Team X

Design Document: Project Scherzo

Rebecca Young, Lineker Tomazeli, Alicia Bendz

February 14, 2012

COMP361 Software Engineering Project

School of Computer Science

McGill University

Table of Contents

Section 1: Introduction

1.1 Purpose of Document

1.1.1 The two-fold purpose

This document follows the Preliminary Design Report and Requirements Document, and serves three main purposes. The first is primarily in Section II, which presents and discusses the general design of the software; this is important for the client. Included here are also architecture and design considerations, general design pattern identification, and deployment information.

The second purpose is to provide a description of the team's programming methods. We have also selected one module, the Request Handler (see Requirements Document), for which we provide the decomposition of the system design with class diagrams, state charts, and other necessary diagrams. The critical sections, implementation constraints, and special features are also included.

The final section serves the third purpose, giving an overview of the testing for the entire software. Detailed test cases, with their type, method, flow paths, and the tests themselves are specified for a selected class, the Active Queue.

1.1.2 Justification

In selecting the design and programming methods for this project, we have taken into consideration certain guidelines which must be met. Primarily, we are bound by time and quality constraints. Since this project is being completed for a 6-credit course at McGill University, we are scheduling development so that the various document and demo deadlines are met. On the other hand, because we plan to market our software, we also have quality constraints (see Requirements Document for a detailed discussion of these constraints).

We recommend readers to be familiar with the Preliminary Design Report and the Requirements Document in order to have a complete understanding of the project's purpose and goals.

1.2 Intended Audience

Section I and II are addressed with the client in mind, while Sections IV and V cover in more detail the actual development process. The opening sections provide an introduction and a general overview of the software design, including strategies, patterns used, and general class relationships. A client or member of the public reading this document will receive a brief reiteration of the project objective and a general idea of the software structure.

Given sections IV and V, a team of programmers will have a general understanding of the module break-up, how the functionality of each module should be implemented, which libraries and resources to use, and the coding and testing procedures; they should be able to code and

develop the software. We have chosen to focus on the Request Handler Module, showing in detail its functionality and desired implementation. Section IV elaborates upon the programming methods we have chosen.

1.3 References and Definition of Terms

This document refers to the previous documents, the Preliminary Report and Requirements Document. For definitions of terms relating to the software modules, please refer to Section 1.3 of the Requirements Document.

We have chosen “Scherzo” as the name of our software.

1.4 Policies and Tactics

Our goal is to follow the design strategies and patterns presented in this document as closely as possible. This will be one of our metrics for analyzing the quality of our chosen software development method and the development process as a whole. The development method is covered in detail in Section IV.

Section 2: The Design

2.1 Design Goals

All design and engineering decisions will be taken with the following principle in mind:

Simple is better than complicated.

We will always try to eliminate choices that will make the usability of the application complicated. Every bump on the user experience must be relentlessly sanded down. Our customers will say: “It’s the easiest thing in the world and It Just Works.” No jargon will remain in the user experience, anywhere.

2.2 Architecture physical considerations

The Scherzo system has two major software components. One runs on a client mobile device and the other on a desktop computer.

Scherzo Server

The Scherzo Server application will be separated into various software modules. The main packages provides a user interface for playing local music files, creating playlists, and a report generator detailing information about played songs. Scherzo Server and its modules will be written in the Java (http://java.sun.com) programming language.

Minimum requirements for Scherzo Server are:

OS: Windows® Vista, Windows® 7, Mac® OS X 10.6.X, 10.7.X

Processor: Intel® Core 2 Duo or AMD AthlonTM 64 X2 4400+
Memory: 2 GB RAM
JVM: Java Virtual Machine (JVM) version 6 or above.

The Scherzo Server is a standalone application. It will not fully work if an Internet connection or a LAN network is not available. Two extra requirements should be consider if you would like to active these features:

- To enable streaming it is required that the computer running Scherzo have access to the Internet. A minimum speed of 512 kbps is required.
- To enable mobile client requests it is required that the computer running Scherzo and the mobile clients are in the same range of ip addresses and submask. (http://en.wikipedia.org/wiki/Classful_network). This could be achieved by providing a wireless router where mobile clients can connect to the network. TeamX removes any responsability for this set up.

Scherzo Mobile

Scherzo Mobile application will be provide a simple and intuitive interface for user to list songs, request songs and send feedback (for more see section xx.xx requirement document).

Scherzo Mobile and its modules will be written in two different languages. Objective-C for iOS devices (http://developer.apple.com/) and Java for Android devices (http://java.sun.com).
Minimum requirements for Scherzo Mobile are:

iOS
OS: iOS 4.3.X and above
Network: Wireless connection

Android
OS: Android 2.3.6 and above
Network: Wireless connection

2.3 Architecture software considerations

This section we will discuss the external resources that we will be using to develop the two major components, Scherzo Server and Mobile. Each major component is separated into various modules.

We will be using Eclipse Galileo with JRE System Library JVM 1.4 for development.

For testing devices we will be using an Iphone 4 and a Samsung Nexus S.

2.3.1 Scherzo Server Modules

2.3.1.1 Music Manager

The Music Manager is the hub of all the modules. It contains and monitors the Active Queue, Playlist, and Music Player, and appropriately processes all song requests sent from the Request Handler. It sends the corresponding success messages back to the Request Handler once the request has been processed.

This module also monitors the Active Queue, assuring that the number of songs there do not fall below a specified minimum. When a certain number is reached, it requests another set from the Artificial Intelligence Song Generator and adds it to the Active Queue. The Active Queue, which keeps track of its list of songs and their playing times, is controlled directly by the Music Manager.

With the first version of Scherzo Server, v1.0, we are planning to support the following file types:

- mp3 - wikipedia.org/wiki/MP3
- wav - wikipedia.org/wiki/WAV

*Later versions will include additional support.

This module will be using the follow libraries and frameworks (to achieve its requirements):

JLayer Mp3SPI	Used to process and play music files from local machine or from a streaming source.
JavaID3 Tag library	Used to access metadata information of mp3 files
Joda Time	Used in place of the Java Date class, for fewer complications and faster access to information http://joda-time.sourceforge.net/

2.3.1.2 Request Handler

The Request Handler is the direct link between the client applications and the Music Manager. It receives all song requests and text feedback which have successfully been transmitted across the network. Feedback is compiled into the database, and song requests are verified to be valid requests and existing in the database, then passed to the Music Manager. This module will be using the follow libraries and frameworks (to achieve its requirements):

java.net.Socket	All communication between mobile devices and the server is done through

	socket. http://docs.oracle.com/javase/1.4.2/docs/api/java/net/Socket.html

2.3.1.3 Database

To persist the information from imported songs, statistics and playlists, Scherzo will rely on a relational database that supports a Java DataBase Connectivity (JDBC) driver. Scherzo will use SQLite for our relational database system.
Scherzo Server will have a software package dedicated to interact will the local database.

This module will be using the following libraries and frameworks (to achieve its requirements):

QLite JDBC Driver	Driver used to interact to a sqlite local database. http://www.xerial.org/trac/Xerial/wiki/SQLiteJDBC

2.3.1.4 Streaming Manager

The Streaming Manager connects to the server from which music files are streamed. It permits searching among available files, oversees the streaming of selected files, and returns status and success messages at the request of a song stream. It is the direct and only link between the streaming server and the rest of the system.

With the first version of Scherzo Server, v1.0, we are planning to support the following streaming services:

- GrooveShark http://en.wikipedia.org/wiki/Grooveshark / http://www.grooveshark.com

This module will be using the follow libraries and APIs (to achieve its requirements):

Grooveshark Public API Version 3	API used to search and stream songs http://developers.grooveshark.com/docs/public_api/v3/
Tinysong API	API used to search and get information about songs http://tinysong.com/api
Google-Gson	Google library for handling json http://code.google.com/p/google-gson/

2.3.1.5 Statistics Manager

The Statistics Manager is responsible for generating reports based on information collected about songs played.

This module will be using the follow libraries and frameworks (to achieve its requirements):

JFreeChart	Library used to draw charts http://www.jfree.org/jfreechart/

2.4 Assumptions

1. We are assuming that the administrator users have some kind of wireless network on-site, as well as a local server onto which they can install the main application. We are assuming that there is minimal demand so that bandwidth will not pose a problem.
2. It's assumed that the router managing the mobile devices will distribute automatically ip addresses for new devices (dhcp)
3. Users must have a premium account in order to stream songs from the streaming service.
4. In order to have any interaction with the system, client users must have either an Android or a smart iOS device. They also must have already downloaded the application, or may download it on-site if wireless internet connection is available.
5. The local computer hosting the server application must be capable of running the Java Virtual Machine for our Java application to operate.
6. The initial version of the software will be in the English language, so users must understand English.
7. We assume that the local computer hosting the server application is connected in some way to the speaker system.

2.5 Dependencies

1. The streaming functionality is dependent upon the requirements from the Streaming Service chosen. Also, if for some reason the Streaming Service is down, the application wont be able to play streaming songs.
2. The streaming functionality also depends on the availability and reliability of the Internet connection

2.6 General constraints

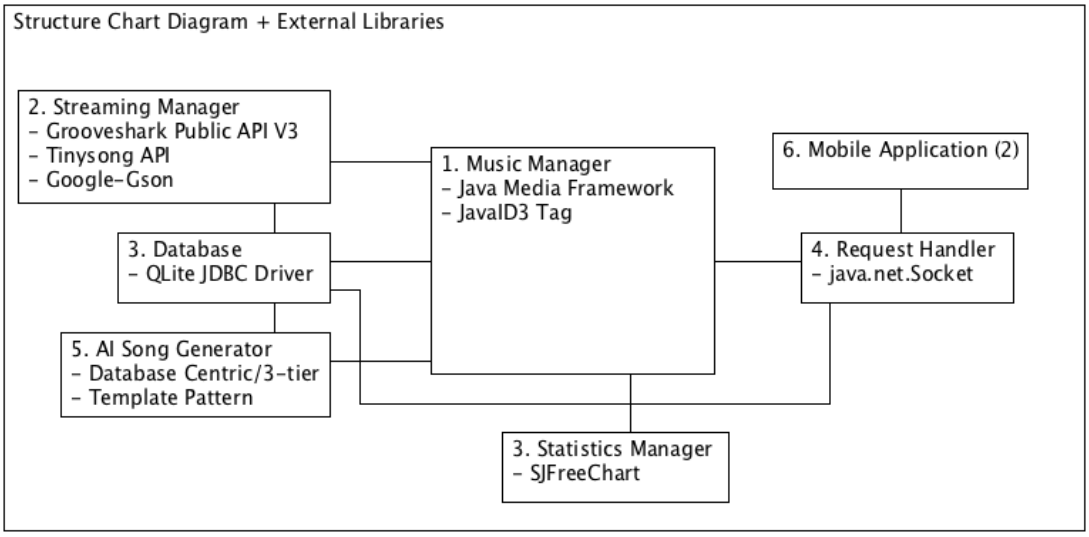
Network limitations: For full functionality to satisfy the user needs constraints, a local wireless network must be available on-site, and the connection must be consistent. The system's Request Handler and Streaming Manager will need to deal with and adjust to slower networks; however the main application will adapt and still play background music even without all the complete functions.

The network security setting could also be an issue. If the setting is private and requires a password, the administrator must make the adjustments that allow, exceptionally, the client application to connect across the network.

Parallel operation: the main application on the server and many applications on smart devices must be able to execute at the same time. The main application's Request Handler will manage receiving multiple requests simultaneously.

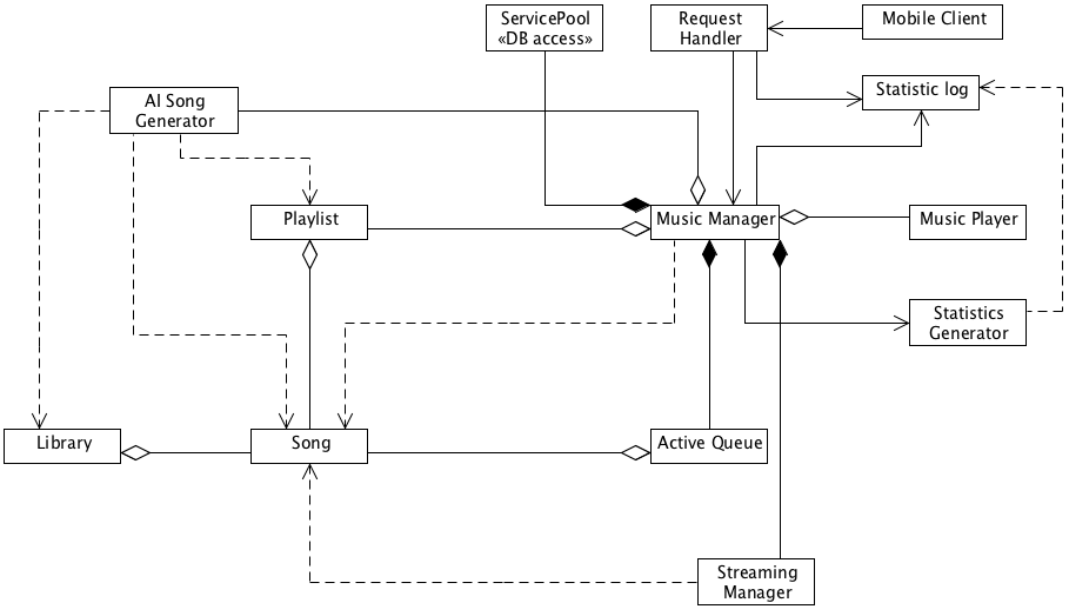
2.7 High-level design

2.7.1 System overview discussion



The diagram above shows the interaction between each module. Also indicates the external libraries that each module is using it. See section 2.3 for more information on the external libraries.

2.7.2 Design pattern identification

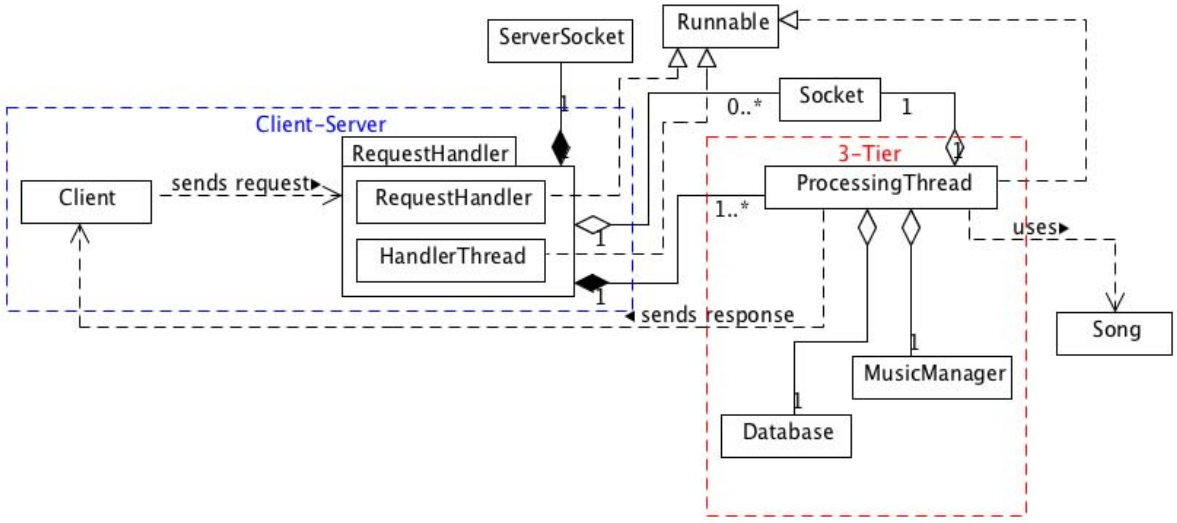


The diagram above represents a final interaction between the major objects in the Main application

Please see below for each module and their patterns identifications.

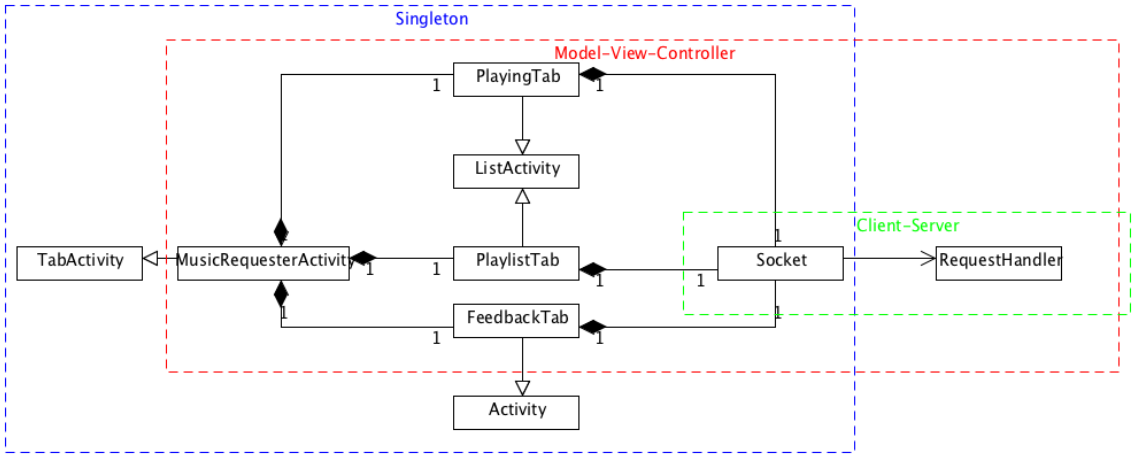
2.7.2.1 Request Handler

- Client – Server: Server interacts with client device via network.
- Database Centric/3-Tier: Server threads interact with database.
- Singleton: There is only one instance of the server.



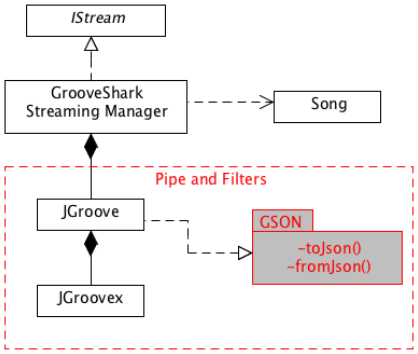
2.7.2.2 Android / iOS

- Singleton: The client only contains one instance of itself.
- Client-Server: The client (Android/iOS device) communicates with the server via network.
- Model-View-Controller: The model is the information on the server, the view is what the device displays to the client, and the controller is the device software.



2.7.2.3 Streaming Manager

- Pipe and filter: The communication done between the Streaming Manager and the Streaming Service is done through json. The request is first formatted in the required format and send to the Streaming service. The information returned then is parsed and converted to objects if necessary.

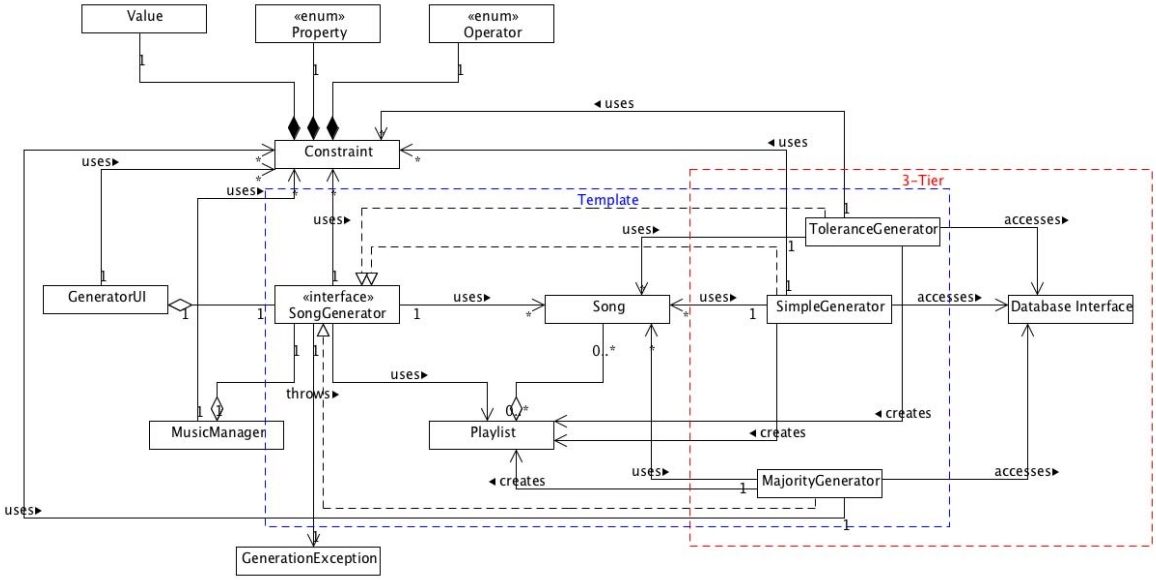


The Streaming Manager Module implements and simple Pipe and Filter pattern converting concrete object like Song to a JSON string format. The JSON data is then sent to the Streaming provider which returns a JSON response. The response is then translated back to object so it can be used through the application.

We are using an Google GSON library to do the JSON handling. For more, refer to http://code.google.com/p/google-gson/.

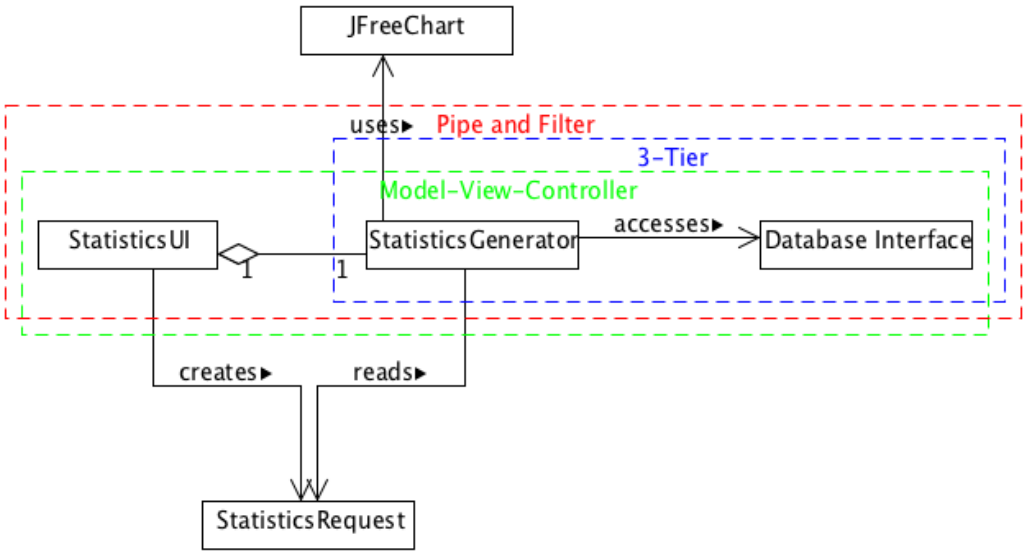
2.7.2.4 Song Generator

- Database Centric/3-Tier: The song generator uses information in the database to find songs.
- Template: The generator is an interface with a simple generator implementation. This is implemented in this way for possible expansion later.



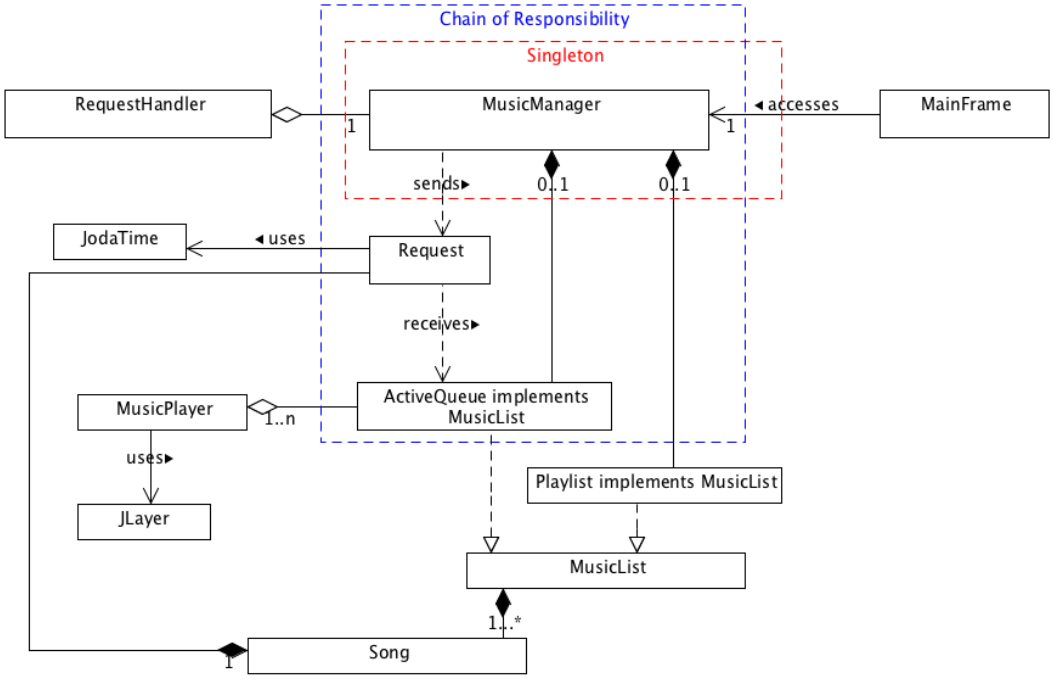
2.7.2.5 Statistics Generator

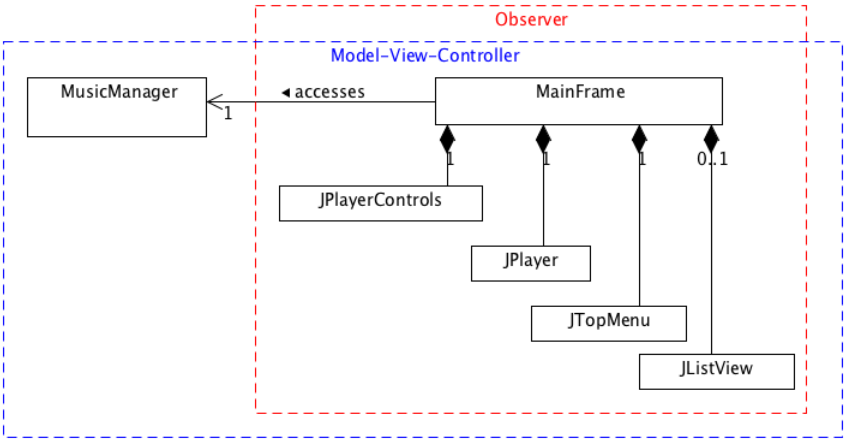
- Database Centric/3-Tier: The statistics generator accesses the database to get data.
- Model-View-Controller: The model is the database information, the view is what the user is shown, and the controller is the statistics generator which displays the data according to what the user requests.
- Pipe and filter: The statistics request is taken from the user, formatted into a request object, sent to the statistics generator and parsed. Once the request has been parsed, data from the database is gathered and the desired output is created. Once the output has been generated, it is returned to the user interface component and displayed to the user.



2.7.2.6 Music Manager

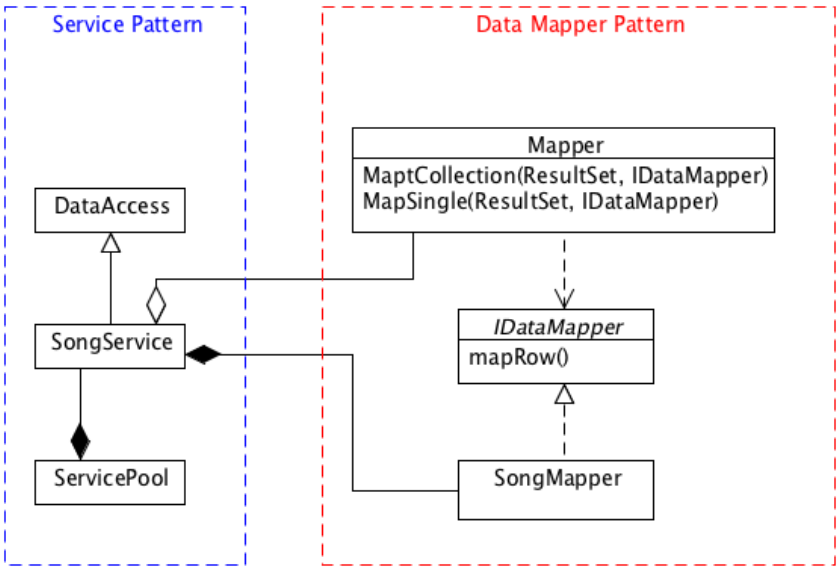
- Observer Pattern: There is a one-to-one dependency between the JPlayerControls, JPlayer, JTop Menu, and the MainFrame. Whenever one object changes state, the MainFrame is notified and updates all of the other dependant objects.
- Chain-of-Responsibility: The event generated by the RequestHandler is passed to the MusicManager, which in turn processes the request and stores it until it can be received by the ActiveQueue.
- Singleton: There is only one instance of the MusicManager, and only one Playlist and ActiveQueue at a time. The ActiveQueue also has only instance of the MusicPlayer thread at once, and closely monitors it such that a new thread is spawned only after termination of the previous one.
- Model-View-Controller: The model is the current contents of the ActiveQueue's list, the view is the GUI that is repainted, and the controller is the MusicManager which imposes changes on the ActiveQueue.


Chain of Responsibility
Singleton
RequestHandler
MusicManager
accesses
MainFrame
1
1
sends
0..1
0..1
JodaTime
uses
Request
receives
MusicPlayer
1..n
ActiveQueue implements MusicList
uses
JLayer
Playlist implements MusicList
MusicList
1..*
1
Song


Observer
Model-View-Controller
MusicManager
accesses
MainFrame
1
1
1
1
0..1
JPlayerControls
JPlayer
JTopMenu
JListView

2.7.2.7 Database

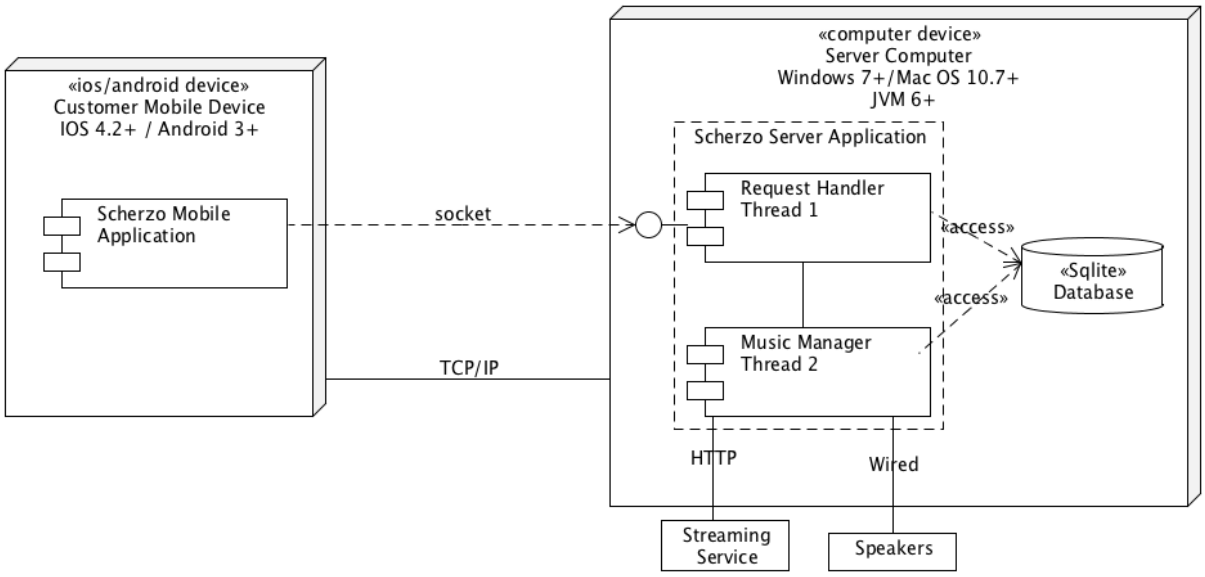
- Data Mapper Pattern : The Data Mapper is a layer of software that separates the in-memory objects from the database. Its responsibility is to transfer data between the two and also to isolate them from each other. A mapper class receives the SQL ResultSet of a query and convert it to a in-memory object. See more http://martinfowler.com/eaaCatalog/dataMapper.html
- Service Pattern : The Service Pattern contains as many Services you want. Every service communicates to the database. Each Service is focused in dealing with a specific object. Example: SongService deals with all action dealing with Songs and the Database, like add new song, update a song and etc. The Service Pattern has a Service Pool which holds a reference to every single service available in the application. The Service Pool make easy the communication from any module to the database through the specific services.



The diagram above represents the two design patterns that our database access module should implement. The Data Mapper Pattern which is pattern that will help us in binding database data into concrete object. The diagram above shows and example using the Song object and it's mapper. The specific IDataMapper implementation and the ResultSet object should be given to the Mapper class which will do the binding according to the IDataMapper given.

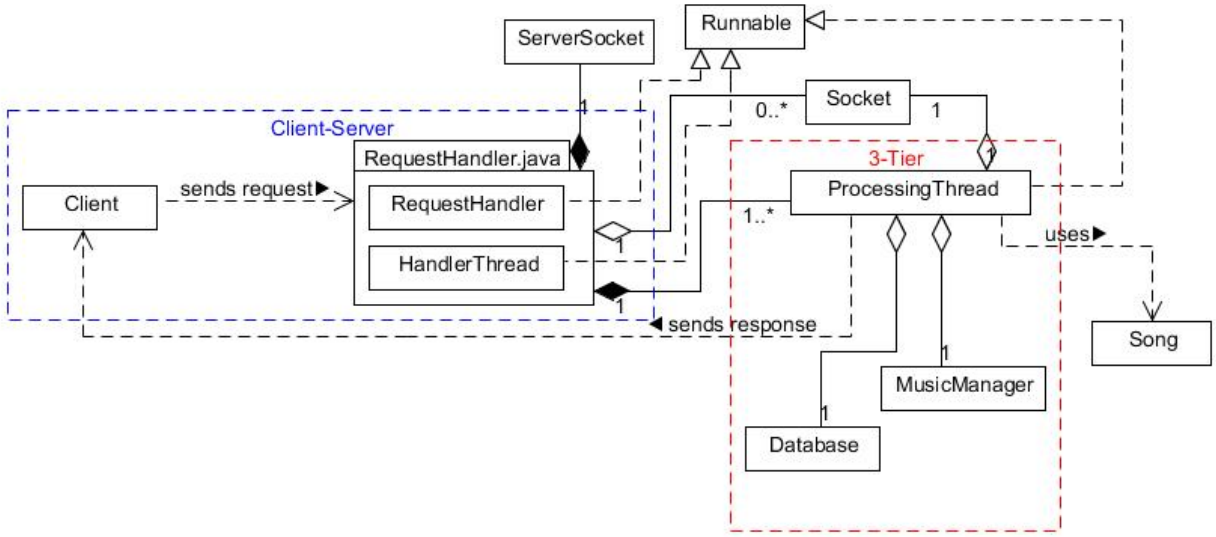
The second pattern is the Service Pattern which has the goal of simplifying and organizing database access. It creates a structure of Service to the developer so it can focus on specific tasks to specific object. For example, the diagram above shows the SongService which should have all the CRUD methods related to Songs. Also it make possible a easy access to all service through the service pool.

2.7.3 Final deployment diagram



Section 3: Handler Module

3.1 Introduction



The above diagram is a simplified class diagram of the Request Handler module. The place of the Request Handler module within the rest of the project can be seen in 2.7.1. The core classes of the request handler are the RequestHandler, the HandlerThread, and the ProcessingThread. Other classes in the diagram are either representative of other modules, such as the Client and Database, or are classes written elsewhere, such as Song and Socket.

The following design patterns are present in the Request Handler module: Client-Server, 3-Tier, and Singleton. In addition to these design patterns, the ProcessingThread on its own internally has execution similar to that of the Pipe and Filter design pattern.

The Client-Server pattern can clearly be seen with the interaction between the Client module and the Request Handler module. The client sends messages via a network connection to the Request Handler. Upon receiving a message from the client, the Request Handler does some execution and may return a message to the Client. The Client and the Request Handler are independent of each other and there can be several Clients connecting to the Request Handler at once.

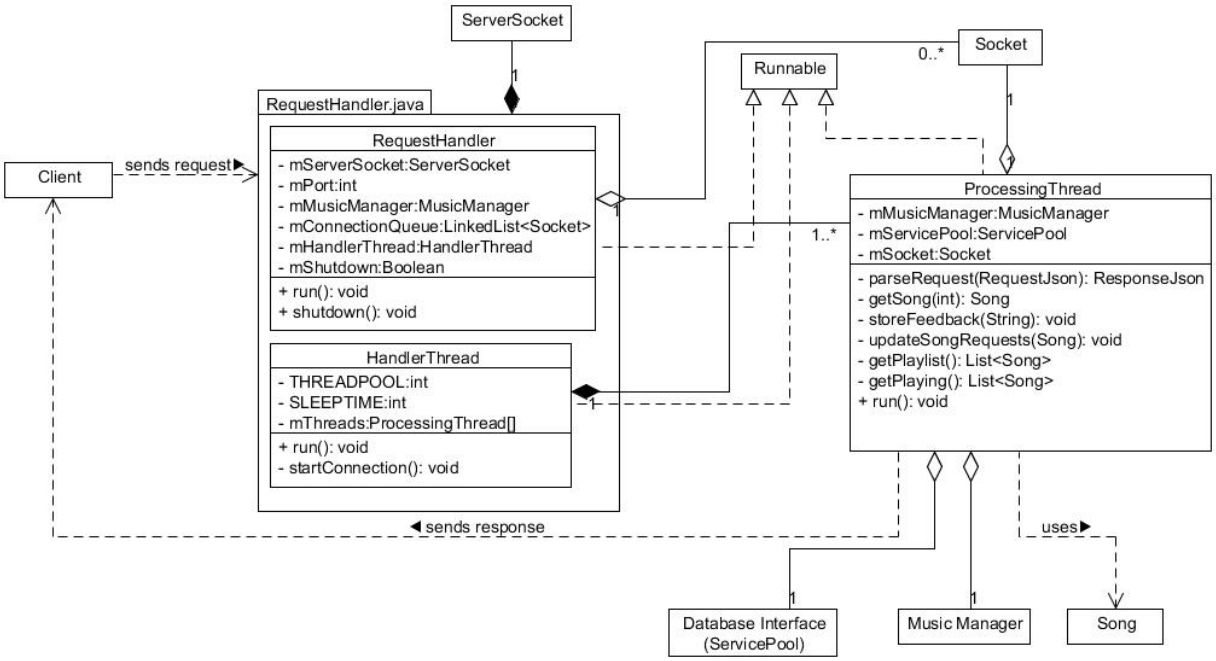
The 3-Tier pattern is also fairly clear between the ProcessingThread and the Database. There will be several ProcessingThreads each using an interface to the Database module. In addition to the ProcessingThreads, other modules of the entire system will also be using the interface to the Database.

There is only one instance of the RequestHandler and its HandlerThread. This is necessary since there should only be one Socket listening for client connections for the application. Here the Singleton pattern in is use.

Internally in the ProcessingThread, a sequence of operations must be done on a received message, bearing resemblance to the Pipe and Filter design pattern. First a message is read by the ProcessingThread. This message is then passed to a parsing method. The parsing method with break the message up and identify the request. Once this is done, the request is passed along to the necessary methods for execution. During execution of the request, additional information may be retrieved from the Database or MusicManager. Once all information is gathered, execution is completed. The end result of execution is formatted and then returned to the client.

The pipes in this case are the network connection, methods that pass the message or request along to other methods, and the components of execution that gather data from other modules. The filters would be any method in the ProcessingThread that breaks or forms a formatted message, or takes components of the request and carrying out some execution to get a final result.

3.2. Class Diagram

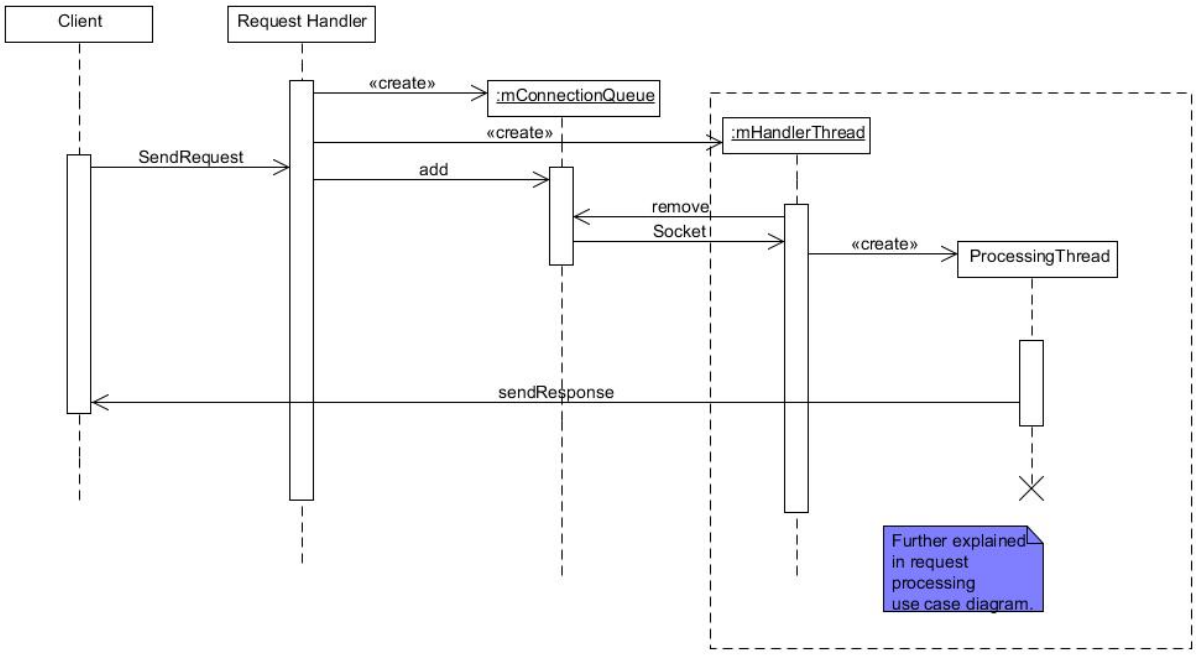


The RequestHandler module itself consists of two main components: the RequestHandler and the ProcessingThread. The module communicates with the Client module as well as the Database and the MusicManager. Each component of the RequestHandler uses Sockets from the java.net library to communicate with the Client. Only the RequestHandler uses a ServerSocket to accept connections from the Client. The RequestHandler has two sub-components: the RequestHandler and the HandlerThread. The HandlerThread is an inner class of the RequestHandler. The RequestHandler, HandlerThread and ProcessingThread all run as

their own thread, all implementing the Runnable interface, to allow multiple requests to be handled at the same time.

A Client will first make a connection with the RequestHandler. This connection will be added to a queue of connections waiting for processing. The HandlerThread goes through the queue of connections and starts and ProcessingThread for each. The ProcessingThread then finishes the interaction with the Client, accessing the Database and MusicManager as required. Requests for Songs will be stored in the Database and sent to the MusicManager. The MusicManager will be needed for requests for the playlist and currently playing set of songs.

3.3. Execution After Client Connection



Upon startup, the RequestHandler creates its list of connections for the Client as well as starting the HandlerThread. Once the RequestHandler has been set up, it can accept connections from the Client module. The Client module makes a connection and sends a request to the RequestHandler. The RequestHandler adds this connection to the ConnectionQueue upon success. The HandlerThread then takes the connection and starts a ProcessingThread to interact with the Client connection. At this point, there are several different paths of execution and more complex interactions between the HandlerThread and ProcessingThread. This execution and interaction is further explained in section 3.4. After execution has completed, the ProcessingThread returns the results to the Client and terminates.

3.4. Processing Thread Interaction

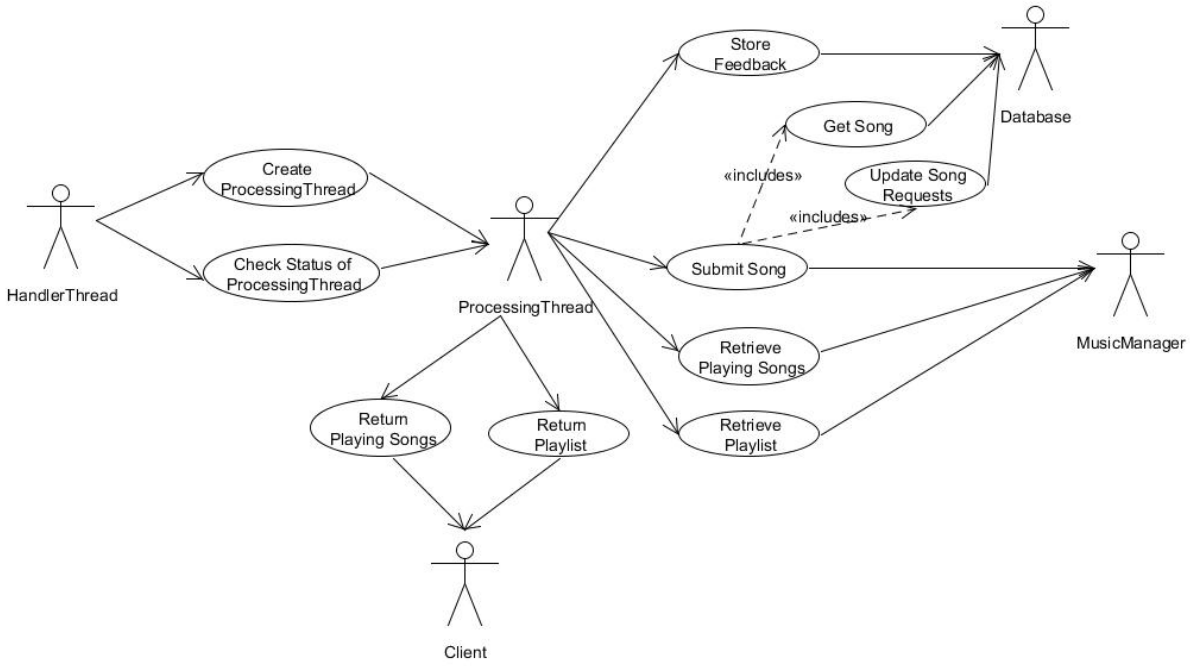
3.4.1. Actors

ProcessingThread	The thread that interacts with the Client after the connection has been made. Any requests the Client sends will be read and responded to by this thread.
HandlerThread	This thread is responsible for starting ProcessingThreads for each Client connection.
MusicManager	This is the module of the application which is aware of currently playing songs and songs that are available for play.
Client	This is the module of the application which is present on the external devices sending requests to the server component of the application.
Database	This is the module storing all the data and information regarding songs, playlists, song request data, and feedback.

Note: The descriptions of the actors in these use cases are simplified for this particular use case. See the module breakdown in the Requirements Document for more detail on the MusicManager, Database, Android application (Client), and iOS application (Client).

3.4.2. Processing Thread Use Case Interaction

3.4.2.1. Use Case Diagram



This use case describes the possible actions of the HandlerThread and the ProcessingThread after a Client has already made a connection with the server and sent a

request. The use case does not show how the Client makes the connection nor how it sends the request. This is shown in section 3.3.

3.2.4.2. Create ProcessingThread Template

Use Case Name:	Create ProcessingThread
Iteration:	Focused
Summary:	The HandlerThread starts a ProcessingThread and gives it a connection to a Client. This Client will now interact with the ProcessingThread for its request.
Basic Course of Events:	Check for waiting Client connections. Check to see if there are not too many threads running already. See template 3.2.4.3. Start a ProcessingThread and give it the current Client connection to begin processing. Return to 1.
Alternative Paths:	If there are no Client connections, wait for a connection. 2.1 If there are too many threads running, wait for one to finish.
Exception Paths:	None.
Extension Points:	None.
Triggers:	RequestHandler is started and ready to accept connections.
Assumptions:	Multiple threads of execution are allowed in the system. The RequestHandler program is successfully creating connections with Clients.
Preconditions:	There is a Client connection waiting for request processing.
Postconditions:	There is one less Client connection waiting for request processing.
Related Business Rules:	None.
Author:	Alicia Bendz
Date:	December 23, 2011

3.2.4.3. Check Status of ProcessingThread Template

Use Case Name:	Check Status of ProcessingThread
Iteration:	Focused
Summary:	Check to see if a ProcessingThread is still running or if it has terminated.
Basic Course of Events:	Find thread position in array of ProcessingThreads. Check status of ProcessingThread. If thread has terminated, set value of this position to null.
Alternative Paths:	3.1 Thread is still running; do nothing.
Exception Paths:	None.
Extension Points:	None.
Triggers:	A new Client connection is waiting and the HandlerThread must find a position in the thread pool to start a ProcessingThread for it.
Assumptions:	The ProcessingThread was created and started at some point previous to this check.
Preconditions:	Status of ProcessingThread is unknown.
Postconditions:	Status of ProcessingThread is known.
Related Business Rules:	None.

Author:	Alicia Bendz
Date:	December 23, 2011

3.2.4.4. Return Playlist Template

Use Case Name:	Return Playlist
Iteration:	Focused
Summary:	The ProcessingThread sends the Client the current playlist via TCP connection in an agreed upon format.
Basic Course of Events:	ProcessingThread creates formats the response with the provided playlist from template 3.2.4.11. ProcessingThread sends the response to the Client via established TCP connection. ProcessingThread disconnects and terminates upon successful arrival of response.
Alternative Paths:	None.
Exception Paths:	3.1 ProcessingThread times out waiting for Client to receive response, disconnects, and terminates.
Extension Points:	None.
Triggers:	The ProcessingThread obtains the playlist from the MusicManager as per previous request of Client to which it must send the playlist.
Assumptions:	A playlist was retrieved previously in the Retrieve Playlist use case. The Client has not timed out waiting for the response. The Client knows of the agreed upon format for the response.
Preconditions:	ProcessingThread still has connection to Client and is running. Client is waiting for response.
Postconditions:	ProcessingThread is terminated. Client is has received response.
Related Business Rules:	None.
Author:	Alicia Bendz
Date:	December 23, 2011

3.2.4.5. Return Playing Songs Template

Use Case Name:	Return Playing Songs
Iteration:	Focused
Summary:	The ProcessingThread sends the Client the currently playing songs via TCP connection in an agreed upon format.
Basic Course of Events:	ProcessingThread creates formats the response with the provided songs from template 3.2.4.10. ProcessingThread sends the response to the Client via established TCP connection. ProcessingThread disconnects and terminates upon successful arrival of response.
Alternative Paths:	None.
Exception Paths:	3.1 ProcessingThread times out waiting for Client to receive response, disconnects, and terminates.
Extension Points:	None.
Triggers:	The ProcessingThread obtains the playing songs from the MusicManager as per previous request of Client to which it must send the list of playing songs.

Assumptions:	The list of playing songs was retrieved previously in the Retrieve Playing Songs use case (3.2.4.10). The Client has not timed out waiting for the response. The Client knows of the agreed upon format for the response.
Preconditions:	ProcessingThread still has connection to Client and is running. Client is waiting for response.
Postconditions:	ProcessingThread is terminated. Client is has received response.
Related Business Rules:	None.
Author:	Alicia Bendz
Date:	December 23, 2011

3.2.4.6. Update Song Requests Template

Use Case Name:	Update Song Requests
Iteration:	Focused
Summary:	Update the request count for a requested song in the Database.
Basic Course of Events:	Following the Get Song Request use case (3.2.4.9.), use the retrieved song to update the request count of this song in the database. Proceed to submit the Song according to the Submit Song use case (3.2.4.8.).
Alternative Paths:	None.
Exception Paths:	None.
Extension Points:	None.
Triggers:	Submit Song use case (3.2.4.8). When a Song is to be submitted via request, the request count for that Song is updated.
Assumptions:	The Song exists in the Database.
Preconditions:	None.
Postconditions:	The Song request count is greater by one in the Database.
Related Business Rules:	None.
Author:	Alicia Bendz
Date:	January 27, 2012

3.2.4.7. Store Feedback Template

Use Case Name:	Store Feedback
Iteration:	Focused
Summary:	Store feedback sent by the Client in the Database.
Basic Course of Events:	Get feedback content from message sent by Client. Store feedback in Database. Close connection with Client and terminate ProcessingThread.
Alternative Paths:	None.
Exception Paths:	None.
Extension Points:	None.
Triggers:	Client has send request to give feedback.
Assumptions:	Client connection has already been established over network.
Preconditions:	None.
Postconditions:	Client feedback has been stored in the Database.
Related Business Rules:	None.
Author:	Alicia Bendz

Date:	Decemeber 23, 2011

3.2.4.8 Submit Song Template

Use Case Name:	Submit Song
Iteration:	Focused
Summary:	A ProcessingThread sends a Song request to the MusicManager after verifying that the Song exists in the Database and storing the request in the Database.
Basic Course of Events:	Get Song according to Get Song use case (3.2.4.9). Update Song Requests according to Update Song Requests use case (3.2.4.6). Submit Song to the MusicManager. Close connection with Client and terminate ProcessingThread.
Alternative Paths:	None.
Exception Paths:	None.
Extension Points:	None.
Triggers:	Client has sent a song request and a ProcessingThread has been started with a connection to this Client.
Assumptions:	Client connection has already been established over network.
Preconditions:	None.
Postconditions:	MusicManager has a new Song requested for play.
Related Business Rules:	None.
Author:	Alicia Bendz
Date:	January 27, 2012

3.2.4.9 Get Song Template

Use Case Name:	Get Song
Iteration:	Focused
Summary:	The ProcessingThread verifies that a Song can be played by attempting to retrieve it from the Database before submitting the Song as a request.
Basic Course of Events:	Within the Submit Song use case (3.2.4.8), retrieve the desired song for request from the Database. Continue with the Update Song Requests use case (3.2.4.6).
Alternative Paths:	2.1 Song does not exist and cannot be requested. Request is ignored and Processing Thread is terminated.
Exception Paths:	None.
Extension Points:	None.
Triggers:	A Song has been requested for play and its existence needs to be verified.
Assumptions:	Song exists in Database.
Preconditions:	None.
Postconditions:	Processing Thread has existing Song reference.
Related Business Rules:	None.
Author:	Alicia Bendz
Date:	January 27, 2012

3.2.4.10 Retrieve Playing Songs Template

Use Case Name:	Retrieve Playing Songs
Iteration:	Focused
Summary:	ProcessingThread requests the list of currently playing songs from the MusicManager.
Basic Course of Events:	ProcessingThread requests list of currently playing Songs from the MusicManager. Continues to Return Playing Songs use case (3.2.4.5).
Alternative Paths:	None.
Exception Paths:	None.
Extension Points:	None.
Triggers:	Client has sent a request to see the list of currently playing songs.
Assumptions:	There is a list of currently playing songs available.
Preconditions:	None.
Postconditions:	None.
Related Business Rules:	None.
Author:	Alicia Bendz
Date:	December 23, 2011

3.2.4.11 Retrieve Playlist Template

Use Case Name:	Retrieve Playlist
Iteration:	Focused
Summary:	ProcessingThread requests the current playlist from the MusicManager.
Basic Course of Events:	ProcessingThread requests the current playlist from the MusicManager. Continues to Return Playlist use case (3.2.4.4).
Alternative Paths:	None.
Exception Paths:	None.
Extension Points:	None.
Triggers:	Client has sent a request to see the current Playlist.
Assumptions:	There is Playlist available.
Preconditions:	None.
Postconditions:	None.
Related Business Rules:	None.
Author:	Alicia Bendz
Date:	December 23, 2011

3.5. Class Breakdown

The RequestHandler component is meant to handle communication between the client and the server on the server side of the system. This module will have three different types of threads running concurrently: RequestHandler, HandlerThread, and ProcessingThread. The RequestHandler is responsible for creating connections to the client. The HandlerThread is responsible for starting a ProcessingThread for each client connection within a limit specified by an integer constant *THREADPOOL*. A ProcessingThread takes a client connection and proceeds

to process the request of that client until the client disconnects. The ProcessingThread may need to interact with the Database and the MusicManager in order to carry out the requested operations of the client. The sections below will describe the different variables and methods in each of these classes.

3.5.1. Request Handler Class

3.5.1.1. Server Socket

The ServerSocket in the RequestHandler is called *mServerSocket*. It is a java.net.ServerSocket. This ServerSocket is initialized with the port in 3.5.1.2, *mPort*, and listens for incoming client connections. Upon each incoming connection, a Socket is created with the connection to the client using the *accept*() method provided by the ServerSocket interface. The newly created Socket is stored in the connection queue described in 3.5.1.4.

3.5.1.2. Port

The *mPort* variable in the RequestHandler is an integer. This variable is initialized on start-up with a parameter from the RequestHandler constructor. This variable is used to initialize *mServerSocket*, described in 3.5.1.1.

3.5.1.3. Music Manager

The RequestHandler keeps a reference to the system Music Manager, *mMusicManager*. This reference is passed to the RequestHandler through its constructor. This variable is used when ProcessingThreads are created so that the reference can be passed along easily.

3.5.1.4. Connection Queue

The connection queue, *mConnectionQueue*, is a LinkedList instantiated with type Socket. This LinkedList will be treated as a queue with the least recently added connection processed first and new connections added to the end of the LinkedList. This queue is implemented as a LinkedList in order to use its queue-like interface. This variable is accessed by both the RequestHandler and the HandlerThread. Section 3.6 describes the solution to the necessary concurrent access for this variable. This queue will operate as a FIFO queue.

3.5.1.5. Handler Thread

The RequestHandler creates and starts a HandlerThread upon start-up and maintains a reference to it, *mHandlerThread*. This reference is used to check the status of the HandlerThread and terminate it when the RequestHandler is shut down.

3.5.1.6. Shutdown Variable

There is a boolean variable, *mShutdown*, that controls whether the RequestHandler and HandlerThread continue to execute. This variable is initially set to false, allowing both the RequestHandler and HandlerThread to continue to execute. When the RequestHandler's *shutdown()* method is called (3.5.1.8), it is set to true.

3.5.1.7. Run

The RequestHandler implements the Runnable interface and overrides the *run()* method of that interface. In this method, all necessary variables are initialized, the HandlerThread,

mHandlerThread, is started, and the RequestHandler accepts connections until the *mShutdown* variable is set to false.

3.5.1.8. Shutdown Method

The RequestHandler stops when its *shutdown()* method is called. This method will stop the RequestHandler from accepting connections by closing the ServerSocket, *mServerSocket*. The RequestHandler ignores any waiting client connections and terminates the HandlerThread, *mHandlerThread*, and itself. Both the HandlerThread and the RequestHandler will run until *mShutdown* is set to true. This variable behaves as a switch to both methods to stop execution.

3.5.2. Handler Thread Class

3.5.2.1. Thread Pool

The HandlerThread has a static integer variable *THREADPOOL* to indicate the maximum number of ProcessingThreads that can be running at the same time. This variable is used to set the size of the array of ProcessingThreads in 3.5.2.3.

3.5.2.2. Sleep Time

SLEEPTIME is a static integer variable used to indicate how long the HandlerThread should wait to check for more connections. If there are no connections in the *mConnectionQueue* (3.5.1.4), it will sleep for SLEEPTIME before checking again.

3.5.2.3. Processing Thread Array

This array of ProcessingThreads, *mThreads*, keeps references to all active ProcessingThreads and is initialized with null values. A null value in this array indicates a free position in the thread pool. Each new ProcessingThread is put into a position in this array and removed when its execution finishes.

3.5.2.4. Run

The HandlerThread implements the Runnable interface and overrides the *run()* method of that interface. In this method, all necessary variables are initialized that were not initialized in the constructor. After initialization is done, until the RequestHandler is shut down this method will loop through starting ProcessingThreads for each waiting connection with the *startConnection()* method 3.5.2.5.

3.5.2.5. Start Connection

The *startConnection()* method is used to start a ProcessingThread with a waiting connection. This method will wait for a free space to open up in the ProcessingThread array in 3.5.2.3 and then create and start a ProcessingThread with the first waiting connection from the connection queue (3.5.1.4).In order to free positions in the array of ProcessingThreads, this method will poll existing ProcessingThreads for their status. If the ProcessingThread is still running, it is left alone. If the ProcessingThread has terminated, its reference will be replaced with a null value to indicate a free position in the array.

3.5.3. Processing Thread Class

3.5.3.1. Music Manager

Each ProcessingThread keeps a reference to the system Music Manager, *mMusicManager*. This reference is used to submit song requests (see use case 3.2.4.8), get the current playlist (3.5.3.8), and get the currently playing list of songs (3.5.3.8).

3.5.3.2. Database Interface

Each ProcessingThread has a database interface, *mDatabaseInterface*. This is used to store feedback (3.5.3.6), get a Song from the Database (3.5.3.5), and update song request counts (3.5.3.6).

3.5.3.3. Socket

Each ProcessingThread has a Socket, *mSocket*, which it uses to communicate with the Client. The ProcessingThread only interacts with once Client through this Socket and when communication is ended, this Socket is closed and the ProcessingThread terminates.

3.5.3.4. Parse Request

The *parseRequest()* method is meant to take the Json object RequestJson and executing the correct action based on its contents. This method will return the response to be sent to the client if there is one in the form of a ResponseJson. There are four possibilities for a request: store feedback (3.5.3.6), request a song (3.2.4.8), request the playlist (3.5.3.8), and request the list of currently playing songs (3.5.3.8).

3.5.3.5. Get Song

The *getSong()* method verifies the existence of a song in the Database by attempting to retrieve it. If the song does not exist, the error is logged and a null value is returned. If the song exists, the Song object for that song is returned. This method is called when a request for a song is made (see use case 3.2.4.8).

3.5.3.6. Store Feedback

If a client has sent feedback, the message sent by the client will contain the feedback and any necessary information for the feedback to be stored. This method takes apart the given request and stores the feedback information in the Database using the Database Interface. See use case 3.2.4.7.

3.5.3.7. Update Song Requests

The *updateSongRequests()* method will take a Song and increment its request count by one in the Database using the Database Interface. See use case 3.2.4.6.

3.5.3.8. Get Playlist and Get Playing

A request for the playlist and for the currently playing list of songs is identical apart from the data in the messages. Since this is the case, both will be described the same way here. A message is received from the client requesting one of the lists of songs (see use cases 3.2.4.11 and 3.2.4.10). The requested list is taken from the MusicManager by calling either the *getPlaylist()* or *getPlaying()* method (use cases 3.2.4.11 and 3.2.4.10 respectively). The result is formatted into a reply Json object. This reply is then sent back to the client and the connection and ProcessingThread are terminated when it is received (use case 3.2.4.4 and 3.2.4.5).

3.5.3.9. Run

The ProcessingThread implements the Runnable interface and overrides the *run()* method of that interface. In this method, all necessary variables are initialized that were not initialized in the constructor. After initialization is done, the ProcessingThread will take the request from the *mSocket* and use the *parseRequest()* method to execute the required action. If the *parseRequest()* method gives a non-null response, this response is returned to the Client, again via *mSocket*. The ProcessingThread must check the status of the client connection before proceeding with execution in case of timeout. If the client connection has timed out, it will disconnect and terminate. Once a response has been sent, the ProcessingThread disconnects by closing *mSocket* and terminates.

3.5.4. CRC

The following tables show the CRC cards for the RequestHandler, HandlerThread, and ProcessingThread classes. On the left side of the table is listed the major responsibilities of each class. On the right hand side of the table is listed the collaborating classes or modules for the class.

Responsibilities	**RequestHandler**	Collaborators
Receive Connections		Client
Queue Connections		Handler Thread
		Processing Thread

Responsibilities	**HandlerThread**	Collaborators
Start Request Processing Threads		Request Handler
Manage Processing Threads		Processing Thread
Dequeue Connections		

Responsibilities	**ProcessingThread**	Collaborators
Execute Requests*		Handler Thread
Return Results to Clients		Client
		Music Manager
		Database Interface
		Song

*Execute Requests consists of the following possibilities (see section3.4.2 for more details):

- Send Feedback
- Request Song
- Update Playlist
- Update Playing Songs

3.5.1 Implementation Constraints

The table below lists some methods in the request handler module that have some necessary constraints as well as an explanation of these constraints.

Method	Constraint	Reason
shutdown() HandlerThread.run()	Lock the mShutdown variable.	If the mShutdown variable is not locked, it is possible that

RequestHandler.run()		attempting to shut down the RequestHandler will fail due to two threads accessing the variable at once.
HandlerThread.run()	Limit the thread pool for ProcessingThreads.	If too many threads are allowed to run, the CPU will thrash and system performance will decrease.
startConnection()	Start a new ProcessingThread for each connection.	For faster response time at each Client, a Client will interact with its own ProcessingThread for the duration of the request-reply communication.
HandlerThread.run() RequestHandler.run() startConnection()	Lock the mConnectionQueue.	Further described in section 3.6.

3.6 Critical Section Specification

The critical section of this module relates to the queue of waiting client connections. Two separate threads will be adding and removing connections from this queue and they must be synchronized. If incorrectly implemented, connections could be lost, requests could be processed repeatedly, and the queue itself could end up in an indeterminate state resulting in further incorrect execution.

The solution to this problem is to ensure that only one thread adds or removes connections from the connection queue at once. Using Java-like pseudocode, below is the presentation of the solution for these sections of the RequestHandler and HandlerThread code.

Request Handler:
Data Structures:

```
Socket clientConnection
ServerSocket mServerSocket
List<Socket> mConnectionQueue
```

Code:

```
while (true)
        clientConnection = mServerSocket.accept()
        lock(mConnectionQueue)
        mConnectionQueue.addLast(clientConnection)
        unlock(mConnectionQueue)
endWhile
```

Handler Thread:
Data Structures:

```
List<Socket> mConnectionQueue
Integer queueSize
Integer SLEEPTIME
```

```
MusicManager mMusicManager
```

Code:

```
while (true)
        lock(mConnectionQueue)
        queueSize = mConnectionQueue.size()
        unlock(mConnectionQueue)

        if (queueSize > 0)
                startConnection();
        else
                sleep(SLEEPTIME);
endWhile
```

Method - startConnection:

```
freeThread = -1
i = 0

while (freeThread < 0)
        if(!mThreads[i].isAlive() && mThreads[i] != null)
                        mThreads[i] == null

        if(mThreads[i] == null && freeThread == -1)
                freeThread = i

        i = ( i + 1 ) % THREADPOOL
endWhile

lock(mConnectionQueue)
mThreads[freeThread] = new
ProcessingThread(mConnectionQueue.getFirst(), mMusicManager)
mThreads[freeThread].start()
unlock(mConnectionQueue)
```

This solves the problem by restricting access to the queue of connections to one thread at a time. The other thread would have to wait until the other thread is finished before accessing it.

Section 4: Programming Management

4.1 Directory Structure and Programming Tools

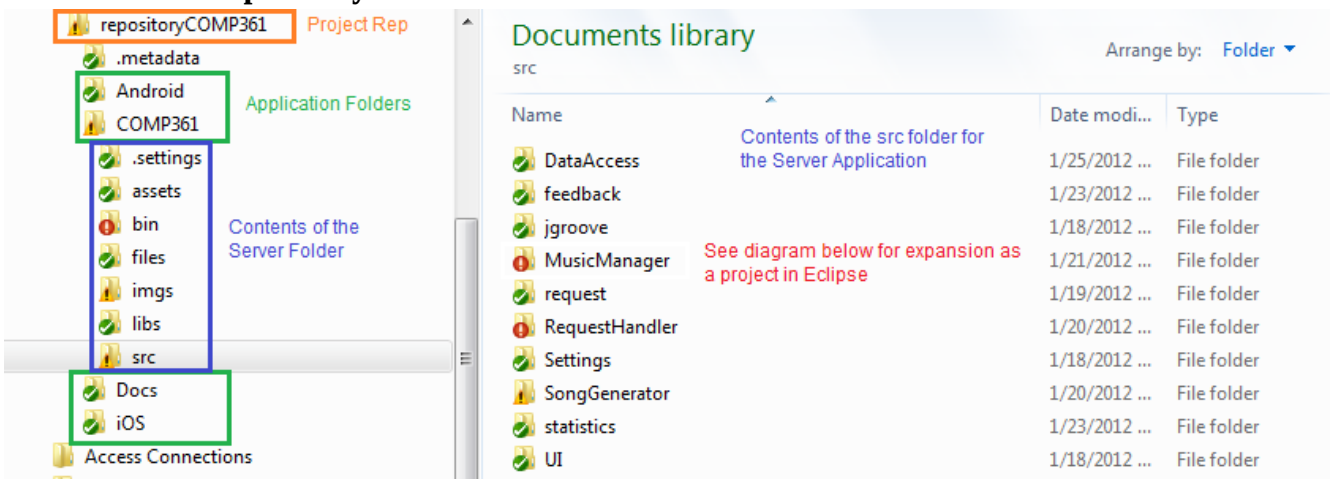
We will be using Eclipse for coding and testing, and SVN for team management and code revision. See below under Coding Agreement for more information on SVN usage and our choice of SVN interface. The server application will be developed in Java for access to libraries and cross-platform compatibility, while the client applications will be developed using the Android SDK and Objective C.

Our shared repository contains four main folders, three for the separate applications – server, iOS, and Android – and one for the documents and all related information.

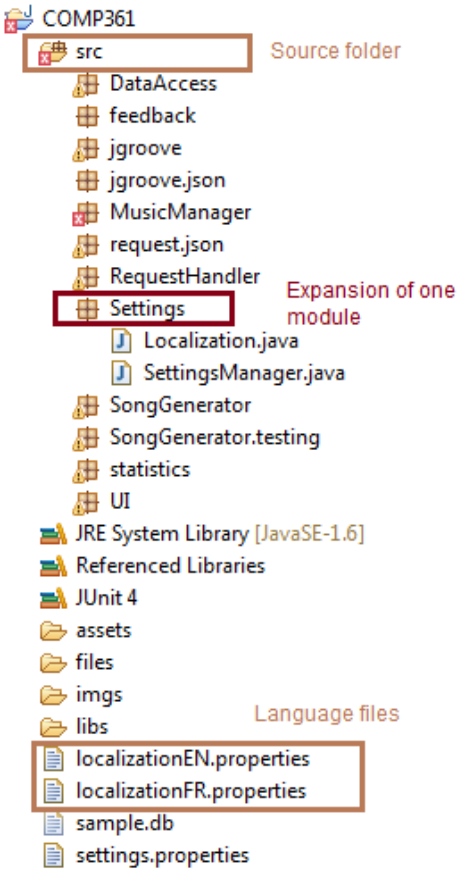
1. The server folder has a standard Eclipse project organization, with three main folders: the bin, the src folder containing all the modules as packages, and libs, containing all the files on the project build path. The necessary database files are also kept here.
2. The Android folder contains the same above folders following the standard set-up for the Android SDK/Google.
3. The iOS folder is organized as a workspace for development for iOS in Objective-C.
4. The documents folder contains a sub-folder for each document. Each of these sub-folders is divided into relevant categories: UI, diagrams, drafts, and final.

The following file tree images illustrate the organization described above.

4.1.1 General Repository Structure



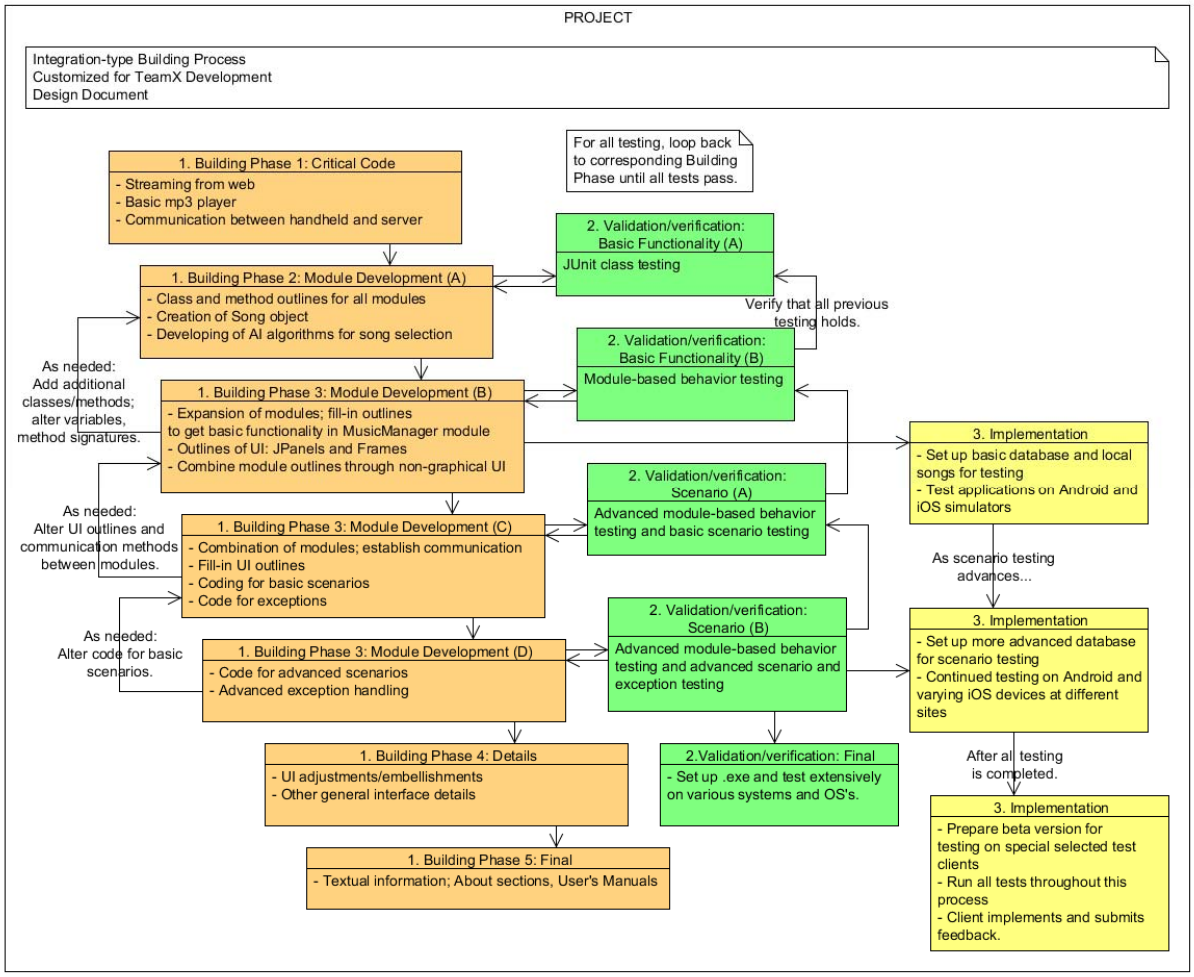
4.1.2 Structure of a Module as a Project in Eclipse



4.2 Software Building Method

4.2.1 The Method

The software building method is a pseudo-integration. Within the building process, looping back to previous phases is permitted, in particular for testing at all stages, responding to client feedback. The building process as customized for our project is illustrated in detail in the following diagram.



A few key points to notice about our building approach: the building phases are carried out by module, with highly structured sequential development. Also, the validation/verification phase is closely linked with the building phase; it is considered a part of it.

4.2.2 The division of work

Our project is divided into separate modules. We have divided the work according to these modules, their main sub-classes, and other independent tasks:

1. Lineker: Streaming Manager, Database, iOS Module.
2. Rebecca: Music Manager, Main Frame GUI.
3. Alicia: Song Generator, Statistics, Request Handler, Android Module.

For each module, the member is responsible for the relevant GUI including normal dialog, error handling, and input/output; testing of all units and related scenarios; and providing of functionalities that other modules will need.

4.3 Revision History

4.3.1 SVN General Usage

Since we are using SVN, we will commit per defined unit of work, making it possible to view revision changes in the formal format as produced by SVN. We have defined a unit of work to be the addition of a functionality (one or more methods or classes), a testing unit, a bug fix, or minor changes such as comment and code clean-up. To keep our team development efficient, each team member must insure that at his/her commit, the project components still compile. For changes that affect other team members' modules, the member making the commit is responsible for informing them and discussing the change if necessary.

4.3.2 Javadoc

Our use of the standard Javadoc commenting for classes, constructors, method headers, internally, and test units will facilitate the revision process. The specific elements of Javadoc commenting in classes we will be implementing is illustrated below:

```
/**
  * Description of the class and any other important information related to
functionality.
  * @author the author's name
  */

/* Description of a class variable – its purpose */
private ActiveQueue mQueue;

/**
  * Description of the method or if a constructor, "Constructor for __.".
  * @param x description and type of the argument
  * @return boolean the type of the return value and its meaning
  */
public boolean methodName(int x) {
      // Minor comments explaining the code implementation within the method.
      // Used only when the code could be ambiguous or complex.
}
```

Methods in test units will begin with @Test, @Before, @After as needed.

4.4 Coding Agreement

4.4.1 Style

To maintain a uniform, clean coding style throughout the project, we will define and impose

a Check-style on the Java portion of the project. The programming done for the iPhone and Android applications will also follow the same styles when applicable. The major highlights of these rules are outlined below:

	Title	Style
1	Indentation	Indentation will be 3 spaces, and used for every new class and nesting.
2	Commenting	Standard Javadoc rules will be applied. See section 4.3.2 above for a defining of which ones in particular we will adhere to.
3	Spacing	A space will be inserted between elements of calculations except between an element and a parenthesis.
4	Variable Nomenclature	Variable names will begin in lower case with each new word beginning with an upper case.
5	Class Nomenclature	Classes are upper case with each new word beginning in upper case.
6	Lines	Lines of code should not exceed 100 characters. Lines longer than this will be split and the second half indented on the new line by one additional tab.
7	Method Nomenclature	Methods begin in lower case with each consecutive word beginning in upper case.
8	Getters and Setters	Getters and setters are named getVarName and setVarName except in the case of type Boolean: isVarName and setVarName. We will use getters and setters for encapsulation when the class variables need to be accessed or modified. In the case that the variables should not be accessed externally, we will not use getters and/or setters.
9	Class Fields	Class members begin with a lowercase m in front of the name, for example: mQueue for the ActiveQueue which is a member of the MusicManager. Note that local variables within class methods are not class members.
10	Final and Global Variables	All global variables and class variables of type “final” will be named in all CAPS.

4.4.2 Team Code Procedures

As mentioned above, we will be using an SVN repository for managing the project code and documents. Team members with a Windows platform will use the Windows shell extension

TortoiseSVN as our source control software. Whenever possible, to maintain a single tip version of the code at all times, the following rules will be followed:

1. A team member must first do an updated from the shared repository before committing local changes to code.
2. The team member committing local changes is responsible for making any necessary merges to the pulled code before submitting changes to the repository, except when there is valid reason to wait.
3. Every change to shared code must be accompanied by a brief comment detailing what was changed and why.
4. Each team member is responsible for testing his/her own classes for unit testing. Acceptance based testing will be the responsibility of the member who oversees that particular module.

4.5 Migration Procedures

We foresee only simple migration as being necessary in the future, for version upgrades. The first upgrade after the beta release will be a language upgrade (refer to Section XXX in the Requirements Document for more details). This migration will involve the same testing procedures outlined in Section V as for the initial development, as well as a thorough verification of the UI by team members and external verifiers to ensure quality and correctness. After this testing the release can be made, and clients (if any) using the beta version will be advised as to downloading the upgrade. The new software functionality and memory requirements will be identical, so database migration will not be necessary.

Another upgrade, perhaps included in the language upgrade version, will include expanded support for different sound files. Currently supported are .mp3 and .WAV; future formats could include .aac and other more semi-commonly occurring formats. Other potential upgrades include support for different mobile devices.

A possible, but not planned version upgrade following the language one could be a UI upgrade.

Complex migration will not be necessary since our choice of technology ensures compatibility with the three major operating systems we are covering. We have no plans at present to migrate the software to a different programming language in the future.

4.6 Installation Procedures

The iOS and Android applications will be made available for download in the respective App stores; download will follow the normal procedures: the user with the device will access his/her App store, search for the application, and select download. The application will then be installed automatically on the device.

The server application will be an executable file which must be placed in an a user-chosen location. The initial set-up should be less than 20 seconds; the program will simply set up or locate the necessary folders and initialize the database.

4.7 Training Guidelines

As per our project goals referred to in the Preliminary Design Report, we hope to develop a software that is intuitive to all new users, regardless of technological background.

For the server application, since clients may, in most cases except possibly for beta release, download the application files on their on or from a CD, there will be no on-site training. Rather we will include with the files the installation instructions - textual user's manual accessible under the Help menu item of the server application – as well as an installation video which will clearly illustrate the set-up.

Given the simplicity of the iOS and Android applications, we will not have a user's manual, but instead will rely on our intuitive design. However, all the applications will include a short "About" section providing information about the developing team, copyrights, and contact information in case of questions or comments.

At the end of development, close to the beta release date, we will have a website with information including an About section, screenshots, FAQ, and contact information. It will also have links to the application download pages and a list of the service providers currently using our software.

Section 5: Testing

5.1. Procedures

5.1.1. Testing Scope and Depth

5.1.1.1. Testing Scope

5.1.1.1.1. Components to Test

Below is listed the difference components of the system with indication of whether it will be included in testing or not.

Component	Testing Scope
Android Device Module	This module will be tested. This module is required for client interaction with the rest of the system and must therefore have in depth black box testing. Other forms of testing will be required to ensure communication times with the server are acceptable.
Database Module	This module will be tested. All data stored for the application will be accessed and modified through this module and the module must carry out all actions correctly to maintain data integrity and validity.
Environment (OS, Network)	This will not be tested. The environment where the project is run is outside of the scope of the project and thus will not be tested.
External Libraries	These will not be tested. We will assume that the developers of the libraries have tested their code in their development of it.
Hardware	This will not be tested. The project is not hardware dependent and so the hardware will not be tested.
iOS Device Module	This module will be tested. This module is required for client interaction with the rest of the system and must therefore have in depth black box testing. Other forms of testing will be required to ensure communication times with the server are acceptable.
Java Virtual Machine	This will not be tested. We will assume that the JVM has been tested in its own development, which is outside of the project scope.
Music Manager Module	This module will be tested. This is the core module of the system and will require extensive testing to ensure that it can handle errors caused by its own components as well as errors caused by other components.
Request Handler Module	This module will be tested. This module is required for all communication over the network. Speed and correctness are critical here so they must be tested.
Song Generation Module	This module will be tested. This module is less critical and will have key features well tested but does not require extensive tests. This module must still be tested as it has some required functionality used by the Music Manager module.
Statistics Module	This module will be tested. This module is less critical and has no other modules depending on it. Only the client interacts with this

	module so testing will focus on response times and usability.
Streaming Module	This module will be tested. Testing on this module is primarily necessary for integration of this module with the other components of the system. Other modules must be able to tolerate failures from this module and integration testing will need to account for this.
Streaming Service	This will not be tested. The streaming service provides its API which we will assume has been tested in its development.
User Interface	This module will be tested. This module is critical for user interaction and usability. The focus of testing on this module will be to ensure a pleasant user experience.

5.1.1.1.2. Types of Tests

Below is a list of which tests will be used for testing of this project.

Type of Testing	Modules to be Tested
Unit Testing	Music Manager, Song Generator, Statistics, Streaming, Database, iOS Device
System Testing	Music Manager, Android Device, Request Handler, Streaming, Database, iOS Device
Functional Testing	Android Device, Song Generator, Statistics, Request Handler, Streaming, Database, iOS Device
Regression Testing	Music Manager, Android Device, Request Handler, Streaming, Database, iOS Device
User-Interface and Feature Testing*	User Interface, Android Device, Song Generator, Statistics, Streaming, iOS Device
Integration Testing	Music Manager, Request Handler, Streaming, Database, iOS Device

*Note: User-Interface and Feature Testing will be done based on use cases from the Requirements Document

5.1.1.2. Module Testing Breakdown

5.1.1.2.1. Android Device Module

Type of Testing	Explanation
System Testing	System testing is required to ensure that the Android device is correctly communicating with the server component of the system. This will require black box testing for all valid messages.
Functional Testing	Functional testing is required to ensure that the Android device software behaves correctly upon receiving particular messages. This testing will require black box testing to ensure correct behaviour from the user point of view.
Regression Testing	Regression testing will require that the system and functional tests be repeated only if the Android module or Request Handler module is changed. This will require a repetition of all the testing for this module.
User-interface and Feature Testing	User-interface and feature testing will be required for the interface of the device. This will be done with black box testing to ensure correct behaviour from the user point of

view.

5.1.1.2.2. Database

Type of Testing	Explanation
Unit Testing	Unit testing will be used to test the connectivity of this module with the SQL database and the ability to run queries. This will be black box testing since the interface is the critical component of this module.
Functional Testing	Functional testing is needed for this module to ensure that all components can store and access data from the database. This will be black box testing.
Regression Testing	Regression testing is required if any changes are made to the database or this module. This will be re-running all other tests for this module.
Integration Testing	Integration testing is required since this module needs to interact with several other components of the system. Integration testing will be black box to test that other modules get expected results from the database.
System Testing	System testing is required since several parts of the system are dependent on the Database. This will be done as black box testing from the UI to ensure correctness from the user point of view.

5.1.1.2.3. iOS Device Module

Type of Testing	Explanation
Unit Testing	Unit testing will be used for the iOS device to ensure core functionality. This will be black box testing since there are no critical components.
Functional Testing	Functional testing is needed to check that the module can carry out all necessary operations. This will be done with black box testing.
Regression Testing	Regression testing will require that the system and functional tests be repeated only if the iOS module or Request Handler module is changed. This will require a repetition of all the testing for this module.
System Testing	System testing is required to ensure that the iOS device is correctly communicating with the server component of the system. This will require black box testing for all valid messages.
User-Interface and Feature Testing	User-interface and feature testing will be required for the interface of the device. This will be done with black box testing to ensure correct behaviour from the user point of view.

5.1.1.2.4. Music Manager Module

Type of Testing	Explanation
Unit Testing	Unit testing is required to make sure the basic components of the Music Manager are correct. Since this module is so large, unit testing will provide a good indication of basic functionality. Black box testing will be used to check each unit for correctness.

System Testing	As the central component of the system, full system tests will need to be done with this module. Black box testing will be used on the interfaces of all components with the Music Manager.
Regression Testing	Regression testing is necessary for any change within this module affecting functioning with other modules or within other modules that the music manager uses. This will be a repetition of all previous tests.
Integration Testing	This module integrates with almost all other components of the system. Each point of integration must be tested for functionality for both the Music Manager and the integrated component. This will use black box testing from the user point of view, both on the client and server side of the application.

5.1.1.2.5. Request Handler Module

Type of Testing	Explanation
System Testing	System testing is required to ensure that the client devices are correctly communicating with this component of the system. This will require white box testing for the critical section in 3.2 and black box testing for all valid messages from a client. In addition to this, stress testing will also be necessary.
Functional Testing	Functional testing will be required to make sure all correct and incorrect messages are handled correcting. This will require black box testing of the module.
Regression Testing	Regression testing of this module would be required if the Database or Music Manager change their interface to ensure correct execution. If this module itself is changed, regression testing will be required to ensure correct functioning of this module within the system. This will repeat the other tests.
Integration Testing	Integration testing is required to ensure that upon executing the requested action from a client, the other components of the system show the required changes, such as a requested song or new feedback. This will be done with black box testing to check that each component gets the desired result after a particular input.

5.1.1.2.6. Song Generation Module

Type of Testing	Explanation
Unit Testing	The Song Generation module has some disjoint basic methods that lend themselves to unit testing. This unit testing would need black box testing to ensure correct results of the units themselves.
Functional Testing	Functional testing is required to check that the Song Generator is meeting all the requirements in producing lists of songs. Black box testing will be used to check the interface of the Song Generator.
User-Interface and Feature Testing	The UI component of the Song Generator will require black box testing to ensure that all its features work properly from the user point of view.

5.1.1.2.7. Statistics Module

Type of Testing	Explanation
Unit Testing	All individual units of the Statistics Module should be tested to make sure they work correctly on their own. This testing will use black box testing.
Functional Testing	The Statistics Module interface must be tested to provide all the required functionality. This testing will be done as black box testing, giving every possible input (every type of chart generation request) and checking the output.
User-Interface and Feature Testing	The UI component of the Statistics Module will require black box testing to ensure that all its features work properly from the user point of view.

5.1.1.2.8. Streaming Module

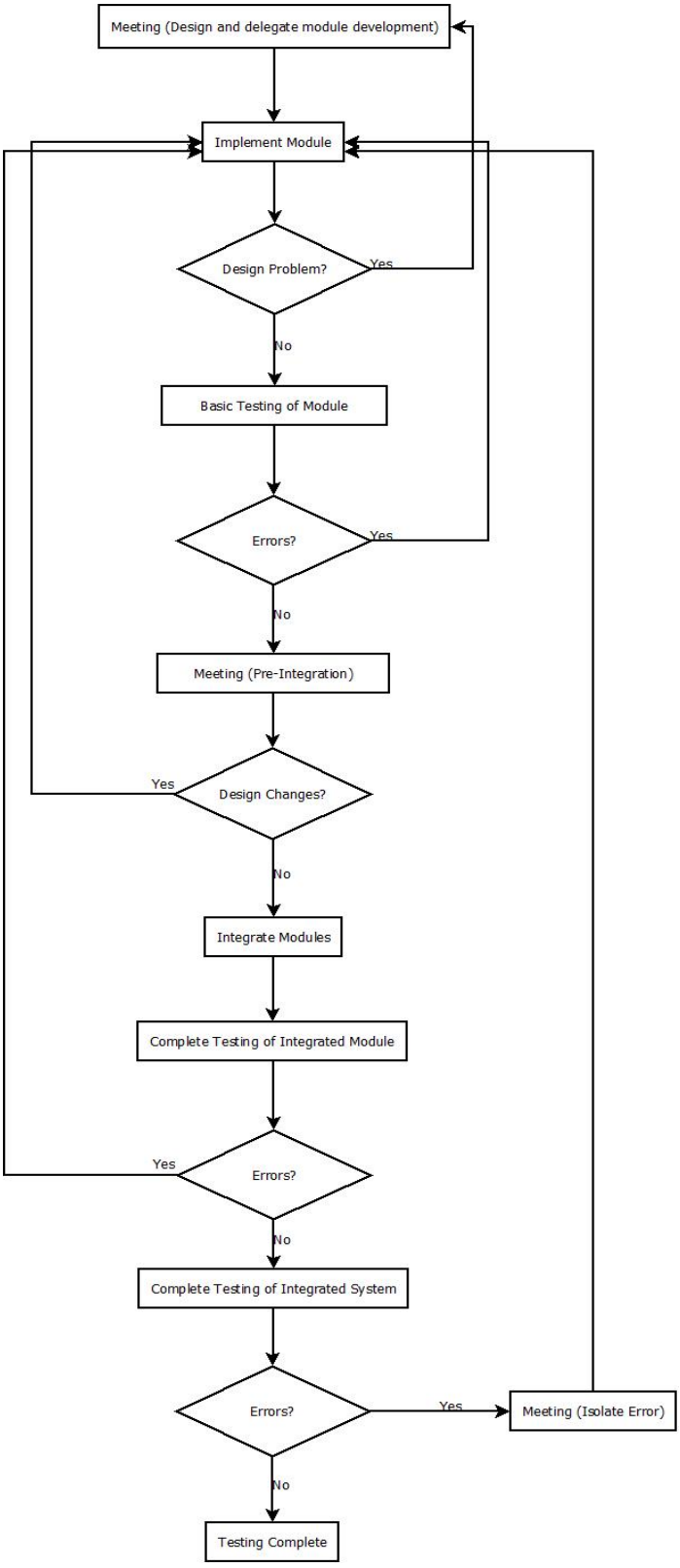
Type of Testing	Explanation
Unit Testing	Unit testing will use black box testing techniques to ensure that basic operations using retrieving streaming songs and streaming song information are correct.
Functional Testing	Functional testing is necessary to check that the streaming module meets the requirements of the project as well as correctly transferring songs over the network. This will use black box testing since there are no critical components involved in this module.
Regression Testing	Regression testing is necessary if any of the Music Manager, Database, or Streaming components are changed. This will require the repetition of previous tests.
Integration Testing	Integration testing is required to make sure that other components of the system can use the Streaming module. This will be done with black box testing with other modules checking expected input from the Streaming module.
System Testing	System testing is necessary to check functionality of Streaming music with the other components of the system, particularly the Music Manager. This will use black box testing techniques to see if the system works when streaming is used.
User-Interface and Feature Testing	UI testing is necessary to ensure a pleasant user experience. This will be done from the user perspective with black box testing of features.

5.1.1.2.9. User Interface

Type of Testing	Explanation
User-Interface and Feature Testing	Since the UI is the main point of interaction with users, it must be well tested. This testing will be done as black box testing since users will be interacting with this component without knowledge of internal workings of the system.

5.1.2. Testing Method and Agreement

5.1.2.1. Testing and Procedure Flowchart



The ideal development of the software is shown in the above diagram. Once the module design is made, a programmer is assigned a module and implements it according to design. Following this, there will be basic testing of the module on its own before meeting with team members to start integration. Once integration has been planned, the module is integrated and tested fully as part of the integrated system. After the module has been fully tested, the integrated system as a whole is tested. Testing is then complete. The cases where there are errors are described in the next section (5.1.2.2).

5.1.2.2. Programmer Agreement

Event	Actions
An error is found in a module the programmer is responsible for.	1. Programmer determines extent of error. 2. If the error affects no other modules directly, fix the error and proceed to 6. 3. If the error affects other modules, report error to other team members. 4. Upon receiving an error report, if this error affects the project design, schedule a meeting to fix the error. 5. When a fix to the error has been decided upon, the programmer(s) responsible for the affected modules will implement the changes. 6. Commit the fixes to the team repository. 7. Notify other members that the error has been fixed and changes committed.
An error is found in a module the programmer is not responsible for.	1. Programmer determines extent and context of error. 2. Report the error to team members and request that the error be fixed, specifying severity, context, and extent of the error. 3. Upon receiving an error report requesting something be fixed, the programmer responsible for the module will determine if the error is minor or not. If it is minor, proceed to 5. 4. If this error affects the project design or multiple modules, schedule a meeting to fix the error. 5. When a fix to the error has been decided upon, the programmer(s) responsible for the affected modules will implement the changes. 6. Commit the fixes to the team repository. 7. Notify other members that the error has been fixed and changes committed.

An error is found in the integrated system but the source cannot be determined or there is conflict between modules.	1. Programmer determines context of error. 2. Report error to other team members and schedule a meeting to resolve. 3. When a fix to the error has been decided upon, the programmer(s) responsible for the affected modules will implement the changes. 4. Commit the fixes to the team repository. 5. Notify other members that the error has been fixed and changes committed.
An error is found after making a fix has been made (during regression testing).	1. If this is a module the programmer is responsible for, notify team members that error has not been fixed and go to 3. 2. If this is not a module the programmer is responsible for, notify team members that this error has not been fixed. 3. Upon notification that a fix was ineffective or introduced other problems, the programmer responsible must either fix the original error or revert to a working version immediately. 4. If the error could not be fixed, schedule a meeting with team members to determine a workaround or solution. If the error could be fixed, proceed to 6. 5. When a fix to the error has been decided upon, the programmer(s) responsible for the affected modules will implement the changes. 6. Commit the fixes to the team repository. 7. Notify other members that the error has been fixed and changes committed.
An error is found in the design during implementation.	1. Notify team members that the design has an error and schedule a meeting to resolve this. 2. Once the solution is determined, programmer(s) return to implementing their modules.
An error is found in the design prior to integration of completed modules.	1. Notify team members that the design has an error causing problems with integration. 2. Meet to determine a solution to the integration/design problem. 3. Programmer(s) re-implement modules as required by solution.

5.1.2.3. Automatic Testing Procedures

The following modules will be using automatic testing: Song Generator, Statistics, Music Manager, Request Handler, iOS device, Database, and Streaming. For the modules written in Java, the JUnit testing library will be used for some automated testing as well as test programs written by the programmers of each module for the purpose of testing the modules. The iOS device module will be tested with the testing package in XCode. Each of the modules will have a single test suite to run all tests for a module.

5.2 Test Cases on Selected Class: Active Queue

5.2.1 Introduction to Selected Class

The Active Queue thread is a crucial part of the Music Manager module. It processes requests one at a time through a watcher thread, retrieving them in the order they were received by the Music Manager, and attempts to add them to the active list of songs to be played. While doing request handling, it also monitors the active list and unless told to stop or pause, spawns the Music Player which plays the song at the head of the queue. The Active Queue waits for the song to terminate, then removes that song from the queue and spawns the player with the next song. It also verifies at each song that the necessary requirements are met (the list length is long enough, the song can be played, etc.)

Testing of this class is crucial since the primary activity of playing music and adding requested songs depends directly upon it.

5.2.2 Testing by Type

5.2.2.1 Unit and Functional Testing

Automated testing of each method will be done using Java's jUnit4. Note that features involving these methods will be done black box method as well. Given the following member variables of the class, below are the white and black box testing cases.

Queue: the queue of songs actively playing
MIN : the integer value indicating the minimum number of songs that must be on the Queue at all times.

Method name	addSong(Request r)
Input/ Parameter value range	Any Request object may be received. Request objects consist of a Song and an optional specified time in hours and minutes. Songs consists of minutes and seconds.
Return value range	Boolean indicating whether the song was inserted or not.

Complete input/param validation rules (boundary conditions)	The object must be a completely instantiated Request consisting of a non-null Song, and integer hour and minute fields. There are no constraints on the integer values; the presence of a negative hour and/or minute value will be handled as "no time requested", and hour values > 24 and minutes > 60 will be treated if they were valid times (granted that time requests can be respected still depends upon the play list). A null Request object will be ignored and the method exited; nothing will be added to the queue and no fields will be changed. An incorrectly instantiated Request with a null song will also be ignored.
Flow paths and tests	Let T=0 be the current time, and Tf be the latest time for which there is a song scheduled to be played (on the queue). Let Tr be the requested time of the Request r if specified. There are thus four possible cases to test and their expected results: Case 1: Tr != null, Tf > Tr. Expected output: A non-requested song Si at some time Ti < Tf has been replaced by r.song, which is inserted within +-3 minutes of the requested time Tr; OR, if the time constraint cannot be met and/or all the songs playing at time T < Tf are requested, r.song will be inserted at Tf. Case 2: Tr != null, Tf < Tr. Expected output: the r.song should be inserted within +-3 minutes of the requested time Tr. The time between Tf and Tr should be filled with other valid songs. If the contents of the current playlist do not allow the time constraint to be met, r.song will be inserted within the smallest feasible range of Tr. Case 3: Tr == null. Expected output: r.song will be exchanged with a song at T < Tf if this non-requested song exists between [0, Tf] that satisfies the +-3 minutes length constraint. Otherwise, r.song will be added to the end of the list at Tf. Case 4: the queue has reached its maximum memory limit. Expected output: the r.song has not been inserted in the queue. (There is no error message and the request is essentially ignored since this is considered an uncommon case, and once songs are played they are removed from the queue so the problem will self-resolve.)
Auto testing or manual	Automated testing will be done by the jUnit tests formatted as described below to cover the three flow paths (and their branches), the parameter boundary values, and the parameter extreme values (negative and large): Let us call the request time specified by r.min and r.hour to be Tr.

Initial Set-up:
Select a playlist with length >= 20.
Terminate the watcher so that no external requests will be processed during testing.
Insert 6 non-requested songs into the queue, Q.
Make a clone of the queue contents, Q'.

Boundary input ranges; black box:
addSong(null): assertEqual(Q,Q').
addSong(Request r) with r.song = null, r.min = n, and r.hour = m where (n, m) are integers: assertEqual(Q, Q').
addSong(Request r) with r.song != null, r.min < 0, r.hour >= 0: assertEqual(Q, Q');
addSong(Request r) with r.song != null, r.min >= 0, r.hour < 0: assertEqual(Q, Q');

Case 1:
Record current time Ti, and the sum S of the 6 inserted songs.
Store Tf = Ti + S.
addSong(Request r) with r.song != null and Tr < Tf.
assertNotEqual(Q, Q').
assert(Q.length >= 7).
A) Retrieve the time Tx that r.song will be played.
assert(Tx < Tf).
assert((Tr – 3) <= Tx <= (Tr + 3)).
B) assert((Tx = Tf) || (Tx < Tf)).
assertTrue(A || B): one of the subpaths must have been executed.

Case 2:
Record current time Ti, and the sum S of the 6 inserted songs.
Store Tf = Ti + S.
addSong(Request r) with r.song != null and Tr > Tf.
assertNotEqual(Q, Q').
assert(Q.length >= 7).
Retrieve the time Tx that r.song will be played.
assert(Tx > Tf).
assert((Tr – 3) <= Tx <= (Tr + 3)). Note that this could fail understandably if the play list contents do not allow the constraints to be met; this is acceptable.

Case 3:
Record current time Ti, and the sum S of the 6 inserted songs.
Store Tf = Ti + S.

	addSong(Request r) with r.song != null and (r.min || r.hour) < 0. assertTrue(Q.length >= 7). assertNotEqual(Q, Q'). Case 4. Create a boolean variable v and insert it inside the condition checking that there is memory available to be allocated for the LinkedList implementation of the queue. Set v = false. addSong(Request r) with r.song != null and (r.min && r.hour) >= 0. assertEqual(Q, Q'). Note: this simulates the scenario of having a memory allocation problem without needing an enormous number of songs available in the playlist. Since song lengths vary and the functionality depends on the play list contents, manual testing through the Music Manager should also be done.

Method name	run()
Input/ Parameter value range	No input; however, the Music Player, the queue of songs, and the MIN variable are accessed.
Return value range	No return values.
Complete input/param validation rules	No input values; but see below for the possible states of the global variables this method accesses.
Flow paths and tests	There are several cases depending on the state of the queue and the current play list. Since this method contains a do/while loop, cases are formed based on the state at each iteration of the loop. (Before the loop begins there is only a single flow path which only contains variable initializing and minor set-up; testing the loop cases is sufficient for testing this path.) Case 1: |Q| >= MIN: Expected behavior: the player will be spawned and start playing the first song on the queue. It will then wait for the song to terminate, then remove that song from the Q and loop back. Case 2: |Q| < MIN: Expected behavior: assuming that this is the first iteration that this case holds true, there still are MIN number of songs on

	Q. So the player will be spawned and will play the MINth song. After the spawn, the Q will attempt to generate songs by calling generateSongs() and add three songs to the Q retrieved from the play list by the Request Handler. Here there are two sub paths: A) The three songs were retrieved and added to the Q. B) Due to play list length and/or contents, it was not possible to add three songs. Some number 0 <= N < 3 songs were added instead. In both sub paths, the method will wait for the termination of the player thread, remove it from the Q, and loop back. This will lead to Case 3. Case 3: |Q| == 0, no valid songs in playlist. Expected behavior: checking for a song on the queue will return a null, so the thread will exit and return, with an error message generated explaining the cause, that it was not possible to add songs to the queue.
Auto testing or manual	These three cases and the sub paths will be auto tested using jUnit in the following manner. Note that boundary cases are included in these tests under the sub path tests. Initial Set-up: Select a playlist with length >= 20. Terminate the watcher so that no external requests will be processed during testing and addSong() will not be called. Case 1: Insert MIN + 1 non-requested songs into the queue, Q. Store a copy of the first song inserted, s1, and the second song inserted, s2. Make a clone of the queue contents, Q'. Call run(). assertTrue(ActiveQueue.player.isPlaying). Wait for the song to terminate. assertTrue(Q.getFirst() == s1). assertTrue(Q.length == MIN). assertNotEqual(Q.getFirst(), s1). assertEqual(Q.getFirst(), s2). Case 2: Insert one valid song s1 in the Q. Set the playlist p1 with >= 3 valid songs. Call run(). assert(Q.length == 3).

	assertTrue(ActiveQueue.player.isPlaying). assertTrue(Q.getFirst() == s1). Insert one valid song s1 in the Q. Set the playlist p1 to < 3 valid songs. Call run(). assert(Q.getFirst() == s1). assert(Q.length = 2). assert(ActiveQueue.player.isPlaying). Wait for the two songs to terminate. assert(!ActiveQueue.player.isPlaying). assert(!ActiveQueue.isActivated). <u>Case 3:</u> Remove all songs from the Q, if any. Remove all valid playable songs from the play list (or make it empty). Call run(). assert(!ActiveQueue.player.isPlaying). assert(!ActiveQueue.isActivated). Again due to the varying nature of the songs and play list contents, manual testing also will be performed for each of these cases.

Feature name	generateSongs() adds three songs requested from the Request Handler and adds them to the end of the current queue.
Program location	A method of the ActiveQueue.
Input range	No input, so no boundary conditions.
Output range	No output.
Stress testing and expected results	<u>Case 1: MIN or more valid playable songs exist in the play list.</u> Expected behavior: after execution, the Active Queue's list will have three more songs added to the end. <u>Case 2: Some 0 <= N < MIN valid playable songs exist.</u> Expected behavior: N songs will be added to the Active Queue.
Auto testing	This method will be tested purely automatically using jUnit for the two cases identified above. This is a black box test.

	Initial Set-up: Ensure that the queue Q is empty. Case 1: Create a playlist containing > MIN valid playable songs. Set it as the current playlist. assertTrue(Q.length == 0). generateSongs(). assertTrue(Q.length == MIN). Case 2: Create a playlist containing MIN valid playable songs. assertTrue(Q.length == 0). generateSongs(). assertTrue(Q.length == MIN). Create a playlist containing 0 valid playable songs. assertTrue(Q.length == 0). generateSongs(). assertTrue(Q.length == 0). Create a playlist containing 0 < N < MIN valid playable songs. assertTrue(Q.length == 0). generateSongs(). assertTrue(Q.length == MIN). Boundary/stress case: Create a playlist containing n > 100 valid playable songs. assertTrue(Q.length == 0). generateSongs(). assertTrue(Q.length == MIN). Note that the boundary values in Case 2 are also covered here.

Method name	startWatcher()
Input/ Parameter value range	No input; but the Request stack of the Music Manager through its method getRequest() is accessed and the local method (see test case above) addSong(Request r) called. Note that getRequest() will be tested for correctness as part of the Music Manager.

Return value range	No return.
Complete input/param validation rules	No input. See the possible accessed variable values below in the flow paths.
Flow paths and tests	The watcher thread spawned at the construction of the Active Queue is an anonymous thread that continuously attempts to get a Request r from the Music Manager as long as the boolean mWatcher is true. There are two cases within this loop: Case 1: r == null, mWatcher == true. Expected behavior: nothing will happen and it loops back. Case 2: r != null, mWatcher == true. Expected behavior: the watcher will call addSong(r) and wait for it to complete; this is executed synchronously. Once the method returns, then the watcher loops back. Case 3: mWatcher == false. Expected behavior: either case 1 or 2 will occur. Then, instead of looping back, the watcher thread will be terminated.
Auto testing or manual	This will be tested using jUnit. Because it uses addSong(), that should be tested before this method. Initial set-up: Ensure that the Music Manager's request stack is empty. Do not allow external Requests to be received by the Music Manager (could be done by disabling addRequest() or setting the RequestHandler to null) during the testing. Set mWatcher = false and do not call startWatcher(). Insert n > MIN number of songs into the queue. Case 1: Set mWatcher = true. Take a copy of current queue, store as Q'. Call startWatcher(). assertEqual(Q, Q'). Note: if the system fails or freezes for some reason out of our control, the current playing song may terminate and will be removed from the queue. In this case, this test would fail. To get a real result make sure the current song has not terminated yet. Case 2: Insert a Request r onto the request stack.

	Set mWatcher = true. Take a copy of current queue, store as Q'. Call startWatcher(). assertNotEqual(Q, Q'). assert(Q.contains(r.song)). <u>Case 3 :</u> Insert a Request r onto the request stack. Take a copy of current queue, store as Q'. Call startWatcher(). assertEqual(Q, Q'). assert(!Q.contains(r.song)). Insert Request r1 and r2 onto the request stack. Take a copy of current queue, store as Q'. Set mWatcher = true. Call startWatcher(). assertNotEqual(Q, Q'). assert(Q.contains(r1.song)). Set mWatcher = false. assert(!Q.contains(r2.song)).
Stress testing and expected results	For stress testing, run the testing for Case 2 above repeatedly, in the following two ways: 1. Intially insert 50 Requests onto the request stack before proceeding. 2. Initially insert one Request but use the MusicManager's addRequest method to add additional requests at short intervals until 50 Requests have been added. Note: since the watcher thread only handles one request at a time, this is more a test of endurance than stress. For this reason, having 50 Requests is sufficient for stress testing. Expected results should be the same as in the regular Case 2.

<u>5.2.2.2 System Testing</u>

Each external class has its own local tests, so the test cases here test for the Active Queue's handling of the external methods and their possible results.

The Music Player

The Active Queue and the Music Player interact as follows: the Active Queue creates a

new Music Player at construction and stores it as a class variable. The player does nothing until given a Song, which occurs only at run(). The behavior of the player will be tested to show that errors are handled and will not affect the Active Queue.

Automated system test	Test1 : mPlayer.play() within run() Set the Active Queue to contain n > MIN songs. Add the last Song s such that s.src is an invalid File reference; ie, "Bob". Call run(). Force the tester to wait 5 seconds - enough time for the thread to iterate once through the while loop: wait(X) (See above section 5.2.2.1 under run() for description of internal implementation.) assertFalse(Q.contains(s)) assertFalse(Q.getFirst() == s) assertTrue(Q.isPlaying) Note: This is an automated test but the user must adjust the value X based on the processing speed so that the condition given above holds true. Once this value is determined further runs of this test are fully automated.

The Music Manager

The Active Queue and the Music Manager interact in the startWatcher() method where the getRequest() method is called. getRequest() may return either null or a Request (see Section 5.2.2.1 for the test cases for both these scenarios). For this particular method, the test cases given are sufficient to ensure the system communication is correct.

The Song Generator

The Active Queue and the Song Generator interact in the fillGap(int n) method where the Song Generator is asked for a Collection of songs of total length close to n. This algorithm is tested within the Song Generator for correctness, and it is understood that the feasibility depends on the playlist and the request. If this returns null, nothing should be added to the queue, otherwise the non-null Collection of songs should be added. For the purposes of the Active Queue, there is no testing to see if the Collection length is close to n.

Test 2: getCollection(int m) within fillGap(int n)
This method is encapsulated within a local method fillGap to allow for error checking outside of the addSong method, thus the test of fillGap ensures that possible malfunctioning of the

external getCollection(int n) is handled.

Method name	fillGap(int n)
Input/ Parameter value range	Integer value n.
Return value range	A Collection of Song. This could contain one or more Songs or be null. The null return is checked for in the addSong() method and handled in the test cases in Section 5.2.2.1.
Complete input/param validation rules	The parameter n must be strictly greater than zero. This is checked; if false, a null Collection is called and getCollection is not called.
Flow paths and tests	Note that this method encapsulates the external getCollection(int m) method. Due to playlist contents, a Collection of Song of cumulative length close to m may not be feasible. As a result we cannot test for anything except correct error handling. Case 1: n <= 0 Case 2: n > 0
Auto testing or manual	Case 1: Collection<Song> x = getCollection(0) assertTrue(x == null) Collection<Song> y = getCollection(-10) assertTrue(x == null) Case 2: Collection<Song> z = getCollection(20) assertTrue(z != null)
Stress testing and expected results	Collection<Song> z' = getCollection(360) assertTrue(z != null)

Section 6: Appendix

6.1 UI Adjustment

There is one change to the UI design from the Requirements Document. Instead of having a separate dialog box (see Requirement Document 3.2.2.1 *Screen Layouts 1: Music Player menu options)* to select songs to add to the playlist, clicking the "Add Song" option will be replaced by a list view in the main frame; in section *3.2.2.1 Screen Layouts: 1 – Queue View* p.21, the area marked by #10 will display the list of songs in the database. Multiple songs can be selected at once, and a right-click will allow the user to add the selections to the playlist.