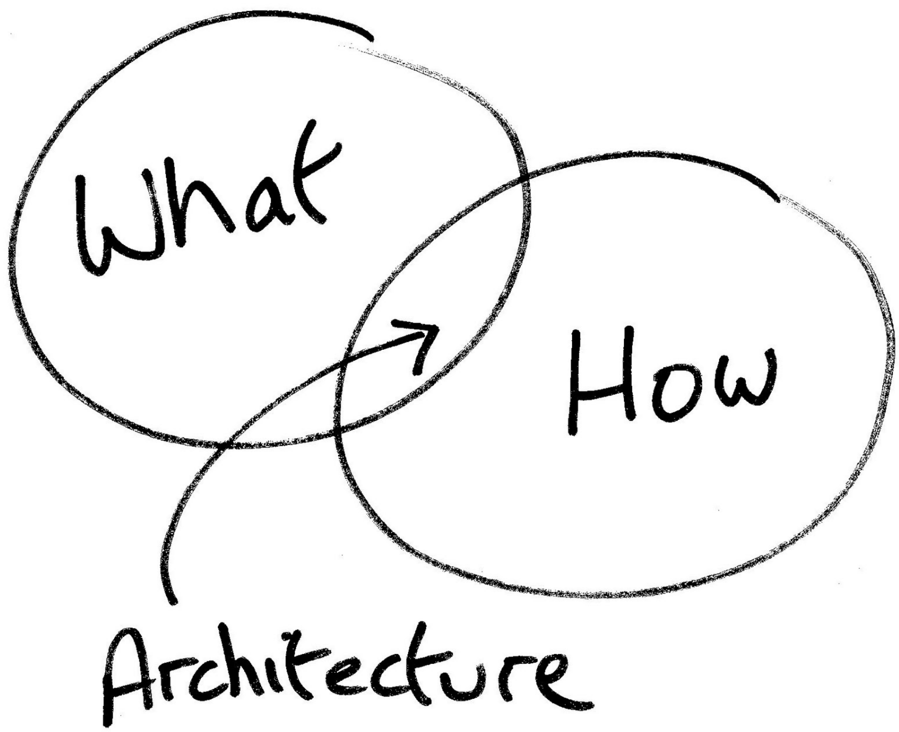
Emergent Architecture Design

Felix van Doorn favandoorn 4299566
Davey Struijk dstruijk 4289080
Roy Graafmans rgraafmans 4299442
Emiel Rietdijk earietdijk 4205383
Millen van Osch mjevanosch 4299426

June 2015



Abstract

This document specifies the architecture of this application and different design choices. Some of these choices include the implemented design patters, the Hardware/Software mapping, Persistent data management and concurrency within our application. We will also attempt to explain the design patters we have used and motivate why we used them.

Contents

1 Introduction

For this project, we are attempting to create complex software in a rather short period of time. In order to prevent bugs and to improve the quality of our product, we of course need to have a good code. After all good implementation of a bad design still leaves you with useless software. This document will also serve as reference guide to third parties interested in our development process and those who are interested in our thought process after inspecting our source code itself.

1.1 Design Goals

1.1.1 Code Quality

The way in which our group works with Git attempts to improve code quality and guarantee that there is always a working version available. We maintain two different versions; A working version on the master branch and a developing version on the develop branch. Changes are first merged onto the develop branch during a sprint. Only once the develop branch is fully functional will we make these changes to the master branch. Pull-based development is a very important factor in ensuring the code quality of our product because of the code reviewing.

1.1.2 Component Independence

As is good practice in Object-Oriented Systems, we would like to achieve Component Independence. For that reason we chose to structure our software using the Model-View-Controller software architecture. Having independent components pays off greatly when you want to implement changes in your project. When two components depend on each other, changes in one component affect the other component. This greatly affects the time it takes to implement changes for the worse. Given that we only have approximately ten weeks to fin

1.1.3 Maintainability

Given that we are working on this project for ten weeks with a group of five people. If we do not pay attention to code maintainability, a lot of time would be wasted rewriting old code first in order for new functionality to be added during each sprint. In order for our code to be maintainable we test rigorously before new features are even added to our developing version of the code, let alone our version on the master branch.

2 Software Architecture Views

2.1 Subsystem Decomposition

We have designed our system following the MVC architecture, dividing it into multiple subsystems. Our project can be divided into three main parts: models, views and controllers.

The models can be found in the contextproject.models package. The classes included in this part of the project enable us to do the calculations and store the data we need to match tracks and construct the playlists for the users. This package contains the following classes : BeatGrid, BeatRange, MusicalKey, Library, LibraryProperty, Playlist, Track and TrackProperty.

The controllers can be found in the contextproject.controllers class. These classes take care of the interaction between our models and our views. One of the classes in this package,the CliController, is intended specifically for our Command Line Interface.
The other controllers: LibraryController,Playlistcontroller,MenubarController, PlayerControlsController and WindowController are to be used in combination with our GUI.

Our views can be found under resources. We have separated them from the other code, because they are not implemented using Java and the team thought it was best not to mix java files with other file formats. Our views themselves are implemented in FXML, which is part of the JavaFX framework. This package also includes CSS stylesheet, which contain all the styling elements of our views. For now we have the following views: library, menu_bar, player_controls, track_info and window. Each of these views has it's own stylesheet.
Our project contains additional subsystems, that perform multiple auxillary functions.

For instance, the contextproject.audio package contains services related to the music player in our application. It contains the EnergyLevelProcessor, the MixerProcessor,PlayerService and TrackProcessor.
The contextproject.formats contains classes that help us store data in the desired format. The class XmlExport helps us relevant information to an XML file. The M3UBuilder class helps us construct playlist that can be played in a media player

The contextproject.helpers is a package that contains the FileName, StackTrace and TrackCompatibility class. The FileName comes in handy when dealing with MP3 files, the StackTrace class helps us during logging and the TrackCompatibility performs the function of mathching two tracks.

The sorters themselves can be found in the contextproject.sorters package. Currently we have two different sorters we can switch between. One of them is the GreedySort sorter and the other is the MaxFlow sorter. The package also contains the data structures used by this MaxFlow algorithm, such as the Graph, TrackNode, weightedEdge, TrackTree and the MaxFlow which calculates the maximum flow in the constructed graph.

Last but not least, in the contexproject package we have the App class. This starts the application itself and in fact properly glues the subsystems together.

2.2 Hardware/Software Mapping

For the moment, our system just runs on one computer at a time. In that case, there is no inter-computer communication. The classes in contextproject.audio rely quite heavily on the functionality of the users soundcard. For now no internet connection is required whatsoever. The only thing that is also required is a music collection or a way to acquire new music. That could be a disk drive and cd's or internet. The choice is up to the user in that specific aspect.

2.3 Persistent Data Management

In this version we store all the user music data in an XML file at the program closing. So when the program starts all the playlist will be there and the calculating part has to be done once per playlist. Upon addition of a new music file, the calculations are run again and the new playlist is added to the library.xml file. The other playlists will remain untouched unless the user deletes them manually. We chose to store this data in XML-format for multiple reasons. It was fast to implement, it is a reliable format for storing data and we were familiar with storing data in this way.

2.4 Concurrency

The initial version worked with a CLI and therefore it was not attractive to implement multithreading in that release. All later releases include a fully-fledged GUI and more intricate sorting algorithms. This gave us enough incentive to implement multithreading in our application.

2.5 Design Patterns

For the GUI of our application, we used the javaFX framework. Our views are therefore implemented through FXML files. This is a markup language and show strong similarities to XML. Our FXML files are all implemented following the composite design pattern. FXML is naturally compositional, because of the tree structure it employs.

Our project also contains classes designed following the Singleton design pattern. Our project only needs to include one instance of the App, Library and Logger classes. Therefore we have implemented them in ways that allow our application to run only one instance of these classes in the entire application.

The reason we use singleton for the library is that when there is more than one library, the libraries will overwrite each other on exit of the program, which will result in missing data. This will hinder one important feature of our project: storing matched playlist. If tracks suddenly go missing from your library, the entire experience we hope to offer to our users is lost on them. That would be very stupid of us to even consider taking such a risk. Going for this pattern was our only option in this case.

The logger has to be a singleton, otherwise you get inconsistencies. If there would be more than one logger, the process of logging would become very unstable. It is possible that the logger would create multiple logging files, one for each instance which would make our application memory inefficient. There could be inconsistencies within a file, such as some data being logged multiple times by each of the different loggers. This is a lot of risk to take for such a small part of our program. Altough these errors are unlikely to occur, it wouldn't be the first time that Murphy's Law worked it's magic in a software project. Therefore we decided it to be best to avoid them altogether by using this design pattern.

3 Glossary

A UML Diagram

B Testing Report

For testing our application we used a combination automatic and manual testing. Most of our packages were tested mostly using JUnit, as they were easy to test automatically. However, two of our packages: THe contextproject and contextproject.audio were very hard to test automatically. That is why we tested these ones manually.
Aside from JUnit, we also used other Frameworks for testing: Mockito and TestFX. We used Mockito for dependency injection were possible. This was not always possible, because we required very specific data from a class, which doesn't go well when mocking these instances.
We used TestFX for testing the controllers package. TestFX includes a robot, the FXRobot, which can simulate user interaction such as clicking, typing and even entering files.