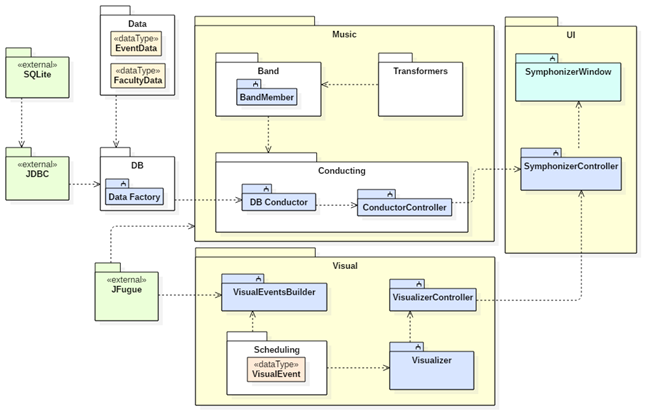
Components Overview

****

# External Libraries

## SQLite + JDBC

Lightweight database with all basic SQL features. Used to hold our event database.

While it doesn’t contain advanced features like aggregate functions, it was vastly more important for us to have a portable lightweight database that can reside as close to the project as possible.

The database is accessed via JDBC using an adapter library, such that apart from the connection to the database, all components accessing the database become unaware of the underlying SQL implementation; SQLite, Oracle, MySQL…

This was important for us as we were not sure if we were going to stick with SQLite when we started, and wanted the option to change quickly. In fact, we used Microsoft Access when we started but later dropped it due to suspicion in regard to portability.

## JFugue

Extensive musical MIDI library, allowing us to compose musical patterns via readable strings. The library comes with the concept of a Pattern Transformer, which reads the tokens parsed by the string, intercepting certain tokens - modifying their values or changing their order.

This allows us to create useful transformers like a layer-based masterer which can change the velocity/amplitude of a track based on the layer within the track, effectively allowing us to remaster patterns.

Of course, in addition to that, develop algorithms to generate music.

# Data

## Responsibility

Separating layer between the musical logic of our project and the actual database.  
Provides data classes and correspondent factories.

## Event and Faculty Data

Our most important and prevalent data classes in the project.

Every event is ‘parented’ by a faculty, and every faculty contains aggregated events pertained to it. This aggregation does not occur on the data layer, and instead occurs while conducting, as the incomplete aggregation state (Total events as of year … for faculty ...) is important for the conducting process.

## Data Factory

Composed of different factory classes for relevant Events database entities, allows us to easily access different information and data tables for different purposes. (Using SQL queries)

# Music

## Responsibility

Provides the main musical functionality of the project, namely modifying and creating musical patterns - while other packages may parse patterns but not modify.

## Band Member

An interface implemented by each faculty, whose implementation guarantees the ability to play a musical sequence catered to a specific role (melody, carpet, etc.) within a bigger musical composition, while dictating each faculty’s musical characteristics (instrument, sequencing type, etc).

## Band

Simply a collection of Band Members to be delivered to the DB Conductor. Can be considered as a mapped strategy pattern for which every faculty ID is mapped to a different band member.

## Transformer

Musical function library focused on modifying a complete pattern in an iterative manner - iterating on staccato tokens which are analogous to MIDI messages.

## Conducting - DB Conductor

Data-based bolero conductor, which iterates on IDC events, dividing years into fixed-length sequences and uses generic commands to procedurally generate a song resembling the structure of Ravel’s Bolero (though not its flavour)

### Events Bolero Structure

Subsystem component in charge of dividing the years into sequences per year statistics.

Scores years by complexity, which is considered (#(Events) \* #(Unique Faculties) \* #(Unique Event Types)), then uses the score to decide how many sequences should represent each year.

This scoring method was inspired by observed traits; where the first year had more events than most other years, but they were all building construction events “by” administration, which meant the year wasn’t eventful in a way that can produce musical complexity, and thus shouldn’t be split into multiple sequences.

### Command

Recurring commands that are executed either on every year, every sequence or every event.

Commands are given the conducting context which holds both the statistical data required to make decisions as well as the composition data structure, allowing commands to collaboratively construct the song.

Using Band Member as well, the commands may be as simple as giving each year the faculty with the most events in the year the melody role - and the faculty, which implements Band Member, may return a musical pattern representing that role.

## Conducting - Conductor Controller

The controller acts as a bridge from a database file to a resulting pattern for the top-level application, providing functionality for asynchronous requests that will not detract from UI responsiveness.

The controller holds two different conductor strategies (and thus async requests):

The first is intended to fully generate music from the database

The second is intended for embedding visualization metadata into an existing MIDI (Planned ahead - for re-embedding the remastered song and visualize its notes properly).

# Visual

## Responsibility

Provide simultaneous visualization of:

* Faculty branching structures
* Faculty names
* Faculty BandMember role distributions
* Musical Playback

… in a fun, and visually pleasing fashion.

## TPoint

A spatial 2D point that utilizes positional vectors, velocity vectors, and basic interpolation techniques to provide a fluid, tacky ‘physics-inspired’ characteristic to visual components for animation purposes.

## Trail

A stringy collection of TPoints with two controllable ends (a head and a tail) which propagate visual properties (radius, position, velocity) from head to tail via utilization of linked lists.

Multiple trails are put together to form the faculty branching tree. Musical notes played by a faculty visually propagate through its corresponding trail.

## Infograph

A text-based visual component with an animated ‘pointing line’ used to provide textual information to graphic information within the Visualizer, specifically each trail’s faculty name.

## Visualizer Controller

Acts as a bridge between a musical pattern and visualization.

Contains the process / request for converting a musical pattern into Visualizer-expected events classes, as well as the ability to play the song in sync with the visualizer.

# UI

## Responsibility

Ability to view and generate the visualization in a user friendly manner using JavaFX.

## Symphonizer Window

The tool’s window, generated by JavaFX fxml, contains almost zero logic as logic is caused via databinding in the fxml level.

Contains three main layout components:

* Control menu for selecting the parameters for conducting requests, as well as starting and stopping the visualization
* Visualization pane containing our visualization
* Friendly log which can show the processing of the events and faculties’ resulting roles, allowing us to compare with the song playing before the visualization is feature complete.

## Symphonizer Controller

Top level controller which applies stateful data binding to the elements in Symphonizer Window;

For instance, binds the conducting button interaction to the availability of the database file.

Contains both lower level controllers - Visualizer Controller and Conductor Controller, and links user action with correspondent interactions in these controllers.