My Phish Stats

Michael Voecks

Final State of the System

The final version of the program allows users to interface with the command line to add and remove shows to their personal collection, as well as run statistics on their collection of shows. The program adds shows by contacting the `phish.net` API to gather setlist data for requested dates, and stores this shows in a database stored locally. Shows that have been previously added to the database will have their information added locally instead of contacting the API. In addition, the information regarding which shows a user has in their collection is also stored locally in the database. Finally, the user is able to view the setlist for any show that is in their collection.

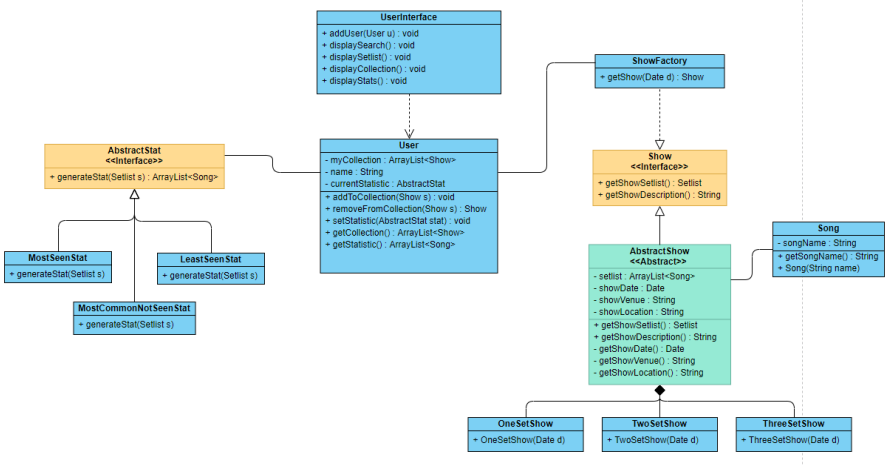
There are four statistics that a user can calculate for their collection of shows:

1. Top 10 Most Seen Songs
2. Top 10 Least Seen Songs
3. All Unique Songs Seen
4. All Unique Shows Seen

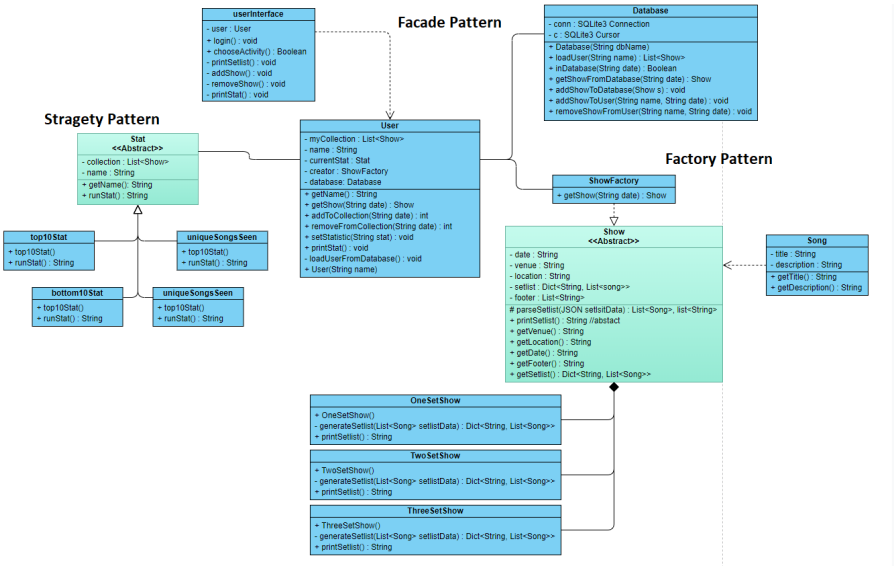
Originally this project was intended to include a graphical user interface for the user to display shows, however due to time constraints and a lack of experience with GUI development, this was replaced with a command line interface. In addition the `phish.net` API limits the number of requests for users, so the ability to search for setlists was not implemented as our API Key would not be able to request every show.

Class Diagram

At the start of the project this was our hypothetical UML Class Diagram:



However over the course of our development some areas of this design were expanded and others removed entirely, resulting in the following class diagram:



Main differences in these two class structures include changing all interfaces to abstract classes, which was done simply because Python's ABC module doesn't easily support interfaces, and abstract class would achieve the same goal. Due to this the factory pattern for generating shows was simplified to not include the Show interface, and instead show factories would directly create abstract shows. This could be a problem if phish shows make a dramatic change that would require a new abstract class to define them and we would have to refactor this code to include Show interfaces. The last notable difference between the two diagrams is the inclusion of the Database class through the Facade pattern. This allowed for simplified code within the User class and abstracted all database work into its own class.

Third Party Code

The following Python modules are required in order to run this code:

- abc : `https://docs.python.org/3/library/abc.html`
- re : `https://docs.python.org/3/library/re.html`
- heapq : `https://docs.python.org/2/library/heapq.html`
- BeautifulSoup : `https://www.crummy.com/software/BeautifulSoup/bs4/doc/`
- requests : `https://realpython.com/python-requests/`
- sqlite3 : `https://docs.python.org/3/library/sqlite3.html`

I had used many of these modules in previous python projects, and only needed syntactic guidance to write code for this program, however I used the modules BeautifulSoup, requests, and heapq for the first time and used the tutorials and docs given in the links in the list above.

Development Process

Getting Started

The start of this project was the most difficult aspect of the entire development process. Traditionally I think of the structure of programs as I write them which made it hard to plan out all the small details of the project before any code was written. However it became very useful to have most of the kinks ironed out of the design as it made the actual production of the code much faster. There was no blind searching for how to make things work and instead their was a clear path to creating a functioning program, which resulted in cleaner more reliable code. In addition I was very ambitious in what I sought to achieve by the deadline and had to scrap out some big features such as the Graphical User Interface.

Commonality and Variability Analysis

When creating the class structure for the program I was trying to pinpoint exactly what each class would do and how to abstract it from my other classes, and even after careful consideration on my first draft many things changed in the development process. For example I realized fairly quickly that the Show interface and Show abstract classes would be to similar in their end result, so I combined them into just one Show abstract class. Similarly I realized that the user shouldn't have to also manage the connection with the database, and so the database functionality was moved into its own class.

Testing

For as hard as it was to put the code into place, an even larger task was making sure it all functioned properly. If given more time it would have been nice to implement unit tests, especially for hard tasks like populating setlist data and database entries/deletions. Instead most of the testing done for this project was test-during development. I had to make sure I could properly input show data into the Show class before I could generate any statistics on the user's collection, however there were multiple occasions where lack of testing led to bugs in class communication and functioning.