Design Document for Code Assignment
Giesecke Devrient

April 13, 2021

Problem Description

The problem discussed in this document consists of modelling the operation of a music venue. At this venue there are, in the initial state, two bands are scheduled to play each night. We must model the bands and all the musicians they consist of. We must also remember to keep track of musicians who are currently looking to join a new band. Additionally we must model the instrument of each musician, as every instrument has some certain properties.

Then, after their performance, each night every band loses a member at random. Finally every musician who is not currently in a band will attempt to join a random band playing at the venue. They will be allowed to join if and only if there is no other musician playing the same instrument already in the band.

On a technical level the solution to this problem must be implemented as a command line interface, with logic to list the current state of the model, to play one night of venue logic (bands losing and gaining random musicians) and to exit the program.

Solution Design

Using the Java programming language, this is how I structured my solution to the above problem.

Command Line Interface

The CLI is how the user interacts with the program. It handles all of the commands listed in the Task Description document, as well as some additional ones for convenience. The available commands are:

- Help, h - Displays the help information
- List, l - Lists the current state of the model
- Play one, Play - Executes one night of Venue logic
- Play X - Executes X nights of Venue logic, where X is a whole number.
- Exit, Quit, q - Exits the program

All of the above commands are capitalization agnostic. Any command entered not in this list will cause *"Invalid command! For more info use 'help'."* to be printed out. I've chosen to use this error message rather than printing the help information directly to avoid spamming the user after a simple typo. Many of the commands also simply exist as short-form commands and are implemented for efficiency.

The "Play X nights" command was implemented as it was easy to add and felt like a logical option to have.

Class Structure

First and foremost it is very important in Java to efficiently use Classes and Object oriented programming to achieve a good solution.

Main.java

The file **Main.java** simply initializes and launches the program. The job of the **Main** class is to create the **Venue** object which hosts the program logic, and to populate it with the initial state as described in the Problem Description document. This is designed to be highly modular and make it easy to alter the initial state or for instance expand the program to read the initial state from a file or similar.

Venue.java

The file **Venue.java** models the Venue, handles the Command Line Interface and executes the problem logic. This code makes up the bulk of the program and will keep track of the current state of the model. This state is represented by two **ArrayList**s as fields, one storing all of the bands and the other storing all musicians not currently in a band.

Calling the **manage()** function starts the CLI and allows the user to interact with the model. It is then the (infinite) while-loop at line 41 which handles reading from Java's *System.in* and handling the input. To do this I used a switch-case, which seemed suitable as there are not terribly many commands to handle and using a switch-case it is easy to add for instance duplicate commands with the same function. Most of the cases here simply call some other function, which should make it relatively easy to implement a more advanced parser using for instance a separate class if one would want many or more advanced command words.

The **Venue** logic is exetuted in the **playNight()** function, and is honestly quite simple. At first it will iterate through the list of **Band**s and remove one member at random. The removed members are stored in a temporary list. Then it will iterate though the list of 'unemployed' **Musician**s who will attempt to join a random band. There was some ambiguity here in the Problem Description document whether the bands were meant to attempt to recruit a random musician, or the musicians attempt to join a random band. I chose the latter interpretation as it seemed more interesting. This means that several musicians can join the same band in the same night. Finally those **Musician**s who were kicked from their bands on this night (and stored in a temporary list) are added to the list of 'unemployed' musicians. I modelled it this way to ensure that no musician could be kicked from and rejoin the same band in the same night.

Band.java

The **Band.java** file is rather uninteresting and simply models a **Band**. It keeps track of the name of the band and a list of its member **Musician**s. The class also contains some logic to add or remove musicians to the roster, for instance ensuring that musicians with duplicate **Instrument**s are not allowed to join. This is done by String comparison on the **Musician**'s **Instrument**'s type.

The Instrument Model

This was the most challenging part of the assignment to model. There are several different types of instrument which share many features, such as **Banjo**s and **Guitar**s sharing notes relating to their number of strings or the string thickness. As such I chose to model **Instrument.java** as an *Abstract Class*, keeping track only of the field which all **Instrument**s have in common - the *manufacturer.*

Additionally there are two more *Abstract Classes* in my modelling of the instruments. Since the information for Banjo and Guitar as well as for Piano and Synthesizer were so similar I created common parent classes for these instruments, **StrumInstrument** and **KeyInstrument**. These files respectively keep track of the fields these instruments share. The **Drums** class does not have this kind of structure, since it does not share features with any instrument in the current model.

Finally each **Instrument** has its own class file which is not *abstract* and can as such be instantiated and used as an object. These files contain any instrument specific fields, like whether or not a **Synthesizer** has LEDs. My reasoning for modelling the **Instruments** like this is that it is now highly modular and would be easy to implement instrument specific logic such as functions regarding playing the instrument or adding more instrument specific fields. It is also very easy to add additional types of **Instrument**, should new **Musician**s show up.

It is also worth noting that I have contained all of the **Instrument** files in their own package, as to avoid cluttering of the main project file. It is useful to keep related files together and apart from the rest of the project for simplicity.

Initial State

As I have already touched upon the Initial State of the **Venue** is currently hard-coded into the **Main** class. This is because the given state was quite simple, only featuring two bands and seven musicians. I have, however, placed this initialization separate from the bulk of the program to make it easy to initialize a **Venue** with a different method. If more advanced models would need to be loaded, it would not be difficult to read the initial state from a file and add them to the **Venue** using the **addBand(Band b)** and **addFreeMusician(Musician m)** functions. A general solution could for instance be using a *.json* file.