework to recommend and play music.

Goal-oriented programming approaches problems by modeling them using Goals, Techniques, and a Planner. The Goals are representations of a user’s intent, while the Techniques are the methods that can be used to satisfy the Goals. The Planner connects the Goals and Techniques in a user-defined way to find solutions to user’s requests. In the MusicPlanner, the Planner defines the top level Goal of recommending music, which can be satisfied by a set of recommendation Techniques. Each of the recommendation Techniques then declares the sub-Goal of playing music, which can be satisfied by a set of play Techniques. The Planner evaluates each of the Techniques and iterates through the results to choose the best set of Techniques to satisfy the top-level goal of music recommendation.

The MusicPlanner allows the user to create personal music stations and for each station, constructs a model of user’s music taste based on queries and feedback to the songs played. The extensible design of the architecture and the ease of implementing the MusicPlanner show how goal-oriented framework can simplify the work for programmers. In evaluating the performance of the MusicPlanner, we demonstrate that the Planner in the goal-oriented framework outperforms each individual recommendation Technique.

Thesis Supervisor: Steve Ward
Title: Professor
Acknowledgments

First and foremost, I would like to thank my advisor Professor Steve Ward and Justin Mazzola Paluska, both of whom had helped me tremendously throughout the project. My weekly meetings with them provided invaluable feedback, and helped me brainstorm for new ideas to explore.

I would like to thank my friends who have participated in my user study and helped me proofread this thesis. I would also like to thank my family for supporting me throughout my academic career.
# Contents

1 Introduction 13  
1.1 The MusicPlanner ............................................. 14  
1.2 Thesis Outline .............................................. 14  

2 Related Work 17  
2.1 JustPlay ....................................................... 17  
2.2 Last.fm ......................................................... 17  
2.3 Pandora ........................................................ 19  

3 Goal-oriented Framework 21  
3.1 Introduction to Goal-oriented programming .................. 21  
3.2 Goals .......................................................... 22  
3.3 Techniques .................................................... 23  
3.4 Planner ......................................................... 23  
3.5 Advantages of using Goal-oriented programming .......... 24  

4 Architectural Design 27  
4.1 Overall Architecture ........................................... 27  
4.2 User Experience ................................................ 28  
4.3 Server Module .................................................. 29  
4.4 Planner Module ................................................ 30  
4.4.1 Goals ........................................................ 31  
4.4.2 Techniques .................................................. 32
List of Figures

3-1 Architecture of the Planner ................................................. 22

4-1 MusicPlanner architecture .................................................. 28
4-2 Design of the Server module ............................................... 29
4-3 Diagram of the relationship between the Goals and Techniques of the MusicPlanner ................................................. 31
4-4 Example graph of similar songs with similarity scores and user ratings 35
4-5 Example graph of relevant tags with relevant scores and user ratings . 36
4-6 Example Goal tree for the MusicPlanner ................................. 38

6-1 User study results .......................................................... 55
List of Tables

5.1 Database tables storing static music data . . . . . . . . . . . . . . . 49
5.2 Database tables storing dynamic user data . . . . . . . . . . . . . . 50
Chapter 1

Introduction

With the rapidly growing availability of digital music, organizing and finding music can be a difficult and tedious task [1]. An average computer user has hundreds or thousands of songs on his computer. While the music players are becoming increasingly feature-rich, the process of getting to the music that the user wants to listen to is still inefficient. This inefficiency stems from two tasks: accessing music the user already likes and finding new music that he would like.

The task of accessing music involves the user opening a music player and finding the song or the set of songs that he wants to listen to at that moment. The ability to access music efficiently requires the user to maintain an organized music library. To organize music, the user has to catalog his music by either grouping songs into folders or by making playlists in a music player. This task is repeated every time the user obtains new songs, which can be frustrating for the user. Furthermore, the user may want to play different types of songs during different activities or at different times in the day. The variation in his music preferences introduces a new challenge for organizing music. On a more fundamental level, the problem lies in the concept of using playlists to keep track of the songs that the user likes. Playlists are static lists of songs; however, a user’s music preferences are constantly changing. As a result, using static playlists requires a user to manually update his playlists as his music preferences change. Therefore, to overcome the challenge of music organization with users’ evolving music preferences, we need an adaptive solution.
There are many web applications that assist users in discovering new music that the user may like. Web applications such as Pandora, Last.fm, and Yahoo Music recommend new songs to a user based on the songs that the user already likes. However, in order to receive recommendations that match the user’s taste in music, the user must spend a considerable amount of time and effort to indicate his music preferences by creating playlists, searching for songs that he likes, and rating songs that he listens to. The inefficiency lies in the fact that the online music recommendation engines are completely separate from the music on the user’s computer and separate from each other. These gaps result in separate music service entities that are not utilizing music preference data pertaining to each user, leaving users with the annoying task of configuration. What is needed, therefore, is a recommendation system that can model the users’ music preferences with minimal user configuration efforts and can extract recommendation data from other sources.

1.1 The MusicPlanner

To reduce the amount of user effort required to access desired music, we developed the MusicPlanner — an adaptive, plug-in based system using goal-oriented programming to utilize multiple music data sources and dynamically recommend songs based on user input. The goal-oriented approach allows the MusicPlanner to utilize new technology and resources as new ways of recommending music and playing music appear. By using simple user feedback to create and update a model of user’s music taste, the MusicPlanner addresses the problem of user’s changing music preferences while minimizing the amount of user configuration.

1.2 Thesis Outline

Chapter 2 discusses related work and lays out the motivation for this project. It also highlights some of the concepts that we adopted into the MusicPlanner. Chapter 3 describes the goal-oriented framework and the use of this framework in the Mu-
sicPlanner. Chapter 4 describes the architectural design of the MusicPlanner and discusses the design decisions made. Chapter 5 describes the implementation details of the MusicPlanner. Chapter 6 discusses the evaluation of the MusicPlanner, including the type of statistics that we collected, baselines for comparison, and an analysis of the results. Chapter 7 concludes the thesis with a discussion of future work and a summary of the project.
Chapter 2

Related Work

There are three important pieces of related work that motivated and provided ideas for the design and implementation of the MusicPlanner.

2.1 JustPlay

As one of the projects that use the goal-oriented framework, the JustPlay system is one of the more important examples of related work [2]. The main function of the JustPlay system is to reduce the amount of users' configuration in using their audio and video devices. It uses a voice recognition system to allow the user to give voice commands to the system and uses the goal-oriented framework to dynamically pick the configuration that the user would most likely choose. The idea of allowing the users to indicate what they want to do, while the goal-oriented framework determines the best ways to satisfy users' requests, is particularly relevant to the what we want to accomplish with the MusicPlanner. Therefore, we adopted the goal-oriented model in designing the overall architecture of the MusicPlanner.

2.2 Last.fm

Last.fm [3] uses a music recommender system called "Audioscrobbler" to build a detailed profile of each user's musical taste [4, 5]. The detailed profile of a user's
musical taste is constructed by recording details of the songs the user listens to, either from Internet radio stations, or the user’s computer or portable music devices. The website provides a user profile that displays user’s information and user’s favorite music as well as social networking and recommendation features. Last.fm allows users to tag and label artists, albums, and tracks, resulting in a site-wide folksonomy of music [6]. The tagging can be by genre, mood, artist characteristic, or any other form of user-defined classification. Users can browse via tags and play music that has been tagged a certain way. Users can also create custom radio stations and playlists from any of the audio tracks in Last.fm’s music library.

While tagging has proven to be effective and has been used by many users, the primary way in which Last.fm recommends music is by using a collaborative filtering algorithm. The concept of collaborative filtering algorithm is to look for groups of users who share the same music tastes and recommend to users music liked by other users in their group. Last.fm presents the users with a list of songs and artists that are not listed on users’ profiles but appear on profile of users who share similar music taste. There is also a recommendation radio station which will play music specifically filtered based on the user’s last week of listening.

There are two ideas that we can borrow from the Last.fm system. One of the ideas is collaborative filtering. Collaborative filtering algorithms have been designed specifically for data sets where there are many more users than items, such as a data set of a million users and ten thousand songs [7, 8]. While collaborative filtering is not appropriate for the MusicPlanner, since we have significantly more songs than users, it highlights the importance of looking at the data set in constructing the music recommendation methods. With the data set for the MusicPlanner being mostly songs and artists, we must take a music-centric approach rather than a user-centric approach in building recommendation methods.

This leads to the other important idea, which is the concept of allowing users to browse music and label music by tags. Tagging has proven to be an effective way of classifying music as tagging have now become one of the primary ways in which users browse music on Last.fm. Since the tag information for all the songs on Last.fm
is publicly accessible, we can use the music tag data in creating recommendation methods. As we will describe later in the thesis, music tags becomes the basis of one of the methods for music recommendation in the MusicPlanner.

2.3 Pandora

Pandora [9] is an automated music recommendation service that utilizes the Music Genome Project [10]. Pandora is built around the concept of using music stations to model user's musical taste. A music station can be created by specifying an artist or song, or a combinations of multiple artists or songs. Pandora responds by playing songs that are musically similar to the ones that the users provided. Pandora makes successive recommendations based on users’ feedback on approval or disapproval of individual played songs. Pandora uses over 400 different musical attributes to recommend music. These 400 attributes are also combined into larger groups called focus traits, such as rhythm syncopation, key tonality, and vocal harmonies. As of March 2010, Pandora had 700,000 tracks in its library [11].

One of the key ideas that we adopted from Pandora is the concept of modeling the user’s musical taste using music stations. This design allows the users to build multiple music stations, each with a distinct “personality”. Furthermore, it greatly simplifies the user interface design for the MusicPlanner, allowing us to focus more on designing and implementing the back-end architecture.
Chapter 3

Goal-oriented Framework

As we have described in Section 2.1, the JustPlay system demonstrates the effectiveness of the goal-oriented framework and motivates the usage of goal-oriented framework in the MusicPlanner. The goal-oriented design turns out to not only reduce the development time, but it also allows us to have an extensible architecture. This chapter explores the goal-oriented programming paradigm and gives a high-level description of how it is used in the MusicPlanner.

3.1 Introduction to Goal-oriented programming

Goal-oriented programming paradigm approaches problems by modeling the problem using three building blocks: Planner, Goals, and Techniques [12]. Figure 3-1 describes the relationship among the elements of goal-oriented programming and the layers of abstraction in the problem model.

Goal-oriented programming consists of two primary layers of abstraction: Task/Intent layer and the Component layer. The Task/Intent layer consists of Goals and Techniques. The Component layer contains resources that the Techniques and the Goals can use. The abstraction between the Task/Intent layer and the Component layer allows both layers to work independently and be replaced as new technology becomes available. In the context of the MusicPlanner, the Task/Intent layer represents the logic behind recommending a song or finding a music source to play a song, and the
components layer represents the various Application Programming Interfaces (API), the MusicPlanner database, and the external music data sources, such as YouTube and music libraries. Adhering to the goal-oriented design pattern leads to an organized and extensible architecture.

3.2 Goals

A Goal is the goal-oriented representation of a user’s intent and an abstraction of specifications. In the MusicPlanner, the Recommend Goal is a request for recommending songs on a music station, and the Play Goal is a request for playing a specified song. On a lower level, a Goal describes the formal parameters of the user intent and the properties any Technique that satisfies the Goal must provide. The Goals are written in the form of a generic procedure call that describes the parameters, properties, and behaviors of the user intent. The parameters are the customization of the Goals, and the properties are what the Planner uses to choose among the Techniques. In the MusicPlanner, the parameter of the Recommend Goal, the music station id, indicates the music station that the recommendation Techniques should use for recommending a song. The parameters of the Play Goal, which consists of the song title and the artist name, provide the play Techniques with the necessary information for playing
the song. These parameters describe user’s music requests and aid the Techniques in satisfying their Goals.

### 3.3 Techniques

Techniques are the competing methods that can be used to satisfy a Goal. Each Technique can satisfy a certain class of Goals. In the MusicPlanner, the recommendation Techniques satisfy the Recommend Goal while the play Techniques satisfy the Play Goal. Techniques are the mixture of declarative statements and arbitrary code that satisfy Goals. They encapsulate existing code modules and accesses to resources, making them easily usable by the Planner. In the MusicPlanner, the recommendation Techniques contain logic for recommending music, and the play Techniques encapsulate logic for finding a source to play music.

The Techniques are executed in three stages: sub-Goal declaration, evaluation, and commit. In the sub-Goal declaration stage, each Technique declares relevant sub-Goals, which must be satisfied before the Technique can proceed. In the evaluation stage, the Technique determines how well an instance of that Technique satisfies its Goal instance. The level of satisfaction is expressed numerically as the satisfaction score. The Planner then chooses the Technique with the highest satisfaction score to satisfy the Goal. In the commit stage, the system configures and updates the various components of the system. The committed Techniques form the actual implementation of the Goal.

### 3.4 Planner

The Planner is the engine that connects the Goals, Techniques, and Resources. The Planner picks the “best” Technique for each Goal based on how well the Technique satisfies the Goal (based on the satisfaction score). All the possible ways to connect Goals, Techniques, and Resources can be illustrated by a Goal tree, which is a decision tree whose nodes consist of Goals and Techniques. Each path represents a particular
Plan, a way in which the top-level Goal can be satisfied. In the MusicPlanner, a Plan is the set of recommendation Technique and play Technique that best satisfies a user's music request. The Planner is responsible for evaluating each of the Plans in the Goal tree to determine the best Plan to use in satisfying the Goal. Once a Plan is found, the Planner commits the Plan. After the Plan is committed, an external application can use the Planner to examine the Goal Tree. In the MusicPlanner, the controller component that handles the user requests uses the Planner to explore the committed Plan.

3.5 Advantages of using Goal-oriented programming

There are many advantages of using the goal-oriented framework in designing the MusicPlanner. On a high level, the goal-oriented programming paradigm leads to a clean abstraction for the music recommendation logic. The Planner can be seen as the kernel component of the MusicPlanner architecture, encapsulating the core logic of handling user’s requests. We describe the details of how the Planner connects to all the other back-end modules in the next chapter.

In the MusicPlanner, the Techniques are used to model different ways of generating music recommendations and searching for different sources to play the music, such as the online music APIs and the music library on users’ computers. The underlying goal-oriented framework in the MusicPlanner enforces a common interface that allows the Techniques to be compared in terms of their fitness to satisfy the Goal. More specifically, each Technique has an evaluation function that computes the satisfaction score of the results returned by the Technique. With this interface, the MusicPlanner simply iterates through the Techniques and chooses the result with the highest satisfaction score. We hypothesize that the “competition” between various Techniques can enhance the quality of results. Furthermore, using Techniques allows more recommendation Techniques or play Techniques to be added to the Mu-
sicPlanner without modifying other parts of the system. The extensibility allows the MusicPlanner to be augmented with more Techniques as more music sources become available in the future.
Chapter 4

Architectural Design

In this chapter, we describe the overall architectural design of the MusicPlanner with the details of each of the components as well as the rationale behind the design decisions. In designing the MusicPlanner, we followed the principle of maintaining a clean separation between logic and presentation [13, 14]. This design principle simplifies the development of the MusicPlanner and improves its maintainability, as the front-end of the application can be modified and new user interface feature can be added with little or no changes in the back-end.

4.1 Overall Architecture

We introduce the overall architecture of the MusicPlanner by describing, step-by-step, the process in which a user request is handled in the MusicPlanner.

Figure 4-1 illustrates the architecture of the MusicPlanner. The architecture is shown as a set of modules and components with functional dependency and hierarchical relationships. In a way, the architecture diagram illustrates the path of information flow through the MusicPlanner. The Graphical User Interface (GUI) captures the user input and sends the input data to the server module. The Server module then translates the user input into requests and calls the relevant functions in the Planner module. Finally, the Planner module handles the requests by calling functions in the Database module and Resource module and provides the Server mod-
ule with the response object to be sent back to the GUI. The design of each module and component is described in the following sections.

4.2 User Experience

The front-end interaction revolves around the concept of music stations, which we have adopted from Pandora, as we have described in Section 2.3. Each music station can be seen as a radio station that plays songs of a certain type. The user may train the music station by providing feedback to songs. The purpose of having music stations and allowing users to create multiple music stations, each with a different music taste or "personality", is to capture the users' multifaceted music taste and organize them. For example, the user may create a music station with upbeat songs for listening in the morning and another music station with classical music for listening at night.

To enable new users to use the MusicPlanner with limited features prior to registering for an account, two types of views are created: a public view and a private view. The public view is displayed when the user has not been authenticated. In the public view, the users can only listen to the public stations, and the user preferences are not saved. The public stations are genre-specific stations that recommend new
songs based on the ratings of a predefined set of songs. Once the user is authenticated, the private view is loaded. In the private view, not only can the user see user-specific information, but he may also add new stations or train existing stations. The user may also add his music libraries, which allows the MusicPlanner to play music from his computer.

### 4.3 Server Module

The purpose of having the Server module is to create an abstraction for the logic of handling user requests from the front-end and sending the response back to the front-end. Since one of the responsibilities of the Server module is to provide content for the front-end, the front-end and the Server module must agree on a specific format for the data being transferred, which inevitably creates a dependency between the front-end and the Server module. To transfer data in the mutually understandable format while maintaining a clear separation between back-end logic and the front-end presentation layer, we introduce two components in the Server module: the View component and the Controller component.

The View component acts as a bridge that connects the front-end to the Server module, communicating with the front-end using a predefined data format and delegating the tasks of generating results for user requests to the Controller component. On the input side of the View component, the View component parses the user requests and calls relevant functions in the Controller component to handle the requests.
On the output side, the View component formats the results from the Controller component before sending them back to the front-end. To maintain a clear separation between the presentation layer and the logic layer, we designed the MusicPlanner in a way such that the View component is the only component in the MusicPlanner code that communicates with the front-end. We further decouple the presentation from logic by allowing the View component to only contain logic for formatting request and response and not any back-end logic for handling requests.

On the other hand, the Controller component acts as a bridge that connects the View component to the rest of the system by retrieving requests from the View component, generating results for the requests, and returning the results to the View component. On the input side of the Controller component, the Controller component contains logic for handling the function calls from the View component by calling functions in the Planner module as well as some function calls to the Database module to retrieve user specific information. On the output side of the interface, the Controller component returns the results of the requests back to the View component.

### 4.4 Planner Module

The Planner module is the Goal-oriented framework for music recommendation. It is the core of the MusicPlanner application, interacting directly with the Server module, Resource module, and the Database module. The Controller component in the Server module interprets the user's music requests and calls the relevant functions in the Planner module to respond to the requests. The Planner module then recommends music by executing database queries and finds sources to play music by calling external API's. The Planner module consists of Goals, Techniques, and a Planner.

Figure 4-3 shows the relationship between the Goals and the Techniques of the MusicPlanner. The subsections below will describe in detail each of the Goals and Techniques and how they relate to each other.
4.4.1 Goals

The Goals in the MusicPlanner are the user’s specification on the music he wants to listen to. There are two types of Goals in the system: Recommend Goal and Play Goal.

Recommend Goal

The Recommend Goal uses recommendation Techniques to find a song that fits the user’s music tastes, and then uses the Play Goal to find the most suitable source to play the music. The Recommend Goal uses the information on the music stations - ratings on the previous songs played and the ratings on the tags related to the previous songs played. The Recommend Goal does not require any song specific parameters other than the music station id and the user id. Once the Recommend Goal determines the best song to recommend via its recommendation Techniques, it declares Play Goal as a sub-Goal in order to find a source to play the recommended song.
Play Goal

The Play Goal takes the exact artist name and song title as input and then finds the most suitable source to play the music. In the case of a Play Goal originated from a user request, the user query often does not have both the exact song title and artist name. Therefore, the Planner must first use the Last.fm’s API to retrieve the best match for song title and artist name before they can be used as a parameter for the Play Goal. It is possible to rely on the Techniques to determine the exact song title and artist name, but since all the operations within Planner requires this information, it is more efficient to process the user query and extract this information prior to the Planner execution. In the case of a Play Goal being instantiated as a sub-Goal to Recommend Goal, the extraction of song and artist information is unnecessary because the top level Goal will have this information.

4.4.2 Techniques

There are two types of Techniques in the system: recommendation Techniques and play Techniques. This section describes these two Techniques in detail.

Recommendation Techniques

The recommendation Techniques use the data collected on the music station, ratings of songs, and indirect ratings of tags, to determine the song that would best fit the user’s taste. There are three types of recommendation Techniques: Similarity Technique, Tags Technique, and Random Technique. The Similarity Technique determines the song to recommend based on the similarity of the recommended song to the songs that have been rated highly. The Tags Technique determines the song to recommend based on the amount of overlap in the set of tags of the recommended song and the set of tags that have been rated highly. The Random Technique recommends a song at random, and it is used mostly as a baseline for comparison in evaluating the performance of the different Techniques.
Play Techniques

The play Techniques search through the various sources of music to find the best one to play a requested song. Each Technique searches for results from a specific source. There are two types of music sources that the MusicPlanner uses to play a requested song: YouTube and users’ personal libraries.

We chose YouTube API [20] as the online music API to use because YouTube is currently one of the top music video providers and it has a very well-established API that allows developers to access their database of music videos. The YouTube Technique utilizes the YouTube API to search for the most relevant music video given the artist name and song title as query parameters.

The decision to use user libraries comes from the fact that most of the music on user’s computer are in MP3 format with playback quality that is often superior to the ones we can stream from the online sources. Furthermore, the music videos found on YouTube are sometimes created by amateur artists while most of the music on user’s computer are music files from the original albums. The installation of a user library creates an interface that allows the MusicPlanner to find and play requested music on user’s computer. This is achieved by running a Python script that creates a music file server and sends a file that contains the descriptions of a user’s music files to the MusicPlanner. The User Library Technique searches user’s music by simply searching in the file that contains the metadata for the user’s music files while the Python installation script for the user library does the work of gathering the metadata.

4.4.3 Technique Scoring Heuristics

The performance of the Planner in recommending music relies heavily on the heuristics for scoring the results from the Techniques. The performance is correlated with the ability of the Planner to rank the recommendation results in a way that aligns with users’ subjective preferences. Since the level of alignment of the recommendation results with users’ music taste dictates, in a large part, whether the user will like the recommended song, we must carefully construct a model upon which the heuristic is
built. The model is constructed based on the history of user feedback on the songs in each user station. This section describes how the model of user's music taste is created from user feedback and explains the heuristics used in scoring each type of Techniques.

Similarity Technique Heuristic

Building a model of users' music taste for the Similarity Technique involves saving the rating of songs and the indirect rating of songs similar to the ones played. The indirect rating of a song is determined based on its similarity score to the song that was directly rated and user’s feedback. As mentioned in the previous chapter, the user feedback is binary, either positive or negative. A positive feedback corresponds to a rating of 1, and a negative feedback corresponds to a rating of -1. Each unplayed song starts with a default rating of 0, and each new user feedback adds the rating for the feedback to the current rating. Thus, if we let \( n \) be the number of rated songs for the current music station, we can express the indirect ratings of a song by the following formula:

\[
\text{Score}_{\text{unplayed song}} = \sum_{i=1}^{n} \text{rating}_{\text{played song}_i} \times \text{similarity score}_{\text{played song}_i \text{ to unplayed song}}
\]

The purpose of saving indirect ratings of the songs is to expand the network of the user’s ratings on songs with the assumption that if the user likes a song, he would also like the songs that are similar to that song. Having an expanded network of user’s ratings on songs allows the MusicPlanner to have more data points to use in recommending new songs to the user. The recommendation heuristic is to find a song that has the most and strongest similarity links to the songs that are rated highly. Let \( n \) be the number of rated songs for the current music station, then the heuristic for scoring results from the Similarity Technique can be expressed as the following:

\[
\text{Score}_{\text{candidate song}} = \sum_{i=1}^{n} \text{rating}_{\text{played song}_i} \times \text{similarity score}_{\text{played song}_i \text{ to candidate song}}
\]
Figure 4-4 shows an example network graph of songs similar to the candidate song “Time of Your Life” by Green Day. The similarity scores are indicated by the number next to the line that connects the similar songs to the candidate song. In this example, all of the similar songs have been played on the current station, so each of the similar songs also has a user rating. The score of the candidate song is the sum of the products of the similarity scores and the user ratings for all the similar songs, which in this case is 2.53.

**Tags Technique Heuristic**

Similarly, the rating for a tag is based on the user’s feedback on a song and the relevance score of the tag with respect to the song. Let $n$ be the number of rated songs for the current music station, then the rating for each tag can be expressed by the following equality:

$$\text{Score}_{\text{tag}} = \sum_{i=1}^{n} \text{rating}_{\text{rated song}_i} \times \text{relevance score}_{\text{rated song}_i \text{ to tag}}$$

The Tags Technique heuristic is to find a song with the most and strongest relevance links to the tags that are highly rated. Similar to the heuristic for scoring the Similarity Technique, we let $n$ be the number of rated songs for the current music station, and then we can express the heuristic for scoring the results from the Tags

35
Figure 4-5: Example graph of relevant tags with relevant scores and user ratings

Technique as the following:

$$\text{Score}_{\text{candidate song}} = \sum_{i=1}^{n} \text{rating}_{\text{rated tag}_i} \times \text{relevance score}_{\text{rated tag}_i \text{ to candidate song}}$$

Figure 4-5 shows an example network graph of tags relevant to the candidate song “Time of Your Life” by Green Day. The relevance scores are indicated by the number next to the line that connects the tags to the candidate song. In this example, all of the tags relevant to this song have been rated on the current station, so each of the tags also has a user rating. The score of the candidate song is the sum of the products of the relevance scores and the user ratings for all the tags, which in this case is 2.89.

Normalization of Recommendation Technique Scores

Because the mean tag score and the similar song scores are different, each of these scores are normalized against the mean of all the scores of its type. For example, the normalized similarity score can be expressed by the following equation.

$$\text{score}_{\text{normalized}} = \frac{\text{score}_{\text{unnormalized}}}{\sum \text{score}_{\text{unnormalized}}}$$

Even after the first normalization, there is a marked disparity between the normalized tag scores and normalized similarity scores: the normalized tag scores are almost
always higher than normalized similarity scores. This difference in mean normalized scores is due to the fact that the number of tags used in calculating tags recommendation Technique score is significantly lower than the number of similar songs used in calculating the similarity Technique score. To account for this difference in mean scores, the normalized scores are normalized for a second time: the scores from the first normalization for each Technique are normalized against the average of the recommendation scores for that Technique during all the previous executions of the Planner.

Play Techniques

Since it is difficult to score the quality of each individual YouTube music videos, the scoring for Play Techniques relies on the assumption that songs found using the User Library Technique is of a better quality than songs found using the YouTube Technique. Because of the lack of API functions for determining the quality of the music in YouTube videos, we simply set the default score for each song found in User Library to 1.0 and YouTube video to 0.5. If the song is not found, the default score is 0.

4.4.4 Planner

In the MusicPlanner, the Planner component serves as a logic engine that handles the user’s music requests by using Goals and Techniques. In handling each request, the Planner is first instantiated with a set of Goals and Techniques, and then it iterates through the Techniques to find the result with the highest score. For example, in handling a recommendation request, the Planner is instantiated with the Recommend Goal and a set of recommendation Techniques: Similarity Technique, Tags Technique, and Random Technique. The Planner then chooses the song with the highest score among the results returned by the three recommendation Techniques.
4.4.5 Example Goal Tree

Each execution of the Planner can be illustrated by a Goal tree, a decision tree that describes the different decisions made by the Planner in satisfying a Goal. Furthermore, the Goal tree indicates the Plan, or the best path through the Goal tree, chosen by the Planner.

Figure 4-6 shows the Goal tree of an example execution of the Planner in recommending music. In this example, the Planner is handling a recommend music request for user music station with id 1. First, the Planner is instantiated with a set of available Techniques and the Recommend Goal with the station id as the parameter. The Planner then iterates through the recommendation Techniques to find a Technique that returns the best recommendation result. Since the Similarity Technique yields the best score, the Planner then declares the Play Goal as a sub-Goal, with the result of the Similarity Technique, “Blind” by Lifehouse, as a parameter. To satisfy a Play Goal with the specified song title and artist name, the Planner then iterates through the play Techniques to determine the best music source for playing this song. Finally, it finds that the User Library 1 Technique satisfies the Play Goal with the highest score. In the above diagram, the elements in gray outline the Plan that the Planner chooses for satisfying this recommendation request.
4.5 Database Module

The Database module is responsible for storing data and providing an API that higher level classes can use to access the data. Because the data used in the MusicPlanner is largely relational, we use a relational database as the primary method for data storage. In designing the way in which we store the data, we separate the data by its usage pattern. The music data is mostly static as we retrieve it from Last.fm, while the user data is dynamic, changing constantly to reflect users' preferences. This section describes the Database module as these two component datasets: music data and user data.

4.5.1 Music Data

The music data in the MusicPlanner consists of data for songs, artists, tags, similarity scores for pairs of songs, and relevance scores for tags. While it is possible to retrieve information about specific songs and tags via the Last.fm API [21] on runtime, storing the information in a local database allows us to construct a wide variety of database queries and database joins with other tables containing user preference information, similarity scores for songs, and relevance score for tags, all of which are not possible by using only the Last.fm API. Once the music data is collected from Last.fm, each existing row of the music data table remains unmodified. Since the music data table is not comprehensive, as only a portion of the music data on Last.fm is stored in the MusicPlanner's database, it is possible that the Planner may need to retrieve songs from Last.fm during runtime. For example, when the user requests a song that's not already stored in the database, the Planner uses the Last.fm API to acquire the music data and inserts new rows of music data into the database before continuing. Hence, a majority of database accesses to the music data tables are reads with occasional inserts.
4.5.2 User Data

The user data in the MusicPlanner consists of data for user authentication, user music stations, and user music libraries. The user authentication data is used to authenticate the users during the login in and sign up process. The user authentication data consists of unique user id's, usernames, and passwords of the users. The user id's are used to connect all the data in music stations and music libraries that are related to the users. The user music station data stores the list of songs that have been played, the average scores for songs, and the average scores for the relevant tags of the rated songs. The data for user station is what enables the recommendation Techniques to recommend songs for the user based on his music preferences. The user music library data stores the information regarding the music libraries, such as their names and locations and the usernames of the users who created the libraries. The data for user music libraries allows the User Library Technique to play music directly from the users' computer.

4.6 Resource Module

The Resource module consists of music API's and user libraries. The music API's used are Last.fm API and YouTube API. The Last.fm API provides a way to disambiguate the user queries. When there are multiple results for a single user query, the Last.fm API returns the most relevant result with the exact song title and artist name. The Youbute API provides access to all the music videos on YouTube, one of the primary sources of music for the MusicPlanner. The exact song title and artist name returned by Last.fm API is used to query the YouTube API, which then returns the most relevant music video to the MusicPlanner.

As described in the previous section in this chapter, the local user's music library acts as another source of music. The user may add multiple user libraries on different computers, and the Techniques in the Planner iterates through all the user libraries to find the desired music. The user library feature allows the MusicPlanner to utilize the high quality music files on the user computers.
The abstraction of Goals, Techniques, and Resources is especially useful here in providing modularity and extensibility. With this abstraction, the Resource module encapsulates the logic for connecting to all the different music data sources, such as APIs and music libraries. As new technology and new music data sources become available, the new resources can be independently integrated into the system as new objects of Resource module, and Techniques can be added to utilize these new resources.
Chapter 5

Implementation

We implemented the MusicPlanner according to the Goal-oriented design described in Chapter 3. Our implementation is written in Python 2.6.4 [15] with the Django web development framework [16]. We chose Python because the backend Planner framework is written in Python. However, none of the MusicPlanner code relies on any features unique to the Python language. We chose to use Django web development framework to implement the front-end more quickly and cleanly. There are two main goals of this implementation. The first is to demonstrate that the design outlined in the previous chapter is feasible. The second is to show that using the Planner as a backend logic component simplifies the implementation.

5.1 Front-end

The language used to build the front-end consists of HTML, Javascript, and CSS. The front-end design focuses on simplicity: it revolves around a main page where users can build music stations, search for music, and use the Planner to recommend new music. The user actions are communicated to the server-end via Asynchronous Javascript and XML (AJAX) requests [17]. This design is chosen so that requests happen asynchronously and only the necessary parts of the web page are updated for each response. Due to the fact that this web application is very request heavy, the AJAX design is more suitable than the traditional POST requests where the user's
In terms of interacting with the music Planner to get music results, there are two types of user inputs: searching for songs directly and giving feedback to songs. The user searches for the song by typing the name of the artist and/or the title of the song into the search box. While the MusicPlanner immediately displays the most relevant result, the user may view other relevant results. This feature addresses the issue of disambiguation when incomplete query information is given, such as when only the artist name or the song title is given. For user feedback, we use a binary rating system with a positive or negative rating, which is presented to the user as a thumbs up or thumbs down button. Other types of user inputs, such as log in, log out, and sign up, are handled by the server module.

5.2 Server Module

The Server module contains a View component and a Controller component. The View component is the only component in the back-end modules that communicates directly with the front-end. It consists of a set of functions that take request objects in the form of Python dictionaries, call the relevant functions in the controller to handle the requests, and finally return the result to the front-end. As mentioned in the previous section, all the requests from the front-end are AJAX requests. In the case of AJAX requests, the front-end sends the data to the server via POST request using AJAX. The responses for these requests are returned to the front-end in the Javascript Object Notation (JSON) format, which can be easily converted to a Javascript object on the front-end [18]. A few requests that are not front-end related are simple POST requests that trigger a database transaction, such as an insert, update, or select. These requests serve as an API to the MusicPlanner for applications that run on client computers. For example, user music library application uses this API to send the MusicPlanner server a data file containing user library music information and updates the server database with its up-to-date Internet Protocol (IP) address and port number. The design of the user library will be discussed in detail in later sections.
of this chapter.

5.3 Planner Module

The Planner module is implemented using the planner4 framework, which is a re-implementation of the O2S planner. More specifically, the classes in the Planner Module sub-class the classes in planner4 framework: Goal classes sub-class planner4.goal.Goal, and Technique classes sub-classes planner4.technique.Technique.

5.3.1 Goals

The main purpose of the Goals is to specify the set of required parameters. We aim to minimize the number of required parameters while still have enough to enable the Techniques to satisfy the Goals.

Recommend Goal

The Recommend Goal requires only the id of the music station on which the user requested music recommendation. The Recommend Goal provides the recommendation Techniques with the id of the music station so that the recommendation Techniques can use the data for the specified music station to recommend music. The reason that the Recommend Goal does not require user id is that each music station corresponds to a unique user id, which can be easily retrieved via a database call. In fact, the database call to retrieve the user id is obviated by the database call that retrieves information for a certain music station. One benefit of not requiring the user id for the Recommend Goal is that this Goal can be used with or without user authentication. The purpose of minimizing the number of required parameters is to allow more flexibility in way that the Recommend Goal can be used. For instance, the public music stations do not have corresponding user id’s and allow the users to listen without authentication. Therefore, the public music station cannot provide a user id to the Recommend Goal. If the Recommend Goal had required the user id, it would not be able to recommend music for the public music stations.
**Play Goal**

The Play Goal requires the name of the artist, the title of the song, and the id of the music station on which the user requested the music. The artist name and the song title are the basic music information that the play Techniques use to search for the song. The id of the music station is used by the Play Techniques to update the history of songs played on the music station.

### 5.3.2 Techniques

As we have described in Section 3.3, Techniques are the methods that the Planner can use to Satisfy the Goals. In this section, we describe the implementation of the Techniques in the MusicPlanner.

Interestingly, during the development of the MusicPlanner, we found that the Techniques can be implemented incrementally. When a Technique class is created, we simply add the Technique to the Technique database, or the collection of Techniques available to the Planner without modifying existing code. This demonstrates the ease of adding new Techniques to the MusicPlanner.

**Recommendation Techniques**

The Similarity Technique and the Tags Technique use a database query to compute the scores for each song in the database, according to the scoring formula described in Section 4.4.3, and returns the song with the best score.

Since the results from all of the recommendation Techniques must be comparable by the Planner, each of the Techniques must return results in the same format. To satisfy this requirement, the results are stored as dictionaries in Python with required parameters. Both Similarity Technique and Tags Technique must return result dictionaries that contain the id of the song in database, the title of the song, the name of the artist, and the normalized score of the song. The normalized scores for the song are used by the Planner to determine which recommendation Technique should be used to satisfy the top level Goal while the other parameters in the result dictionaries
are used mostly for displaying the result in the front end.

Play Techniques

The YouTube Technique utilizes the YouTube API to search for the most relevant music video given the artist name and song title as query parameters. More specifically, the YouTube API returns video feed data from which the video url of the most relevant video feed is extracted; the video url is then returned as the result of the YouTube Technique.

The User Library Technique searches music recursively in a specific directory on the user’s computer. As described briefly in Section 4.4.2, the user libraries are installed using a python script. There are many steps in installing the user libraries. Once the user runs the Python installation script, the script immediately creates three background threads. One of the threads launches a server instance that makes the user library accessible by the MusicPlanner. The second thread sends, in fixed time intervals, the IP address and PORT number of the python server to the MusicPlanner via a POST request. The third thread recursively traverses through the user library directory and collects the the ID3 data and relative path of all the songs in the user library, writes the information in a text file, and sends it to the MusicPlanner server via a POST request. On the server end, the IP address and PORT number is stored in the database. The text file that contains the ID3 information is stored in a folder with user library id of the user library database table as its unique file name. This design allows the MusicPlanner to be able to search through a user library by simply parsing the text file stored locally on the server and to delegate the tasks of recursively traversing through the user library directory and parsing ID3 tags to scripts running on client computers. Once the User Library Technique finds a match in the user library text file for the requested song, it returns a url of the song, which is constructed by concatenating the IP address and PORT number of the user library and the relative file path to the song.

The results from the YouTube Technique and the User Library Technique are scored according to the scoring formula described in Section 4.4.3. Similar to the
interface for recommendation Technique results, the play Techniques must return results in the same format. Again, this is implemented as dictionaries with required parameters. The Play Techniques must have result dictionaries that contain the url to the music location and the score for that result as well as other information for the song.

5.4 Database

The data of the MusicPlanner is stored in a MySQL database. This section discusses how the music data is collected and how the music data and user data are stored in the MusicPlanner.

5.4.1 Data Collection

The music database is built with the data collected from Last.fm. The data collection is done by using scripts that sends request to the Last.fm API, parses the XML response data, and stores it in the database. The script uses a random seed song and calls the similarity API in Last.fm to find information for all the songs similar to the random seed. The information for these similar songs is then saved in the music database and used as seeds for subsequent API calls. By calling similarity API recursively, we are able to collect information for 600,000 songs and 61,000 artists.

During the repeated calls to the similarity API in acquiring music data, the similarity information between songs is also stored in the database. The similarity information allows us to construct Techniques that recommend songs based on the degree of similarity between songs. Furthermore, the information about the tags for each song is collected by calling the tags API in Last.fm. The tags information can be used to build models of users’ music taste, which can be utilized in recommending songs. A total of 133000 tags were collected. The disparity between the number of songs and number of tags is due to the fact that many songs shared the same tags. For example, the “rock” tag is shared by all the songs of that genre.
Table 5.1: Database tables storing static music data

<table>
<thead>
<tr>
<th>Database Tables</th>
<th>Database Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>Songs</td>
<td>id, name, song image, artist id</td>
</tr>
<tr>
<td>Artists</td>
<td>id, name</td>
</tr>
<tr>
<td>Tags</td>
<td>id, name, song id</td>
</tr>
<tr>
<td>Song Similarity Scores</td>
<td>song id 1, song id 2, score</td>
</tr>
<tr>
<td>Tag Relevance Scores</td>
<td>tag id, song id, score</td>
</tr>
</tbody>
</table>

5.4.2 Database Tables

The schema design of each database table adheres to the Boyce-Codd normal form (BCNF) [19]. The database normalization ensures that the data are free from undesirable characteristics such as insertion, update, and deletion anomalies, which could lead to a loss of data integrity.

Music Data

Table 5.1 shows the schema design for the database tables that store the music data. Each table contains a primary key “id” that can be used to uniquely identify each row in that table. The primary key can also be used in other tables as a foreign key to connect related data. For example, the “id” in the Artists table is represented as “artist id” in the Songs table to indicate the artist for the song. The use of foreign keys such as “song id” and “tag id” imposes some restrictions on the order in which we collect and store the data. For instance, because the Song Similarity Scores table contains a pair of “song id’s” in each row, we must collect the similarity scores of the songs after we finish inserting songs into the Songs table. The Songs table and the Artists table provide the basic information for the music used in the MusicPlanner, while the Song Similarity Scores table and the Tag Relevance Scores table contain the data used by the Similarity Technique and the Tags Technique, respectively.

User Data

Table 5.2 shows the schema design for the database tables that store the user related data. The Users table contains the user authentication information. The “user id”
Table 5.2: Database tables storing dynamic user data

<table>
<thead>
<tr>
<th>Database Tables</th>
<th>Database Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>Users</td>
<td>id, username, password</td>
</tr>
<tr>
<td>Techniques</td>
<td>id, name</td>
</tr>
<tr>
<td>Music libraries</td>
<td>id, name, ip address, user id</td>
</tr>
<tr>
<td>Music stations</td>
<td>id, name, user id</td>
</tr>
<tr>
<td>Music station song ratings</td>
<td>station id, song id, rating</td>
</tr>
<tr>
<td>Music station tag ratings</td>
<td>station id, tag id, rating</td>
</tr>
<tr>
<td>Music station Technique scores</td>
<td>station id, Technique id, score</td>
</tr>
</tbody>
</table>

is used as the unique identifier to indicate the creator of each music station and music library. The *Techniques* table contains the list of Techniques that are currently available to use by the MusicPlanner. We made the implementation decision to store the Techniques in a database in order to maintain consistency with the other relational tables that uses id’s as identifier and to avoid having to statically define the Techniques in the code. The *Music libraries* table stores information that allows the User Library Technique to find and play songs on the users’ computers. The *Music stations* table, the *Music stations song ratings* table, and the *Music station tag ratings* table stores the basic information for the music stations and the user preference data related to each music station. The *Music station Technique scores* table keeps track of the average score for each Technique. This score is used in the normalization process of the recommendation Technique scores as we have described in Section 4.4.3. The difference between the “ratings” and “scores” in the database tables is that the “ratings” are given by the user whereas the “scores” are from the static music data, such as the relevance score for tags and similarity scores for songs. The *Music station Technique scores* table is an exception to this rule — the values in the *Music station Technique scores* table are considered “scores” in the sense that they are not directly rated by the user but are indirectly calculated by the MusicPlanner.
5.5 Resource Module

The Resource module is implemented as a set of wrapper classes for the music API's and the user music libraries. In implementing the wrapper classes, we define an “interface” for input and output: the member functions take a common set of parameters and return the results in a predefined format. For example, for the music searching functions in the Youtube API wrapper and the user library wrapper classes, each function takes the artist name and the song title as parameters and returns a dictionary containing the url of the song as the result. With this design, the play Techniques, knowing the required set of parameters and the format of the results, can safely interact with the wrapper classes to search for the desired music. This “interface” allows us to keep the implementation organized, and more importantly, it complements our extensible architectural design by serving as a template for implementing wrappers classes for the new resources in the future.
Chapter 6

Evaluation

In Chapter 3, we hypothesized that the Planner should perform better than individual Techniques in recommending songs. However, the Planner can only outperform the Techniques given that there does not exist a Technique that always performs better than the other Techniques, in which case the Planner performs only as well as the best Technique. This chapter describes the methods used to test this hypothesis and analyzes the results.

6.1 Experimental Setup

In order to test this hypothesis that the Planner is superior, a user study is created to assess the quality of the songs recommended by the Planner and each of the Techniques. The user study consists of two stages: the training stage and the evaluation stage. In the training stage, a temporary music station is created to learn the users' music taste by saving their feedback to a set of songs. In the evaluation stage, the music station recommends songs using various Techniques, and users' feedback to the songs are used to measure the performance of each Technique.

The training stage starts with a seed song that the user provides at the start of the user study. The reason for starting with a seed song instead of a random song is that the user-specified seed song allows us to recommend songs that is likely to fit the users' music tastes, while choosing a random song could introduce confounding variables as
the user may or may not like the random song. After the user supplies the seed song, the MusicPlanner will then suggest a total of 10 songs using the Planner. In other words, for each of the songs in the training stage, the Planner selects the song with the highest score among the songs recommended by the Similarity Technique, Tags Technique, and Random Technique. Each user feedback is saved in the temporary user station corresponding to the user study. The cumulative user feedback for the songs played is then used to recommend each subsequent song. The training stage can be seen as building a model of users' music taste so that it can be used to evaluate the different Techniques in the evaluation stage.

The objective of the evaluation stage is to compare the quality of the music recommendation using each of the three Techniques and the Planner, which returns the best result among the results returned by the three Techniques. The quality is measured, in a subjective way, using users' feedback on the recommended songs. Similar to the evaluation stage, each user feedback, along with the user feedback on the previous songs, is used to recommend the next song. In order to prevent the feedback on songs recommended by different Techniques from interfering with each other, a total of four temporary user stations — one for each of the three Techniques and one for the Planner — are created by cloning the temporary station from the training stage. Hence, the user feedback for the songs recommended using each recommendation Technique is only recorded on its corresponding user station. Similar to user feedback on songs outside of the user study, the user feedback in the evaluation stage is limited to a positive or a negative feedback, where a positive feedback corresponds to a score of 1 and a negative feedback corresponds to a score of -1. The cumulative score for each Technique is stored in the database and are updated on each user feedback. For this user study, a total of 20 songs are evaluated: five songs for each of the three Techniques, and five songs for the Planner. At the end of the evaluation stage, the average scores of the recommendation results from each Technique and the Planner are computed.
6.2 Results

Figure 6-1 summarizes the results of the user study conducted on five users selected at random. The user study could be considered as a double-blind experiment as the users were not given any information that could bias their results and the computer recorded the results.

The results indicate that none of the Similarity Technique, Tags Technique, or the Random Technique is always better than the others. This observation confirms the condition specified in the hypothesis: there is no "best" Technique. By comparing the average scores for each of the Techniques with the average score for using the Planner, we can conclude that the hypothesis is correct in that the Planner performs significantly better than using each of the Techniques by itself. Also, the Similarity Technique, Tags Technique, and the Planner greatly outperform the Random Technique for all five users, proving the effectiveness of the recommendation Techniques we have created.
Chapter 7

Conclusion

7.1 Future work

There are three areas that we want to extend the MusicPlanner project. First, we need to improve the scoring heuristic for Techniques so that instead of being a predefined formula, the scoring method changes during runtime based on how well the user liked the recommended songs. The dynamic version of the scoring heuristic can scale the Technique score up or down by a certain factor before returning the score. In this design, the Planner can put more weight on certain Techniques that have been performing well for the current user. Since the original scoring heuristic is an one-size-fits-all approach based on the subjective data from Last.fm – the similarity scores of similar songs and relevance scores of relevant tags – and the method we constructed that computes the score of each candidate song, it doesn’t take into account of the differences in each user’s musical taste. By dynamically modifying the formula based on how well the MusicPlanner is performing for each user, we are more likely to improve the recommendation performance.

Another area that can be improved is the transparency of the Techniques. Since the Techniques are the backbone of the MusicPlanner as all the user’s music requests are ultimately being handled by the competing Techniques, it is important, from the user experience standpoint, to allow the users to view and modify existing Techniques as well as add new Techniques. We can allow users to view and modify of existing
recommendation Techniques by creating a user interface where they can change important data points and parameters, such as the rating of each individual tag, the rating of each of the previously played songs, and the timeframe of the user's past rating history that should be considered in music recommendation. For play Techniques, we can allow the user to rank the different sources of music. To allow users to create new Techniques, we can break down the users into two types: programmers and non-programmers. For the programmers, we can create an API to allow them to call database functions and sub-class the Techniques in the Planner. They can write the new Technique as a Python script and add the Technique by simply uploading the Technique to the MusicPlanner. For the non-programmers, it is challenging to create an interface that can provide them with the flexibility that the programmers have in constructing new Techniques. One possible solution that offers limited flexibility is to create a base Technique for each class of Techniques and allow the user to create new Techniques by modifying the parameters of the base Technique. The base Techniques are hybrid Techniques that combine all the Techniques of its class. For example, the can be added as the new Techniques. For example, the base Technique for recommendation Technique would be a hybrid between Similarity Technique and Tags Technique, and it suggests songs that have the largest weighted sum of component score by Similarity Technique and Tags Technique. The user could customize the weights of the component score by each Technique as well as other parameters of each component Technique.

Lastly, we can extend the MusicPlanner by adding Techniques that "mine" other sources of music recommendations. This thesis describes Techniques that use the recommendation data from Last.fm, but we can also develop Techniques that utilize data from music recommendation websites such as Pandora and Yahoo! Music. We have shown that having competing Techniques can generate better recommendation results; thus we are likely to improve the MusicPlanner by adding Techniques based on the other approaches of music recommendation. This ability to incrementally enhance the MusicPlanner by adding new Techniques highlights the main advantage of using a goal-oriented approach.
7.2 Conclusion

This project is a first step toward a goal-oriented music recommendation system. The results from our limited user study are encouraging: using the Planner achieves better results than using each of the Techniques by itself. In the design and implementation process, we noticed several areas where we benefited from using a goal-oriented framework. In the backend, the goal-oriented design leads to an extensible architecture that allows us to easily extend the MusicPlanner by adding new Techniques. In the frontend, the MusicPlanner intelligently plays the songs that the user may like, requiring only simple positive or negative feedback from the user, and thus minimizing the amount of configuration by the user.

While this project provides a good foundation for the MusicPlanner, there is much to improve on both in the functionality and in the quality of the music recommendation. We described some of the ways to extend the functionalities of the MusicPlanner in our discussion of future work, but the recommendation quality is bottlenecked by the music recommendation methods currently available. Tackling the problem of music recommendation is an ongoing effort. As people come up with new ways to recommend music, using a goal-oriented design allows us to continually improve the MusicPlanner by adding Techniques that utilize these novel approaches.
Bibliography


    http://code.google.com/apis/youtube/1.0/developers_guide_python.html.
