Muze Music Library System v2

Design Documentation
Prepared by Team 05:

- Jarred Moyer <jam4936@rit.edu>
- Cameron Riu <cmr6689@rit.edu>
- Shane Burke <sdb5978@rit.edu>
- Fahd Masood <fxm1492@rit.edu>
- Ryan Borger <rlb8800@rit.edu>

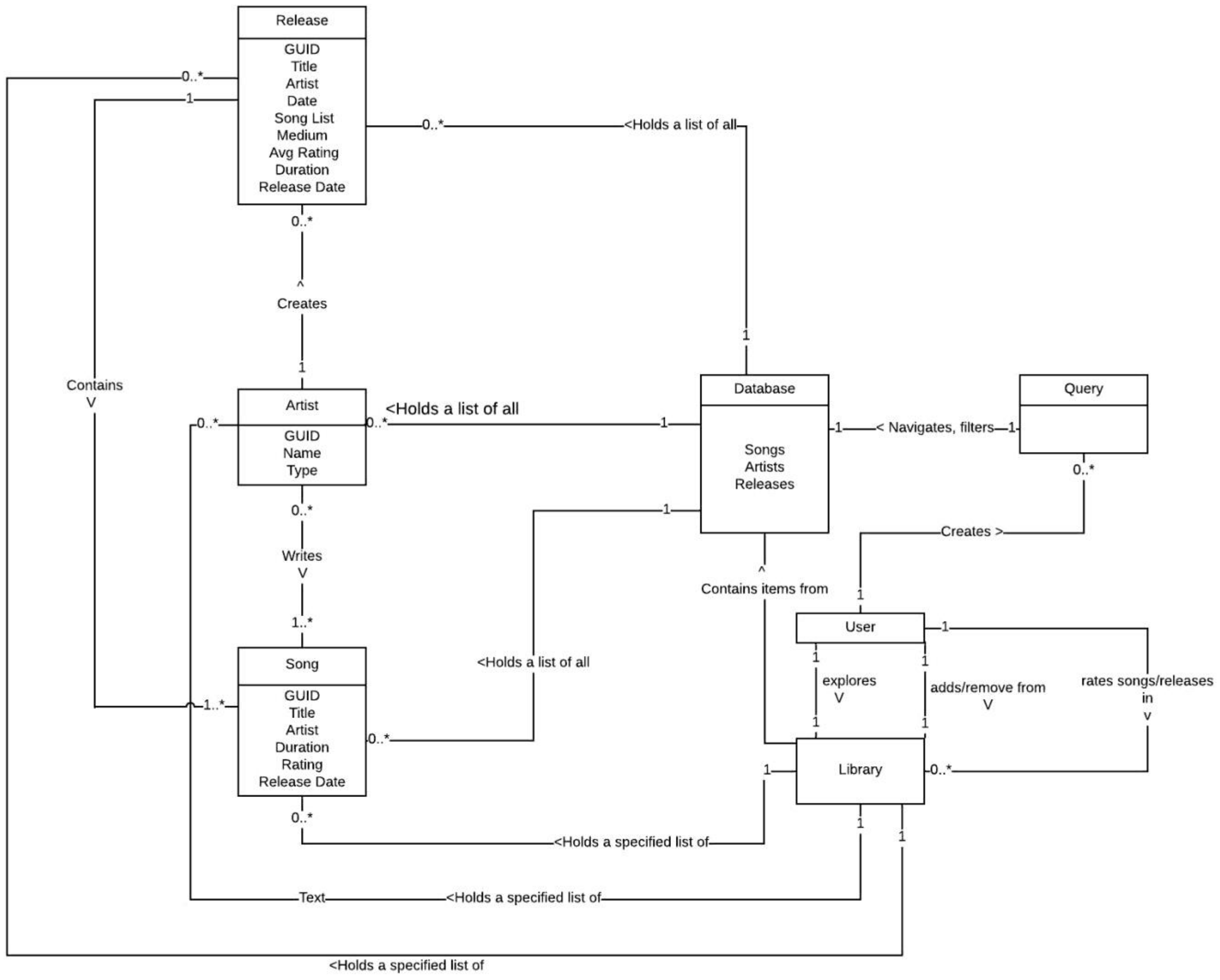
Frequently Used Terms

Term	Definition
MMLS	Muze Music Library System
Library	User defined collection of releases, artists, and songs
Database	Program defined list of all possible songs, artists, and releases

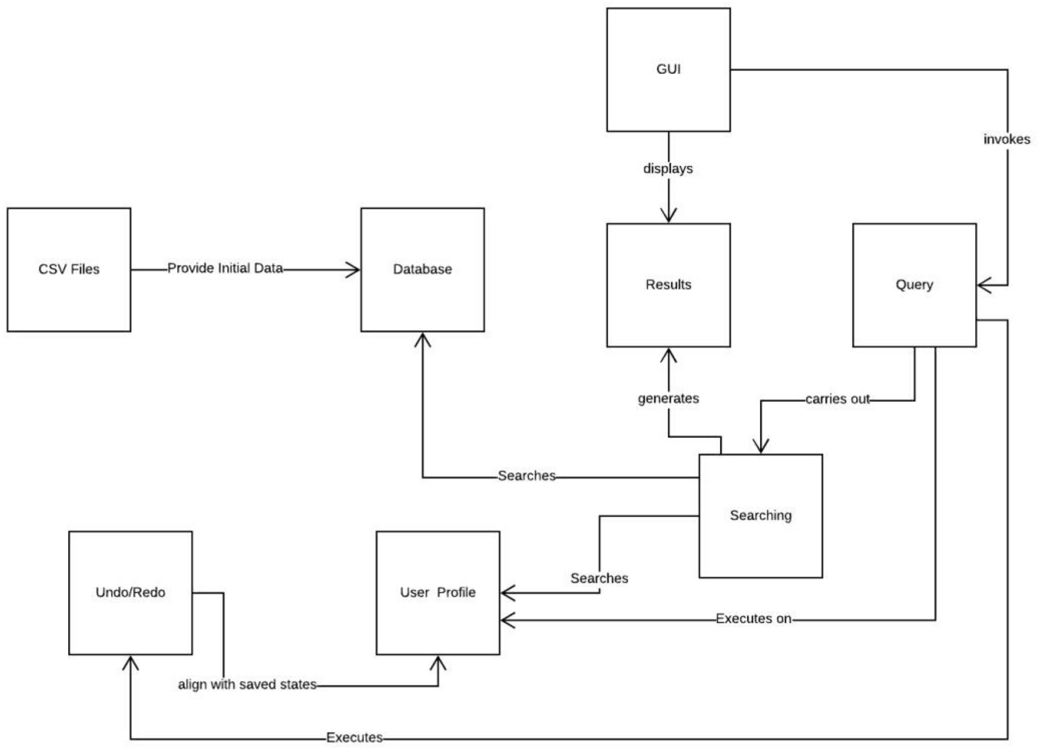
Summary

The MMLS aims to allow music-lovers a way to track the artists, songs, and releases in their collections. This system has two components, the library, and the database. The database contains a list of all possible items that a user could have in their database. Users cannot mutate this database. This database can be sourced from two locations, a remote database or directly parsed on the client's machine. Users can specify where they want to source the database on launch. The other component, the library is mutable to the user. This library can have songs and releases added (from the database) and removed from it. The user interacts with this system with a query. Queries can search the database, filter results, sort results, sort the library, and apply ratings to releases and songs. These interactions can be carried out by a GUI or the command line, which the user can switch between at whim by changing windows.

Domain Model



System Architecture



Database: Controls the construction and handling of both online and offline databases. Controls all communication for the online database.

Query: Converts user specified queries to executable statements.

Results: Stores the output of a query to run further commands against and to view

Searching: implements filters, sorts, as well as executes query commands to generate results

GUI: Provides the user an easy way to navigate and interact with the Muze Music Library System.

Undo/Redo: Manages and records changes made to the personal library such that they can be undone or redone during the session.

User Profile: Controls the persistence and management of a users personal library.

Subsystems

This section provides detailed design for specific subsystems described in the system architecture.

Searching

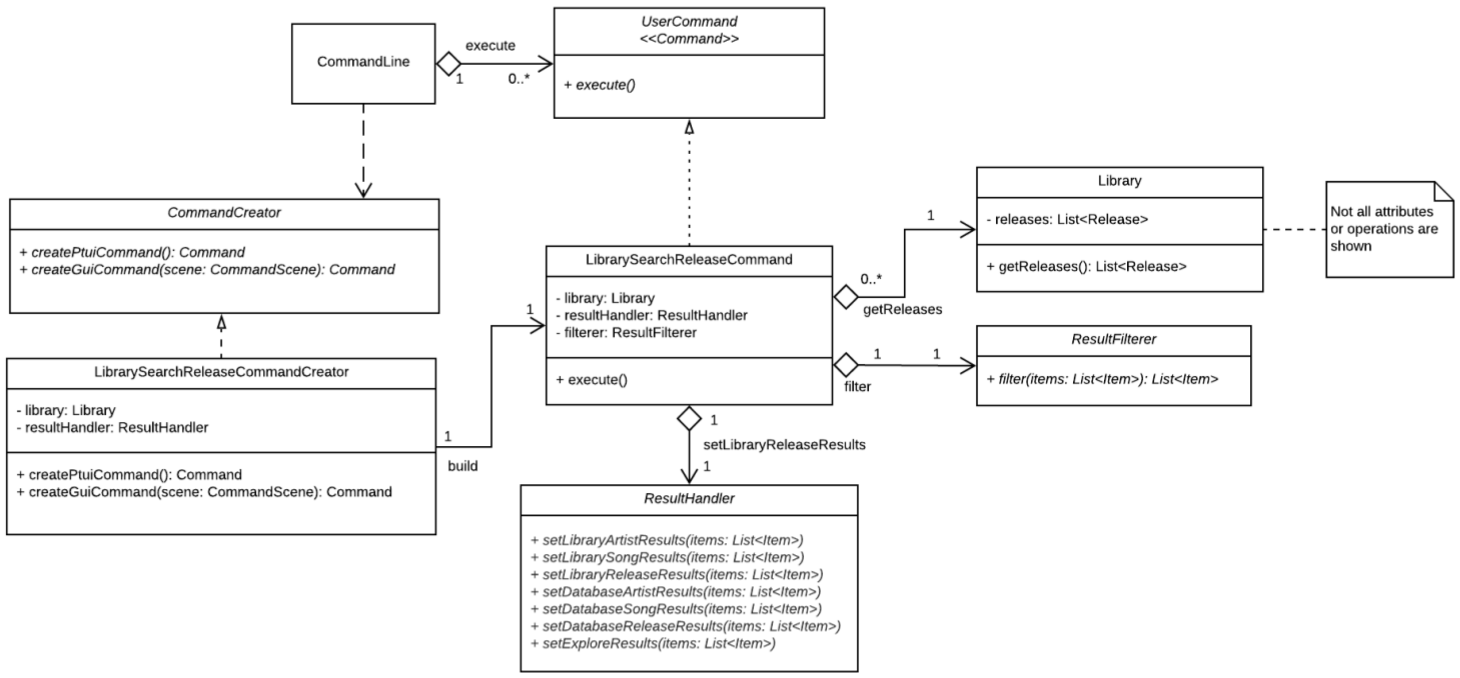

CommandLine
execute
1
0..*
UserCommand
<<Command>>
+ execute()
CommandCreator
+ createPtuiCommand(): Command
+ createGuiCommand(scene: CommandScene): Command
LibrarySearchReleaseCommandCreator
- library: Library
- resultHandler: ResultHandler
+ createPtuiCommand(): Command
+ createGuiCommand(scene: CommandScene): Command
build
LibrarySearchReleaseCommand
- library: Library
- resultHandler: ResultHandler
- filterer: ResultFilterer
+ execute()
getReleases
filter
setLibraryReleaseResults
Library
- releases: List<Release>
+ getReleases(): List<Release>
Not all attributes or operations are shown
ResultFilterer
+ filter(items: List<Item>): List<Item>
ResultHandler
+ setLibraryArtistResults(items: List<Item>)
+ setLibrarySongResults(items: List<Item>)
+ setLibraryReleaseResults(items: List<Item>)
+ setDatabaseArtistResults(items: List<Item>)
+ setDatabaseSongResults(items: List<Item>)
+ setDatabaseReleaseResults(items: List<Item>)
+ setExploreResults(items: List<Item>)

Figure: The searching subsystem as shown in the context of a command line search for releases in the library.

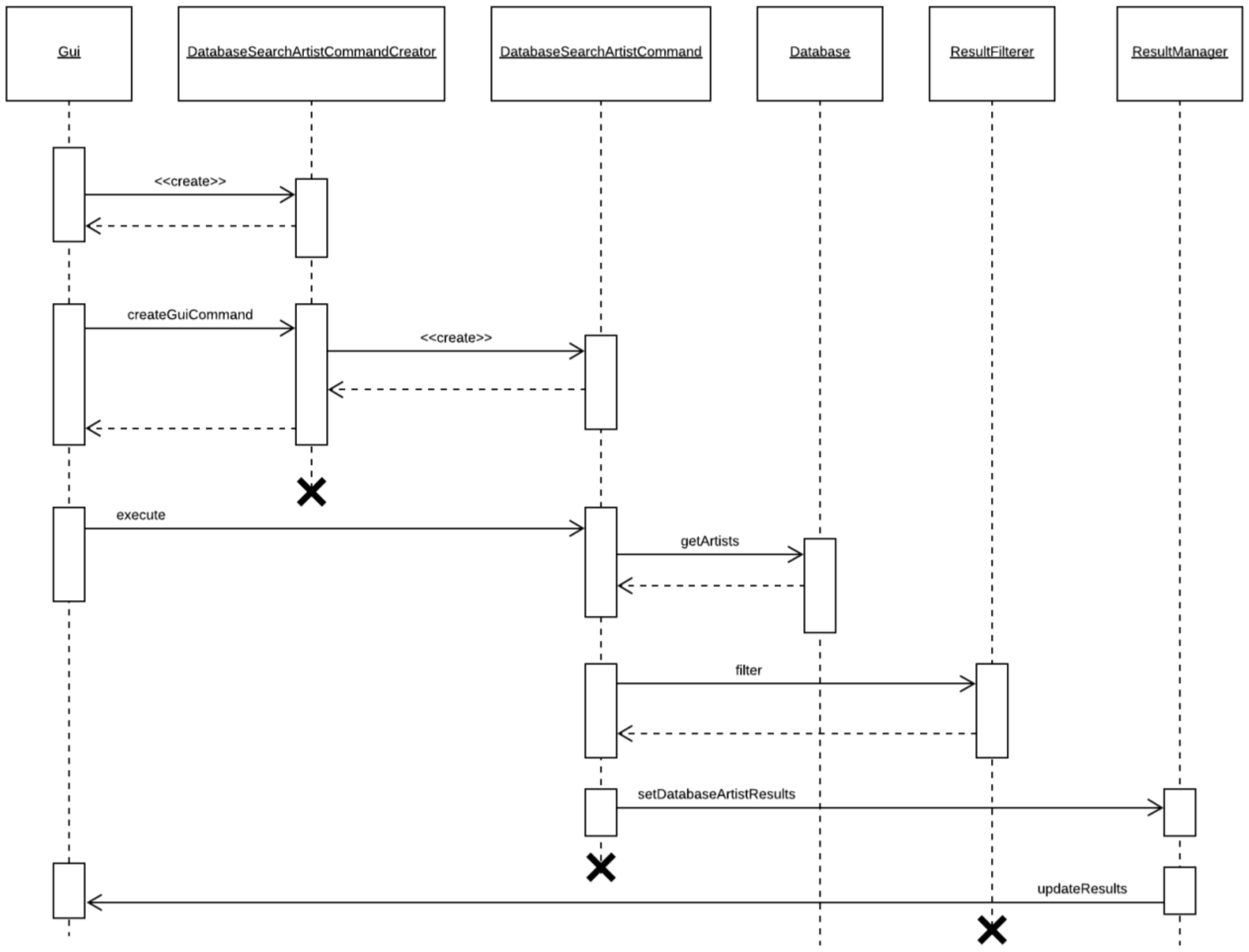


Figure: Sequence diagram for a search for artists in the database, originating from the GUI. The *DatabaseSearchArtistCommand* is created, after which the database's list of artists is obtained and filtered. The *ResultManager* is updated with the new results, then it notifies the *Gui* of the change. To learn more about the interaction between *ResultManager* and *Gui*, see the Observer pattern in the Results Subsystem section.

Summary

The searching subsystem carries out queries that retrieve and filter results from either the library or database. When the user makes a query (from either user interface) that requests music items as its results, the user interface (acting as the client) creates a *UserCommand* object through the *CommandCreator* interface. The *ResultHandler* object, and the *Library* or *Database* object (as applicable to the command) are passed into the *CommandCreator.* The Decorator pattern is used to construct complex filters for search results. While creating the command, the *CommandCreator* creates and decorates a *ResultFilterer* object with filters for all desired search parameters. The created *UserCommand* is then invoked by the client.

Design Pattern: Decorator

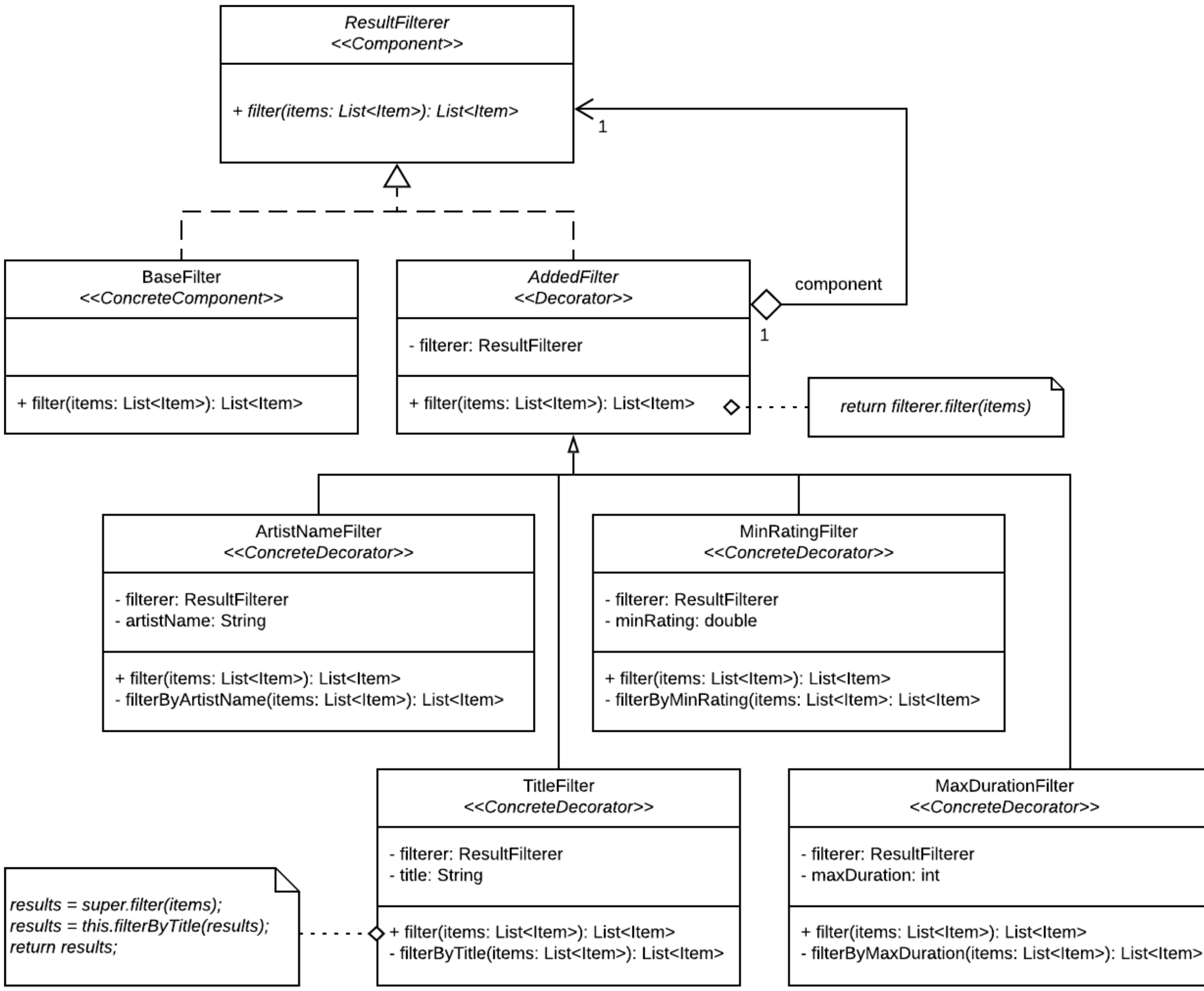

ResultFilterer
<<Component>>
+ filter(items: List<Item>): List<Item>
1
component
1
BaseFilter
<<ConcreteComponent>>
+ filter(items: List<Item>): List<Item>
AddedFilter
<<Decorator>>
- filterer: ResultFilterer
+ filter(items: List<Item>): List<Item>
return filterer.filter(items)
ArtistNameFilter
<<ConcreteDecorator>>
- filterer: ResultFilterer
- artistName: String
+ filter(items: List<Item>): List<Item>
- filterByArtistName(items: List<Item>): List<Item>
MinRatingFilter
<<ConcreteDecorator>>
- filterer: ResultFilterer
- minRating: double
+ filter(items: List<Item>): List<Item>
- filterByMinRating(items: List<Item>: List<Item>
TitleFilter
<<ConcreteDecorator>>
- filterer: ResultFilterer
- title: String
+ filter(items: List<Item>): List<Item>
- filterByTitle(items: List<Item>): List<Item>
MaxDurationFilter
<<ConcreteDecorator>>
- filterer: ResultFilterer
- maxDuration: int
+ filter(items: List<Item>): List<Item>
- filterByMaxDuration(items: List<Item>): List<Item>
results = super.filter(items);
results = this.filterByTitle(results);
return results;

Figure: The Decorator pattern's class structure. Note that some Concrete Decorator classes have been omitted, however their descriptions can be found in the GoF pattern card below.

Role

When searching the library or database, the list of all songs, releases, or artists is narrowed down through the application of successive filters. The Decorator pattern makes it so that this process can be completed using a single object's interface method. A single compound result filter is built by applying additional filters to a base filter (which does not remove any results). By recursively decorating the filter with any number of decorations, a customized filter object is created with many additional behaviors.
The resulting result filter object is supplied to a command object, wherein it is used upon command execution. The filter object, given a list of items, applies the behaviors of each decoration, and finally

the base filter, to produce a fully filtered list of results.

Design Considerations

The *BaseFilter* class actually has no filtering behavior. It simply returns an identical list of items to the one supplied to it in the *filter(...)* method. This may make the class seem pointless. On the other hand, it represents the fact that no base filtering exists on results until it is applied. For example, a command to list all artists in the library carries no filtering requirement, so a *BaseFilter* may be used. The same *ResultFilterer* may still be used, thus hiding the information of what filters were applied to the results.

Each result-producing query in the application has a defined set of parameters (filters) it accepts. Without the Decorator pattern, classes that provide the possible filters for each type of search command could have been created. This model would have been inflexible, as new filter options could not be added to an existing command without modifying the customization behavior of that command's class. Additionally, it inhibits code reuse as the filters cannot be freely applied to any existing filtering process (as is the case with the Decorator pattern).

The *AddedFilter* abstract class defines the possession of a parent ResultFilterer, which may be another *AddedFilter* or a *BaseFilter*. The possibility of having *AddedFilter* be an interface was considered, however it was deemed appropriate for all *AddedFilter* subclasses to share this trait. This decision is low risk, and it reduces the complexity and redundancy of the system. Future *AddedFilter* sub-classes will undoubtedly carry the same attribute.

Class	**Role in GoF pattern**	**Participant's contribution in the context of the application**
ResultFilterer	Component	An interface representing objects which accept, filter, then return a list of items. The filtering object may carry any number of decorations, but this is unknown to the client. Used by commands to filter search or exploration results.
BaseFilter	Concrete Component	A filter which removes no results from the supplied list. It is used in cases where no filtering is required, but the filtering process remains to promote uniformity with results that require filtering. It is the base component upon which decorative filters are applied.
AddedFilter	Decoration	Represents a filter that adds additional filtering behavior to an existing BaseFilter or already decorated filtering object. It is supplied another Component so that its existing filtering behavior may be performed in addition to the added behavior.
ArtistNameFilter	Concrete Decoration	When supplied a list of items, returns a new list that only includes the items for which the artist name matches the one specified in the filter. Used when searching for items by a specific artist.

MinRatingFilter	Concrete Decoration	When supplied a list of items, returns a new list that only includes the items for which the minimum rating is equal to or greater than the one specified in the filter. Used when searching for library items that have a rating above a minimum threshold.
ArtistGuidFilter	Concrete Decoration	When supplied a list of items, returns a new list that only includes the items for which the artist GUID matches the one specified in the filter. Used when searching for items by a specific artist.
TrackNameFilter	Concrete Decoration	When supplied a list of items, returns a new list that only includes the items for which an included track’s name matches the one specified in the filter. Used when searching for releases containing a specific track.
TrackGuidFilter	Concrete Decoration	When supplied a list of items, returns a new list that only includes the items for which an included track’s GUID matches the one specified in the filter. Used when searching for releases containing a specific track.
TitleFilter	Concrete Decoration	When supplied a list of items, returns a new list that only includes the items for which the title matches the one specified in the filter. Used when searching for items with a specific title.
MinDurationFilter	Concrete Decoration	When supplied a list of items, returns a new list that only includes the items for which the total duration exceeds the specified minimum. Used when searching for songs or releases with a minimum duration.
MaxDurationFilter	Concrete Decoration	When supplied a list of items, returns a new list that only includes the items for which the total duration is lower than the specified maximum. Used when searching for songs or releases with a maximum duration.
ArtistTypeFilter	Concrete Decoration	When supplied a list of items, returns a new list that only includes the items for which the artist type matches the specified type. Used when searching for artists of a specific type.
ReleaseTitleFilter	Concrete Decoration	When supplied a list of items, returns a new list that only includes the items for which the release title matches the specified title. Used when searching for songs belonging to a specific release.
ReleaseGuidFilter	Concrete Decoration	When supplied a list of items, returns a new list that only includes the items for which the release GUID matches the specified GUID. Used when searching for songs belonging to a specific release.

MinReleaseDateFilter	Concrete Decoration	When supplied a list of items, returns a new list that only includes the items for which the release date is later than the specified date. Used when searching for releases with release dates after a certain date.
MaxReleaseDateFilter	Concrete Decoration	When supplied a list of items, returns a new list that only includes the items for which the release date is earlier than the specified date. Used when searching for releases with release dates before a certain date.
SortFilter	Concrete Decoration	When instantiated, the *SortFilter* is supplied a *Comparator* object. This object is used to sort the results given by the *filter(...)* method. In contrast to the other *AddedFilter* subclasses, this decorator does not remove any results from the given list. It simply sorts the results and returns them.
Deviations from the standard pattern: None		

Requirements being covered:

- The user may specify various parameters when searching for music items.
- Applying multiple parameters to a search will return results matching all of the specified parameters.
- Some search parameters may be omitted, as long as the nature of the desired results is somehow specified.
- Different types of searches accept different possible parameters.

Query

The Query subsystem handles the processing of user queries. These queries may be generated by the triggering of GUI elements, or through textual command line entries by the user. Through the Command pattern, user input is transformed into an executable command. When invoked, this command may generate results in the form of a list of music items. These results, and the determination of valid operations utilizing them, are managed by the State pattern. When results are generated, they are communicated to user interface controllers through the Observer pattern.

Design Pattern: Command

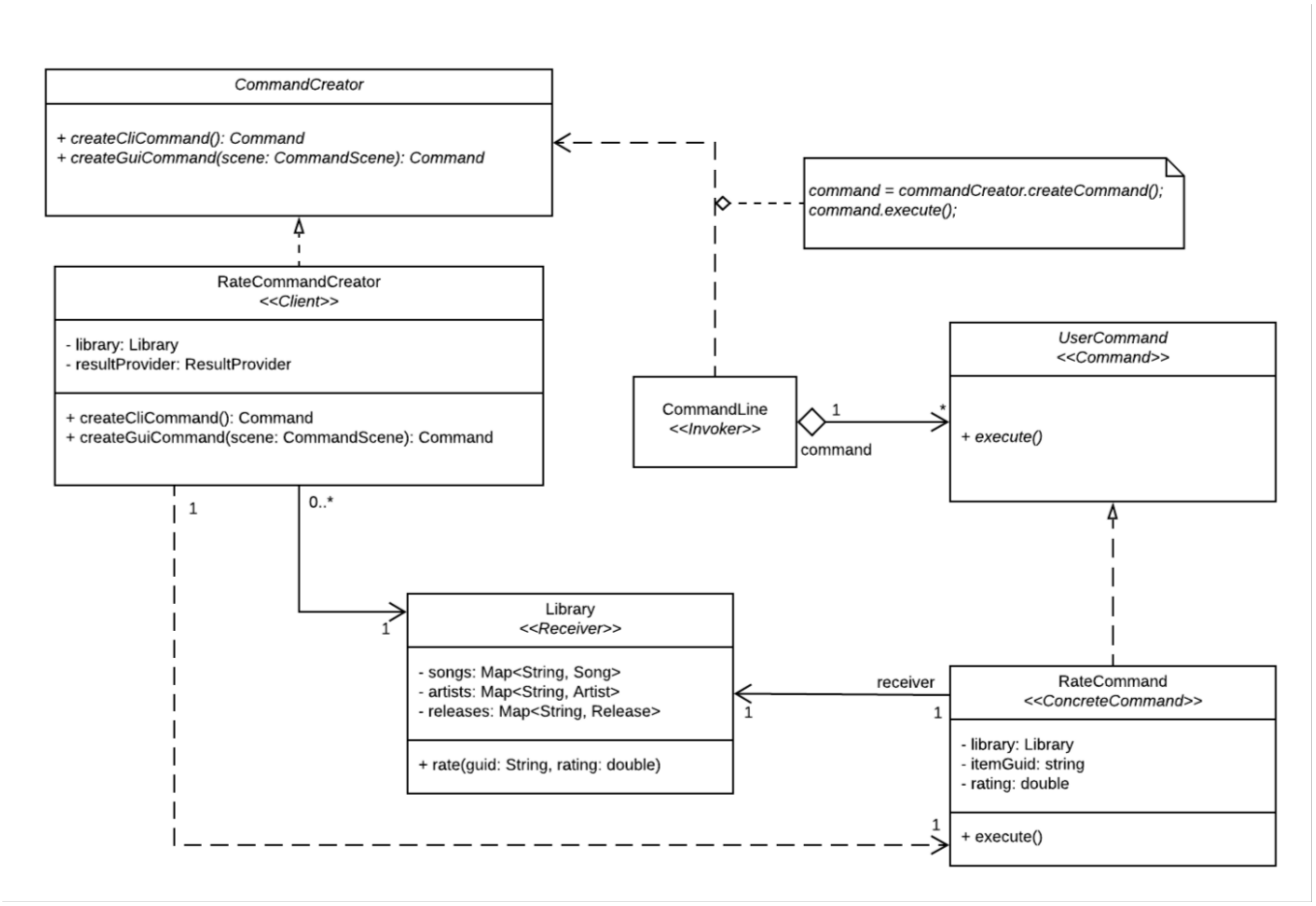

CommandCreator
+ createCliCommand(): Command
+ createGuiCommand(scene: CommandScene): Command
command = commandCreator.createCommand();
command.execute();
RateCommandCreator
<<Client>>
- library: Library
- resultProvider: ResultProvider
+ createCliCommand(): Command
+ createGuiCommand(scene: CommandScene): Command
CommandLine
<<Invoker>>
1
*
command
UserCommand
<<Command>>
+ execute()
1
0..*
1
Library
<<Receiver>>
- songs: Map<String, Song>
- artists: Map<String, Artist>
- releases: Map<String, Release>
+ rate(guid: String, rating: double)
receiver
1
1
RateCommand
<<ConcreteCommand>>
- library: Library
- itemGuid: string
- rating: double
1
+ execute()

Figure: Class diagram for the Command pattern, as pertaining to a command line request to rate an item in the library. Other commands follow the same structure, but with different implementations of the *CommandCreator* and *UserCommand* interfaces.

Role

The Command pattern is essential to the system's implementation of user commands. All user commands (searching, rating, requesting help, etc.) exist as Concrete Commands within the pattern's structure. When the user requests a certain command, the relevant Client (a *CommandCreator* object) creates and returns the command to the Invoker. All needed materials, including the Receiver object, are supplied to the Command through the Client when it is instantiated. The Command object's *execute()* command is called, thus dispatching the command.

Design Considerations

There are many possible user requests, and they perform very different functions from one another. It was decided that all commands can be represented by the *UserCommand* interface, despite the wide range of behaviors. This way, the nature of a given command is always hidden from the invoker, and all user query commands are handled uniformly. Different Concrete Commands require different resources to be effective (*ResultProvider, Library, Database,* etc.). The invoker, being the information expert on these resources, supplies them to the *CommandCreator* upon its instantiation.

Command classes tend to be coupled with several other classes in order to perform their actions. This is mitigated by using the *ResultProvider* (for commands that only need to access the results) and *ResultHandler* (for commands that only need to set the results) instead of the *ResultManager* class itself.

Class	**Role in GoF pattern**	**Participant's contribution in the context of the application**
CommandLine	Invoker	Processes a user's request for a certain command through the command line interface. Initiates the process of Command creation. Invokes Commands once created.
Gui	Invoker	When the GUI is being used to make a query (through a button press, etc.), processes the request through the Client. Invokes Commands once created.
CommandCreator	Client	Orchestrates the creation of a particular Command. Defines abstract methods for creating command line (Cli) or GUI commands. For CLI commands, the user is prompted for all parameters before the Command object is created. Returns the created Concrete Command object to the Invoker.
Library	Receiver	Provides interfaces needed by commands which utilize or operate upon the user's library (rating, searching, listing, etc.). Used by Commands to retrieve items of a given type belonging to the library.
Database	Receiver	Provides interfaces needed by commands which utilize a database (searching commands). Stores music items belonging to a database. Used by Commands to retrieve music items of a given type from a database.
UserCommand	Command	Describes all user commands. Provides a common interface, used by the Invoker, to execute commands. Hides the implementation and type of a particular command from the Invoker.
RateCommand	Concrete Command	When executed, assigns a rating to an item in the user's library. Invoked by the Invoker after the user requests to rate an item.

ListCommand	Concrete Command	When executed, provides the list of artists whose music resides in the user's library. If in command line mode, formats and prints the list to the command line.
LibrarySearchReleaseCommand	Concrete Command	When executed, searches the library for releases matching the specified criteria. Results are passed to the ResultHandler that was supplied by the Client.
LibrarySearchArtistCommand	Concrete Command	When executed, searches the library for artists matching the specified criteria. Results are passed to the ResultHandler that was supplied by the Client.
LibrarySearchSongCommand	Concrete Command	When executed, searches the library for songs matching the specified criteria. Results are passed to the ResultHandler that was supplied by the Client.
DatabaseSearchReleaseCommand	Concrete Command	When executed, searches the current database for releases matching the specified criteria. Results are passed to the ResultHandler that was supplied by the Client.
DatabaseSearchArtistCommand	Concrete Command	When executed, searches the current database for artists matching the specified criteria. Results are passed to the ResultHandler that was supplied by the Client.
DatabaseSearchSongCommand	Concrete Command	When executed, searches the current database for songs matching the specified criteria. Results are passed to the ResultHandler that was supplied by the Client.
ExploreCommand	Concrete Command	When executed (and the most recent item results are of artists in the library), the artists' releases and other songs are listed for the user. If another ExploreCommand is executed while in this state, the corresponding release's songs will be displayed for the user.
AddCommand	Concrete Command	When executed, adds the item corresponding to a search result to the user's library. The identifier of the search result to add is specified when the command is created through the Builder pattern.
RemoveCommand	Concrete Command	When executed, removes the item corresponding to a search result from the user's library. The identifier of the search result to add is specified when the command is created through the Builder pattern.
HelpCommand	Concrete Command	When executed, displays a help message instructing the user on the use of the application.
BackCommand	Concrete Command	When executed (and the user is currently exploring an artist in the library), the list of artists in the library will be displayed to the user again.
LoginCommand	Concrete Command	When executed with a username and password, checks for a matching username and password in the user database. If the credentials match, the user is logged in and taken to the home screen

CreateAccountCommand	Concrete Command	When executed with a password and a username that is not already in the database, a new account is created in the user database with these credentials.

Deviations from the standard pattern: The Invoker creates a CommandCreator object, which creates and acts as the Client for the Command. The Library reference is passed to the CommandCreator upon instantiation.

Requirements being covered:

- The user may submit various types of queries to the system
- Queries may be submitted from the GUI or command line interface
- A given query will have access to the necessary information for it to be carried out

Design Pattern: Strategy

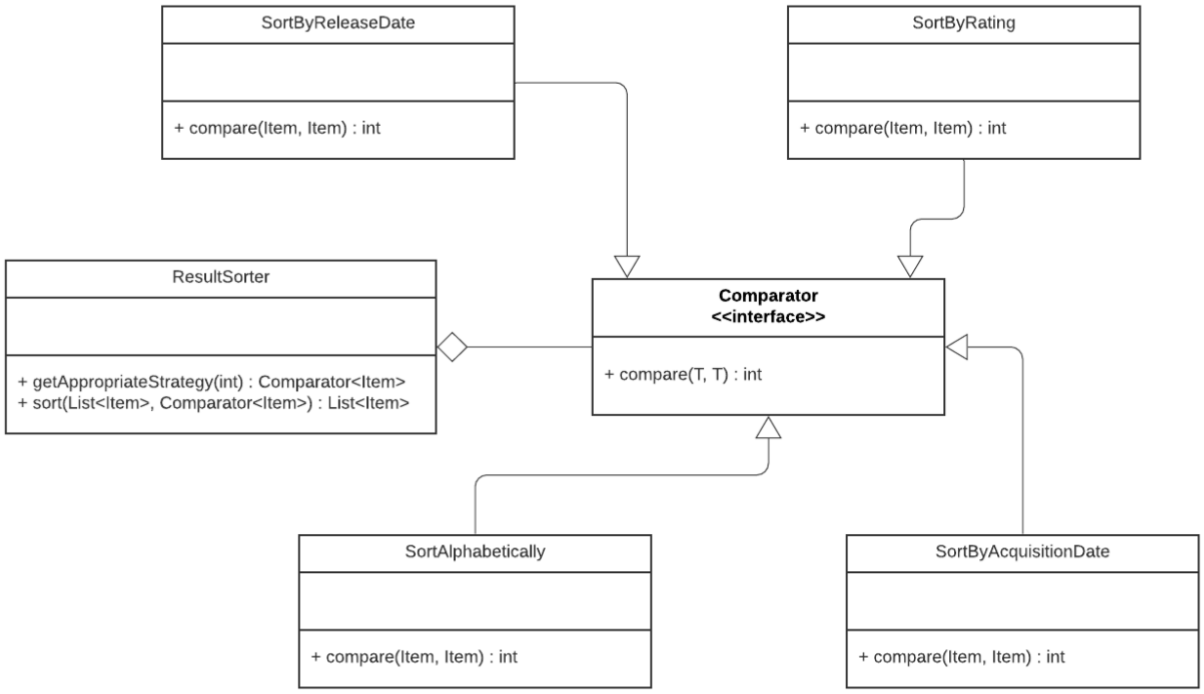

SortByReleaseDate
+ compare(Item, Item) : int
SortByRating
+ compare(Item, Item) : int
ResultSorter
+ getAppropriateStrategy(int) : Comparator<Item>
+ sort(List<Item>, Comparator<Item>) : List<Item>
Comparator
<<interface>>
+ compare(T, T) : int
SortAlphabetically
+ compare(Item, Item) : int
SortByAcquisitionDate
+ compare(Item, Item) : int

Figure: The class diagram for the Strategy Pattern. The Comparator interface only shows the methods we used to simplify the diagram.

Role

The user must be able to sort various search results in different ways. A search for artists may be sorted alphabetically. A search for songs may be sorted alphabetically, by rating, or by acquisition date. A search for releases may be sorted alphabetically, by rating, by acquisition date or by the date of the release. Using the Strategy pattern gives us the ability to define the algorithms for sorting and make them interchangeable with one another therefore allowing them to be independent from the client. The strategy class has the role of declaring an interface that is common for the supported algorithms or the supported sorting methods in our case. Instead of creating our own interface we used the built in *Comparator* interface of Java and we use this to compare the individual results of a search by the user. The context class of the Strategy pattern is the *ResultSorter* class which uses the concrete strategy classes to properly sort one search result against another. Once the entire list of search results has been sorted it is returned to the client.

Design Considerations

Since artists, songs, and releases can be sorted in different ways using the Strategy pattern prevents us from having to implement each sorting algorithm into their respective classes differently. This also prevents the artist, song, and release classes from having too many responsibilities. Putting the sorting algorithms into interchangeable classes separates the concerns and lowers the coupling of the system. The Strategy pattern is the best choice for sorting because we need different variants of an algorithm. We could have had a class similar to the *ResultSorter* class for every sorting mechanism, each implementing how to sort artists, songs, and releases if allowed but using the Java *Comparator* interface as the common strategy makes this much simpler. Using the *compare(T, T)* method from Comparator, we can simply use the different attributes of the item in the method in order to get the desired result. For example, the *SortByRating* class only works for songs and releases so we pass in two of the search results and use the *getRating()* method to compare the doubles to sort properly. This is a much simpler approach than having to write different classes for sorting by rating of songs and artists.

Name: Strategy		**GoF pattern:** Strategy
Participants		
Class	**Role in GoF pattern**	Participant's contribution in the context of the application
ResultSorter	Context	The ResultSorter delegates sorting implementation to the Comparator Interface.
Comparator	Strategy	Comparator declares an interface common to all the sorting strategies
SortAlphabetically	ConcreteStrategy A	Implements the algorithm using the Comparator interface to sort songs, releases, and artists alphabetically.
SortByAcquisitionDate	ConcreteStrategy B	Implements the algorithm using the Comparator interface to sort releases and songs by acquisition date.
SortByRating	ConcreteStrategy C	Implements the algorithm using the Comparator interface to sort releases, artists and songs by rating.
SortByReleaseDate	ConcreteStrategy D	Implements the algorithm using the Comparator interface to sort releases, artists and songs by release date.
Deviations from the standard pattern: ResultSorter is a class instead of an interface. It contains case statements, which determine which sorting technique to implement.		
Requirements being covered: • Allows the user to sort search results of artists alphabetically. • Allows the user to sort search results of songs alphabetically, by rating, or by acquisition date. • Allows the user to sort search results of releases alphabetically, by rating, by acquisition date, or by the date of the release.		

Results

The Results subsystem dictates the interactions between user interfaces, the queries they invoke, and the generated results they display. The State pattern is used to manage the results of the most recent query. The Observer pattern is used to notify user interface views of new results, so that they can be displayed to the user.

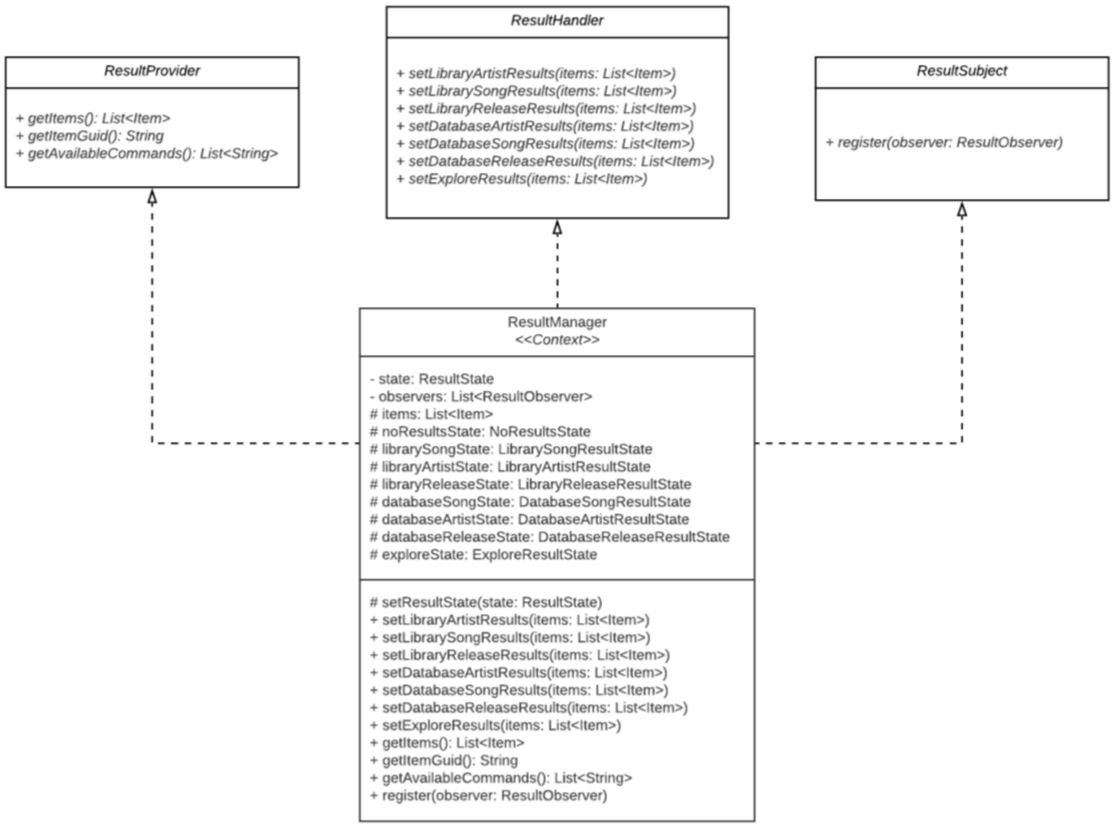

ResultProvider
+ getItems(): List<Item>
+ getItemGuid(): String
+ getAvailableCommands(): List<String>
ResultHandler
+ setLibraryArtistResults(items: List<Item>)
+ setLibrarySongResults(items: List<Item>)
+ setLibraryReleaseResults(items: List<Item>)
+ setDatabaseArtistResults(items: List<Item>)
+ setDatabaseSongResults(items: List<Item>)
+ setDatabaseReleaseResults(items: List<Item>)
+ setExploreResults(items: List<Item>)
ResultSubject
+ register(observer: ResultObserver)
ResultManager
<<Context>>
- state: ResultState
- observers: List<ResultObserver>
# items: List<Item>
# noResultsState: NoResultsState
# librarySongState: LibrarySongResultState
# libraryArtistState: LibraryArtistResultState
# libraryReleaseState: LibraryReleaseResultState
# databaseSongState: DatabaseSongResultState
# databaseArtistState: DatabaseArtistResultState
# databaseReleaseState: DatabaseReleaseResultState
# exploreState: ExploreResultState
# setResultState(state: ResultState)
+ setLibraryArtistResults(items: List<Item>)
+ setLibrarySongResults(items: List<Item>)
+ setLibraryReleaseResults(items: List<Item>)
+ setDatabaseArtistResults(items: List<Item>)
+ setDatabaseSongResults(items: List<Item>)
+ setDatabaseReleaseResults(items: List<Item>)
+ setExploreResults(items: List<Item>)
+ getItems(): List<Item>
+ getItemGuid(): String
+ getAvailableCommands(): List<String>
+ register(observer: ResultObserver)

Figure: The *ResultManager* implements 3 interfaces. For particular usages of its behaviors, the appropriate interface is given to decouple *ResultManager* from clients. *ResultHandler* is used to set the list of results. *ResultSubject* is used to register an observer of the results. *ResultProvider* is used to obtain the list of results.

Design Pattern: Observer

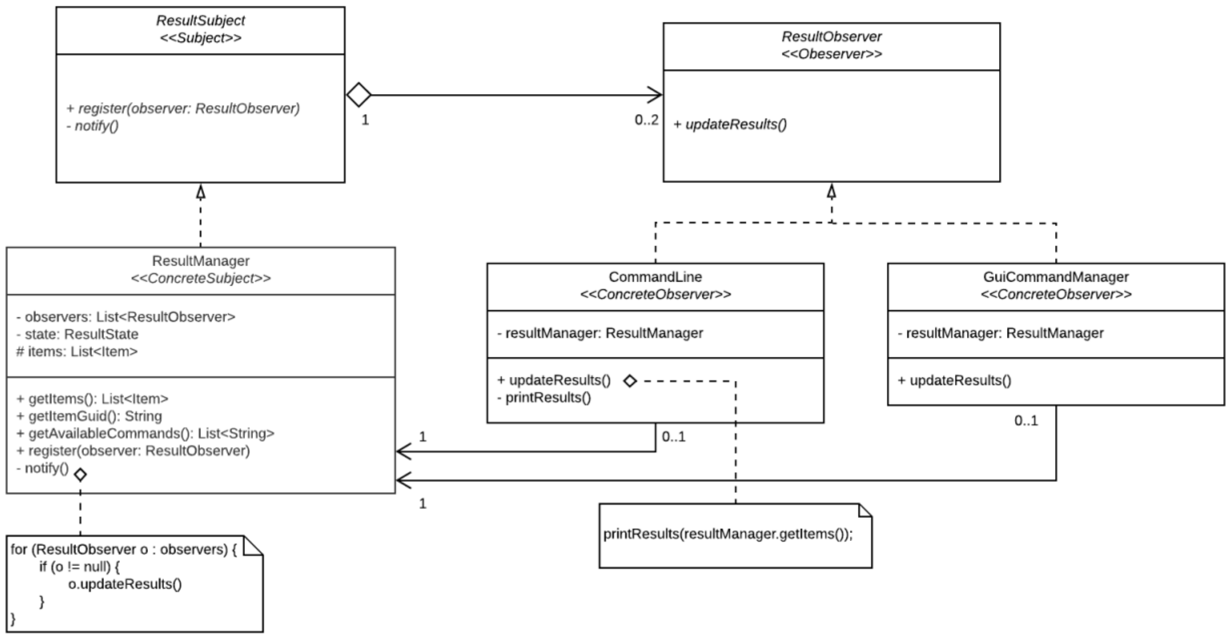


Figure: The structure of the Observer pattern.

Role

The Observer pattern is used here to handle the distribution of query results to active user interfaces.

When results are generated by an executed command, the GUI and the command line interface should be notified so that these results can be displayed in both formats. This ensures that both interfaces are always up-to-date with the most recent query results. Thus, the user may switch between the two interfaces at any time.

Result-producing commands share their results with the *ResultManager* object (the Concrete Subject in the pattern's context). Upon receiving results, it is desirable to have both user interfaces be notified through the same mechanism.

The Observer pattern allows for the sharing of results with both user interfaces at any time, through the same program interface. The user interfaces' controlling classes are registered as observers of the *ResultSubject* object. When new results are received by this Subject, it notifies all registered observers through the uniform *ResultObserver* interface's *updateResults()* method.

In the Concrete Subjects, the invocation of this method triggers user interface-specific behavior for handling new query results. In the command line interface, this manifests as the printing of results and the prompt for a new command. In the GUI, this takes the form of displaying results graphically for the user.

Design Considerations

The use of this pattern promotes extensibility following the open/closed principle. Suppose a new user interface object (either a new Concrete Observer or another instance of an existing Concrete Observer). The Subject and Observer interfaces will require no modification. The new Concrete Observer need only implement the *ResultObserver* interface and call the *ResultSubject*'s *register(...)* method when instantiated.

There is the question of whether the pattern implementation should follow a "push" or "pull" mentality of providing results to Concrete Observers. The "Gang of Four "-given structure for this pattern specifies that the Concrete Observers maintain state corresponding to the Concrete Subject's state. While the Concrete Observers require access to the *ResultManager's* list of result items (through *getItems()* or *getItemGuid(...)*), they also require access to the *getAvailableCommands()* method. The returned values of these methods may not be needed right away when the *updateResults()* method is called. Also, *getAvailableCommands()* does not provide static state data -- it provides dynamic behavior that is heavily tied to the updating of results. Therefore, it was deemed appropriate to simply notify the Concrete Observers of the presence of new results. The Concrete Observers then have the freedom of deciding when to "pull" information from the Concrete Subject.

If an Observer object no longer exists when the Subject is attempting to notify it, this causes a program error and likely failure. This may occur if, say, the GUI window is force closed. The GUI then does not have the chance to notify the Subject that it is being deleted. Thus, the *notify()* method does a null check on each observer before notifying it.

Class	**Role in GoF pattern**	**Participant's contribution in the context of the application**
ResultSubject	Subject	Defines the interface for registering and notifying observers. This reusable interface may be implemented by any class that produces results and desires to inform observers of them.
ResultObserver	Observer	Defines the interface for notifying an observer of new results in the Subject. It includes a single abstract method *updateResults()*. It requires that the Concrete Observer "pulls" the results from the Concrete Subject.
ResultManager	Concrete Subject	The primary implementation of the ResultSubject interface. Implements the tracking and notifying of observers. Also provides public methods *getAvailableCommands()* and *getItems()*, which are used by Concrete Observers to "pull" the updated state once notified.
CommandLine	Concrete Observer	Represents the command line interface class that uses the results and list of available commands to accept and execute user queries. When its *updateResults()* method is called, it pulls the results and displays them, and then it uses the list of

		available commands to prompt the user and validate their next query.
Gui	Concrete Observer	Represents the GUI class that displays results and presents the user with graphical options to access available commands. When its *updateResults()* method is called, it pulls the results and displays them, then modifies the controls for command selection to reflect the current list of available commands.
Deviations from the standard pattern: The Concrete Observers do not maintain direct references to the Concrete Subject's state. Instead, they maintain a reference only to the Concrete Subject itself. Operations using said state are performed on demand through the ResultManager reference. The *updateResults()* method is moreso used to notify the Concrete Observer that the state has changed.		
Requirements being covered: • When generated, the results of queries are displayed to the user. • Results are displayed in the location currently being used by the user (command line or GUI) • A list of recent query results is maintained for use by subsequent commands		

Design Pattern: State

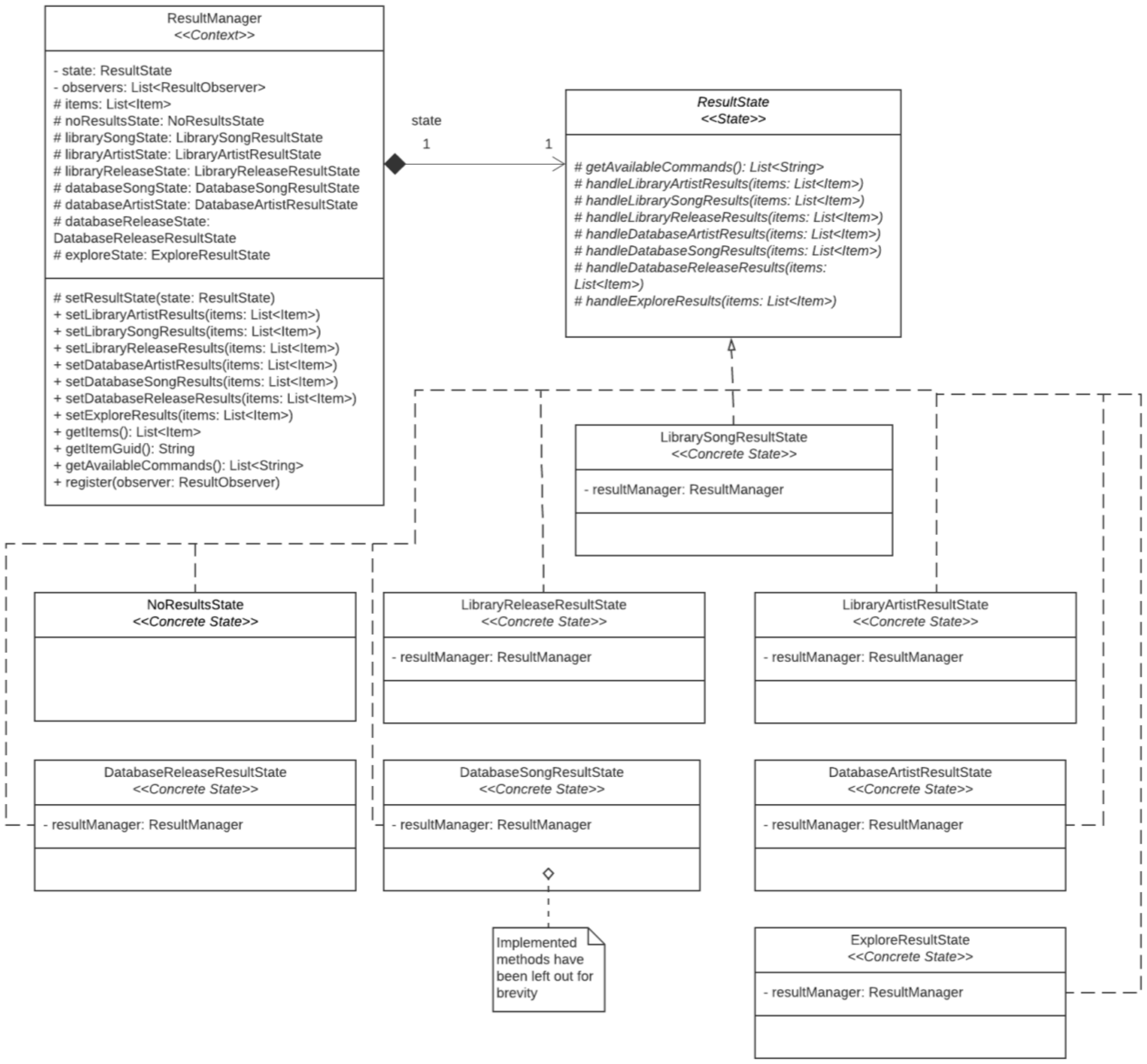

ResultManager
<<Context>>
- state: ResultState
- observers: List<ResultObserver>
# items: List<Item>
# noResultsState: NoResultsState
# librarySongState: LibrarySongResultState
# libraryArtistState: LibraryArtistResultState
# libraryReleaseState: LibraryReleaseResultState
# databaseSongState: DatabaseSongResultState
# databaseArtistState: DatabaseArtistResultState
# databaseReleaseState: DatabaseReleaseResultState
# exploreState: ExploreResultState
# setResultState(state: ResultState)
+ setLibraryArtistResults(items: List<Item>)
+ setLibrarySongResults(items: List<Item>)
+ setLibraryReleaseResults(items: List<Item>)
+ setDatabaseArtistResults(items: List<Item>)
+ setDatabaseSongResults(items: List<Item>)
+ setDatabaseReleaseResults(items: List<Item>)
+ setExploreResults(items: List<Item>)
+ getItems(): List<Item>
+ getItemGuid(): String
+ getAvailableCommands(): List<String>
+ register(observer: ResultObserver)
state
1
1
ResultState
<<State>>
# getAvailableCommands(): List<String>
# handleLibraryArtistResults(items: List<Item>)
# handleLibrarySongResults(items: List<Item>)
# handleLibraryReleaseResults(items: List<Item>)
# handleDatabaseArtistResults(items: List<Item>)
# handleDatabaseSongResults(items: List<Item>)
# handleDatabaseReleaseResults(items: List<Item>)
# handleExploreResults(items: List<Item>)
LibrarySongResultState
<<Concrete State>>
- resultManager: ResultManager
NoResultsState
<<Concrete State>>
LibraryReleaseResultState
<<Concrete State>>
- resultManager: ResultManager
LibraryArtistResultState
<<Concrete State>>
- resultManager: ResultManager
DatabaseReleaseResultState
<<Concrete State>>
- resultManager: ResultManager
DatabaseSongResultState
<<Concrete State>>
- resultManager: ResultManager
DatabaseArtistResultState
<<Concrete State>>
- resultManager: ResultManager
Implemented methods have been left out for brevity
ExploreResultState
<<Concrete State>>
- resultManager: ResultManager

Figure: The class diagram for the State pattern.

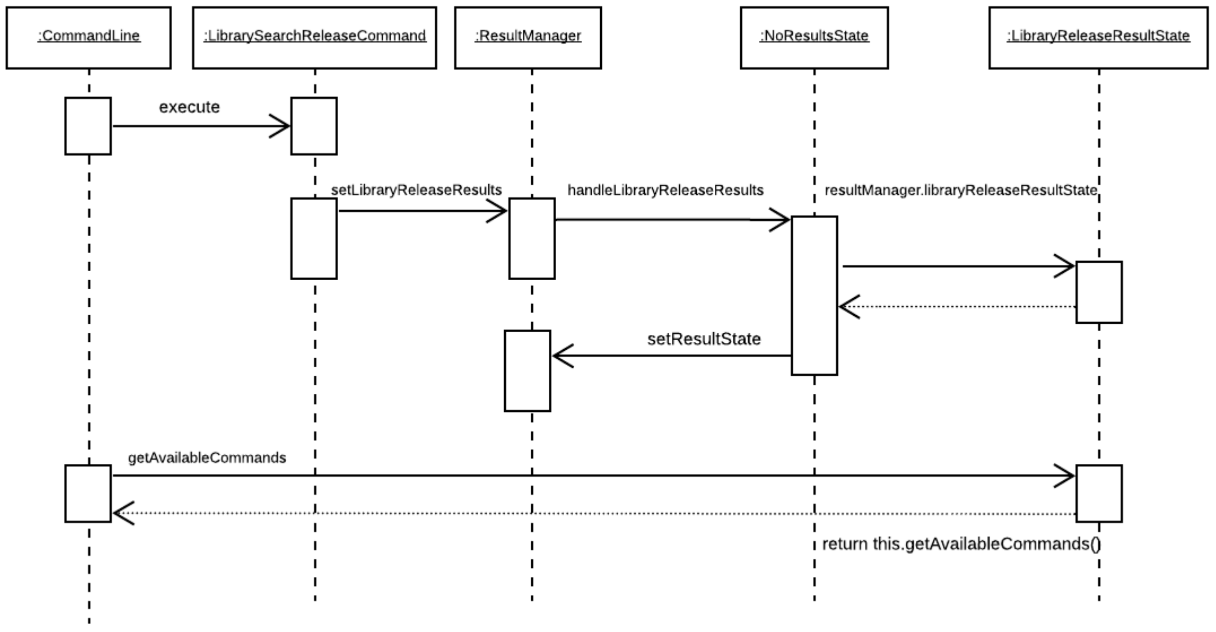

:CommandLine
:LibrarySearchReleaseCommand
:ResultManager
:NoResultsState
:LibraryReleaseResultState
execute
setLibraryReleaseResults
handleLibraryReleaseResults
resultManager.libraryReleaseResultState
setResultState
getAvailableCommands
return this.getAvailableCommands()

Figure: Sequence diagram for a changing of the state after receiving new results from a search for releases in the library.

Role

Depending on the nature of the most recent search or explore results, the list of available user commands changes. Using the State pattern effectively solves the problem of determining the list of valid commands at any given time. To retrieve this list, a client with a reference to the Context object may invoke *getAvailableCommands()* to retrieve the list of user commands that are valid from the current state. Additionally, the Context object stores the list of results. The Concrete States reference this list when deciding which commands are currently valid. Localizing this dynamic behavior to each Concrete State implementation separates the concern of determining available commands, and the logic used to generate the list can be changed at any time using the interface provided by the Context. The State pattern promotes extensibility through the Open/Closed Principle (this is explained in the following section).

Design Considerations

The current state represents the nature of the most recently received search results. For example, receiving the results of an artist search in the library will generate a transition to the LibraryArtistResultState. Most of the state transitions are shared among states – the same event tends to generate a transition to the same state, no matter the current state. State transitions are defined in implementations of the ResultState interface through the *handleX(…)* set of methods. In the current release, this leads to redundancy among implementations. The same code will be used for these methods in most implementations.

The alternative solution is to remove the *handleX(…)* methods from the State participant, moving transition logic to the Context. This solution would reduce redundancy at the cost of flexibility and extensibility. Say, for example, that receiving “explore” results while in the *ExploreResultState* must now transition the system to a new *NestedExploreResultState* object. This new Concrete State may be created, and the *ExploreResultState*’s *handleExploreResults(...)* method is modified to transition to this new state. This way, the Context remains untouched. If the Context handles state transitions, then modification would be necessary.

An instance of each Concrete State is created up-front in the Context object. State transitions happen frequently because all search commands generate a transition. When returning to a previous state, the Concrete State object needs no new information. Thus, the current Concrete State may utilize the Context’s instance of the new state instead of creating a new object.

results.setResultState(results.librarySongState);

The *getAvailableCommands()* method returns a list of the user commands available from the current state. This includes dynamic behavior. For example, the “rate” command will not be included after a library song search if the search returned no results (*results.items.size() == 0*). The list of available commands is highly dependent on the current state, so this was deemed the appropriate location for this logic. The Concrete State classes are not entirely cohesive since they provide two functions (listing available commands and deciding state transitions). This was deemed an acceptable tradeoff for the flexibility mentioned above.

Class	**Role in GoF pattern**	**Participant's contribution in the context of the application**
ResultManager	Context	Stores the most recent search or explore results. Provides interfaces for storing/retrieving results as well as retrieving the list of available commands.
ResultState	State	Defines the list of available commands for a given state. Updates the Context’s internal state upon receiving new results. Provides a common interface for all Concrete States. Hides the implementation and type of the current state from the Context.
NoResultsState	Concrete State	Defines the list of available commands before any result-producing command has been executed. Handles the Context’s state transitions upon receiving new results while in this state.
LibraryReleaseResultState	Concrete State	Defines the list of available commands when the most recent search results are from a release search in the library. Handles the Context’s state transitions upon receiving new results while in this state.
LibrarySongResultState	Concrete State	Defines the list of available commands when the most recent search results are from a song search in the library. Handles the Context’s state transitions upon receiving new results while in this state.
LibraryArtistResultState	Concrete State	Defines the list of available commands when the most recent search results are from an artist search in the library. Handles the Context’s state transitions upon receiving new results while in this state.

DatabaseReleaseResultState	Concrete State	Defines the list of available commands when the most recent search results are from a release search in the database. Handles the Context's state transitions upon receiving new results while in this state.
DatabaseSongResultState	Concrete State	Defines the list of available commands when the most recent search results are from a song search in the database. Handles the Context's state transitions upon receiving new results while in this state.
DatabaseArtistResultState	Concrete State	Defines the list of available commands when the most recent search results are from an artist search in the database. Handles the Context's state transitions upon receiving new results while in this state.
ExploreResultState	Concrete State	Defines the list of available commands when the most recent results are from the "explore" command. Handles the Context's state transitions upon receiving new results while in this state.

Deviations from the standard pattern: The variation wherein Concrete States dictate state transitions is used.

Requirements being covered:
- Recent query results are maintained for use by subsequent commands
- Only valid commands will be accepted by the application
- The user may obtain a list of available commands

GUI

- The GUI makes it easy for users to be able to navigate the Muze Music Library System. It provides a user friendly experience that makes it easier to understand. The GUI handles different elements of the system such as searching, user library, web database, and an account system. The GUI acts like the skin, something the user only sees, but the underlying functionality is handled elsewhere. The user will enter input into GUI elements such as text fields for searching. Once entered, it will be handled and processed by the GUI interface. From there, the query will be processed outside the GUI, and then return to the GUI to display to the user either the results, or options of their input query.

<u>GUI Class Diagram</u>

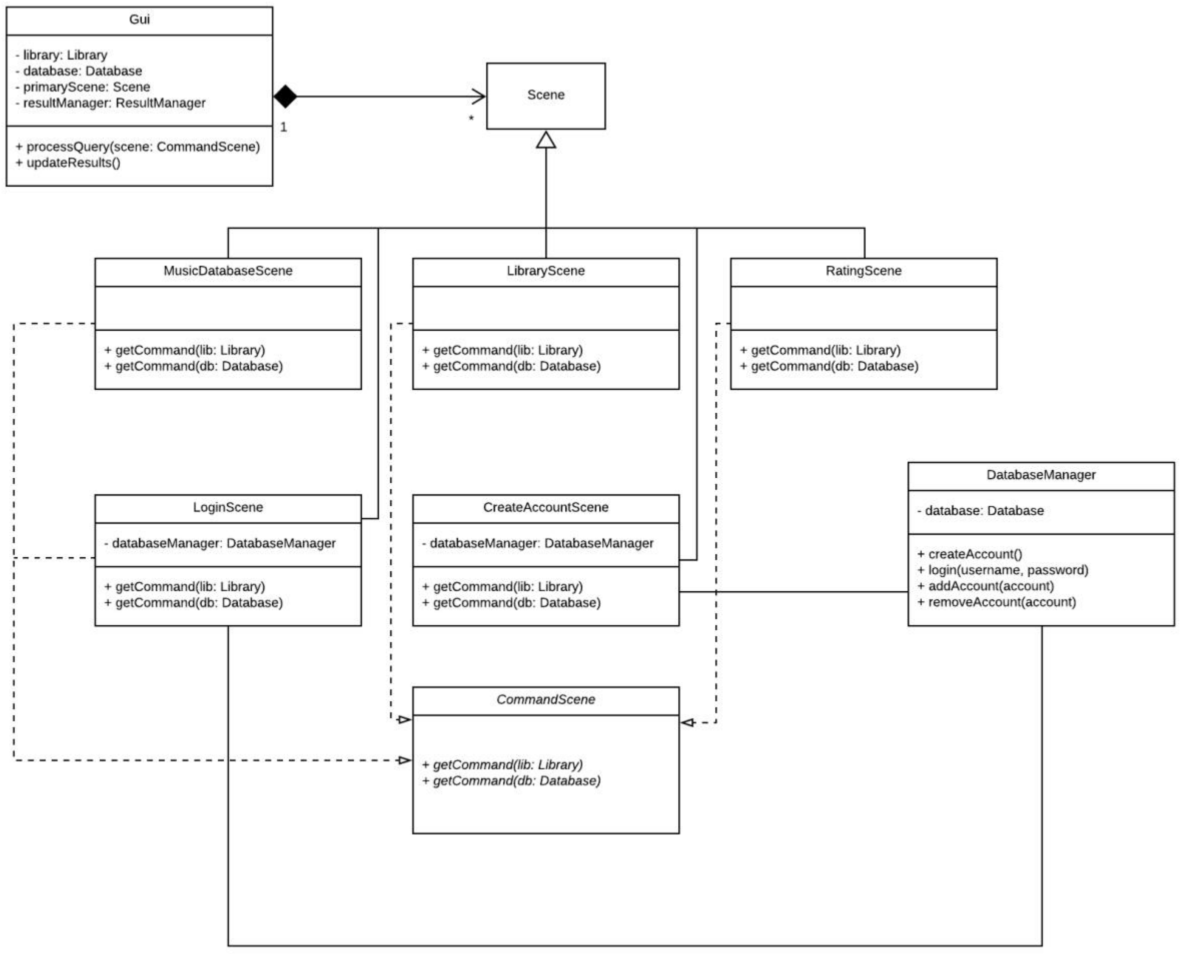
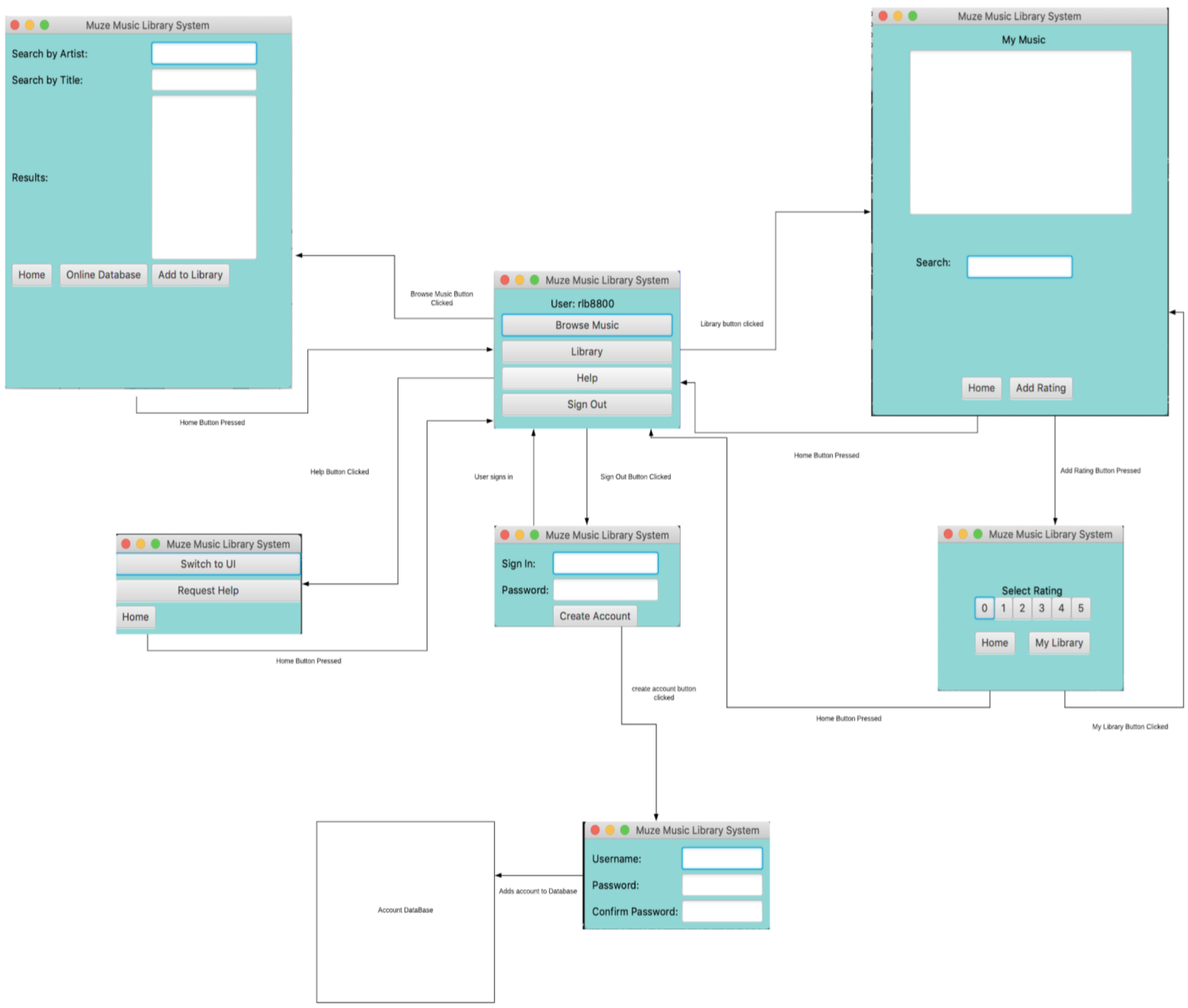
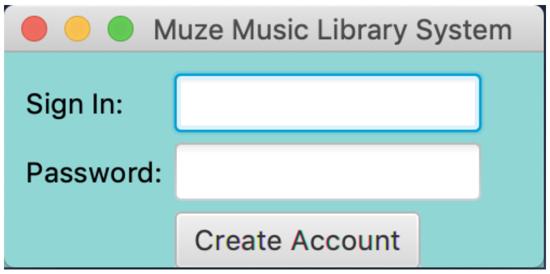

Gui
- library: Library
- database: Database
- primaryScene: Scene
- resultManager: ResultManager
+ processQuery(scene: CommandScene)
+ updateResults()
1
*
Scene
MusicDatabaseScene
+ getCommand(lib: Library)
+ getCommand(db: Database)
LibraryScene
+ getCommand(lib: Library)
+ getCommand(db: Database)
RatingScene
+ getCommand(lib: Library)
+ getCommand(db: Database)
LoginScene
- databaseManager: DatabaseManager
+ getCommand(lib: Library)
+ getCommand(db: Database)
CreateAccountScene
- databaseManager: DatabaseManager
+ getCommand(lib: Library)
+ getCommand(db: Database)
DatabaseManager
- database: Database
+ createAccount()
+ login(username, password)
+ addAccount(account)
+ removeAccount(account)
CommandScene
+ getCommand(lib: Library)
+ getCommand(db: Database)

Figure: The class structure diagram for the GUI

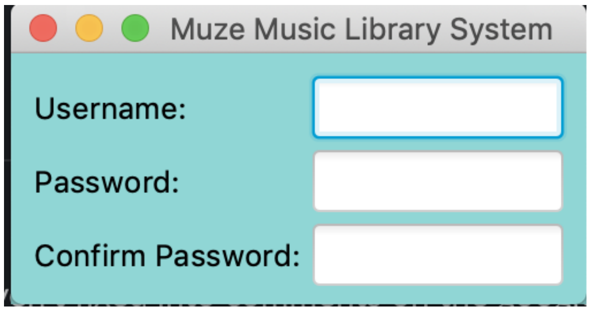
<u>GUI State Chart</u>



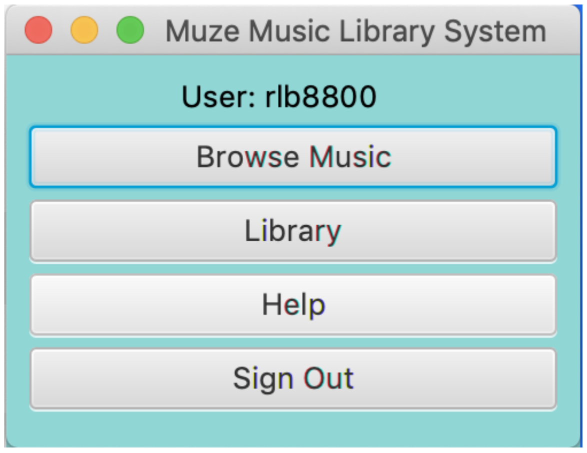
- On the initial startup the user will be taken to a login screen. If the user doesn't have an account, they can create one. Once a user enters a username and password, it will be verified against the account database to make sure the credentials are valid.



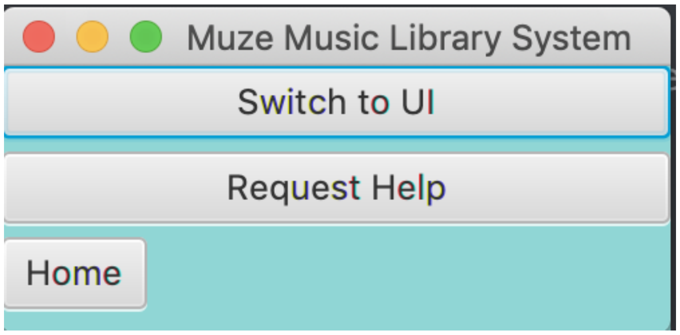
- If the user selects to create an account, they can do so here. Once the fields are submitted, the account database will make sure the account doesn't exist. If it doesn't it will then add the account to the database and the user will be asked to login.



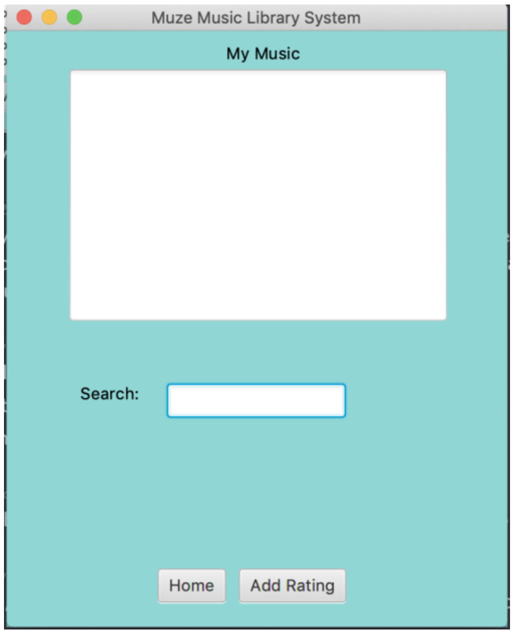
- Once the user has logged in, the user will be brought to the homescreen. From there the user can select a variety of options.



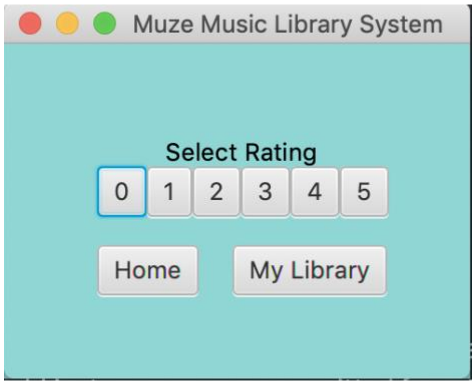
- If the user presses the help button, they will be brought to the help menu. From there, the user can learn about how to format queries for searching, as well as switch from the GUI to the UI.



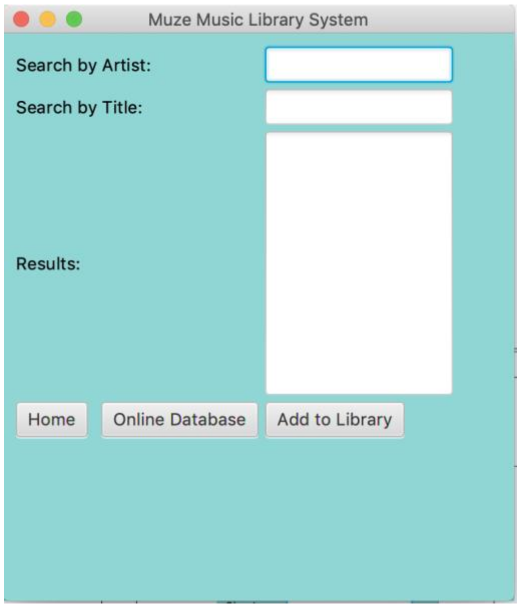
- If the user clicked the Library button, they will be taken to their personal Library. From here they can search their personal Library for songs, artists, and releases. They can also select to add a rating.



- If a user decided to rate a song, release, or artist. they can rate it from 0-5 stars.



- From the homescreen, if the user decides to search for music, they will be brought here. Here they have the option to either search the local database, or the online database. Despite the database being used, the search syntax will be the same. The user also has the option to add searched songs or artists to their library here.



Role:

The GUI uses separate classes for each scene that requires an input from the user. Each scene class extends the CommandScene. CommandScene will get the command the user inputs. Each class scene also implements the GUI interface, which can process user queries and update the GUI with the results. The interface is also responsible for changing scenes, and adding functionality to each scene.

Design Considerations:

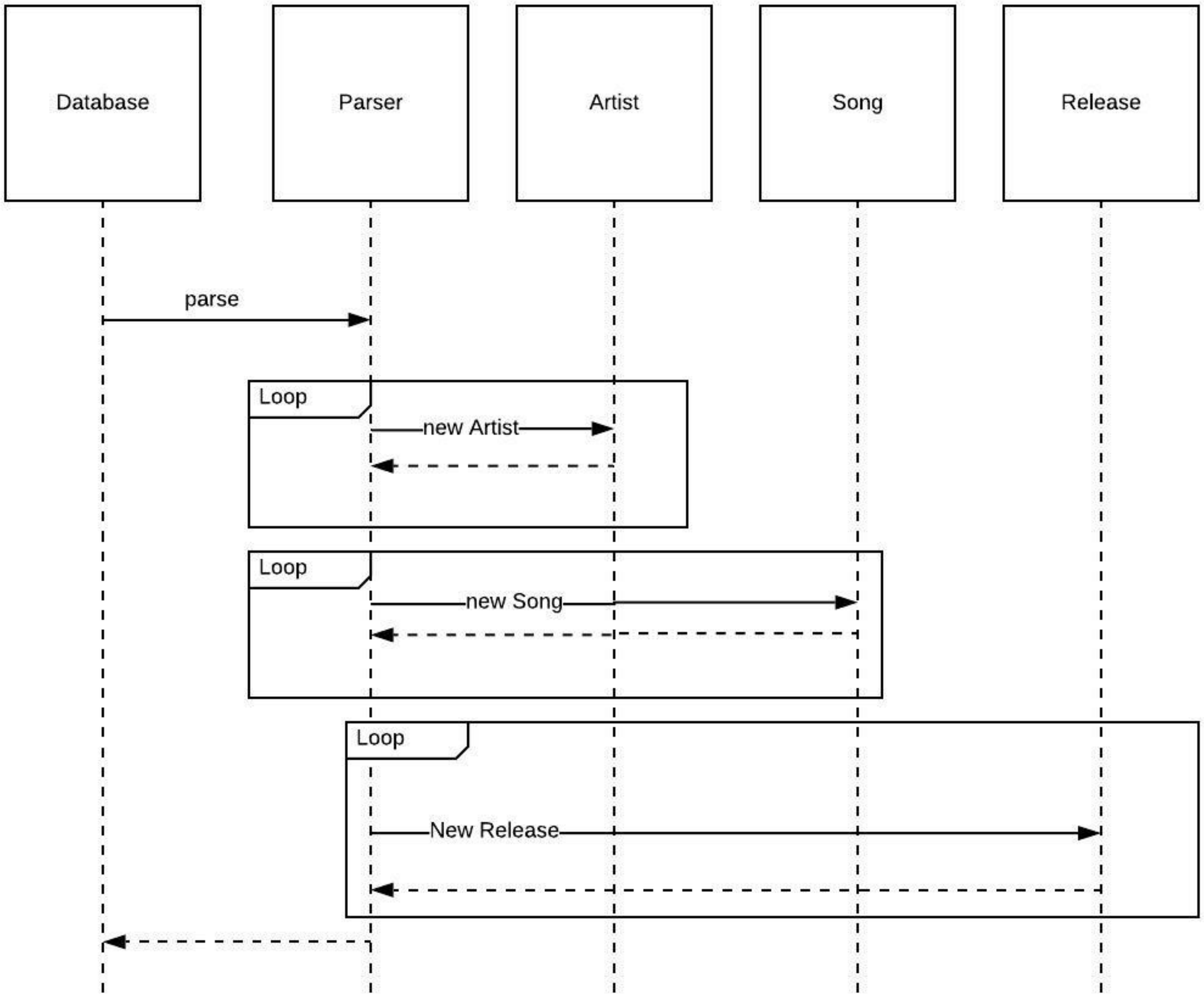
The GUI layout satisfies separation of concerns because it separates key features such as searching the database, searching the user's library, and rating into their own scenes. All elements that are required to search for example, are in the searching scene, whereas all elements required in the personal Library are handled in the personal Library scene. This makes it easier for the user to navigate as well as satisfying the design principle of separation of concerns.

Database

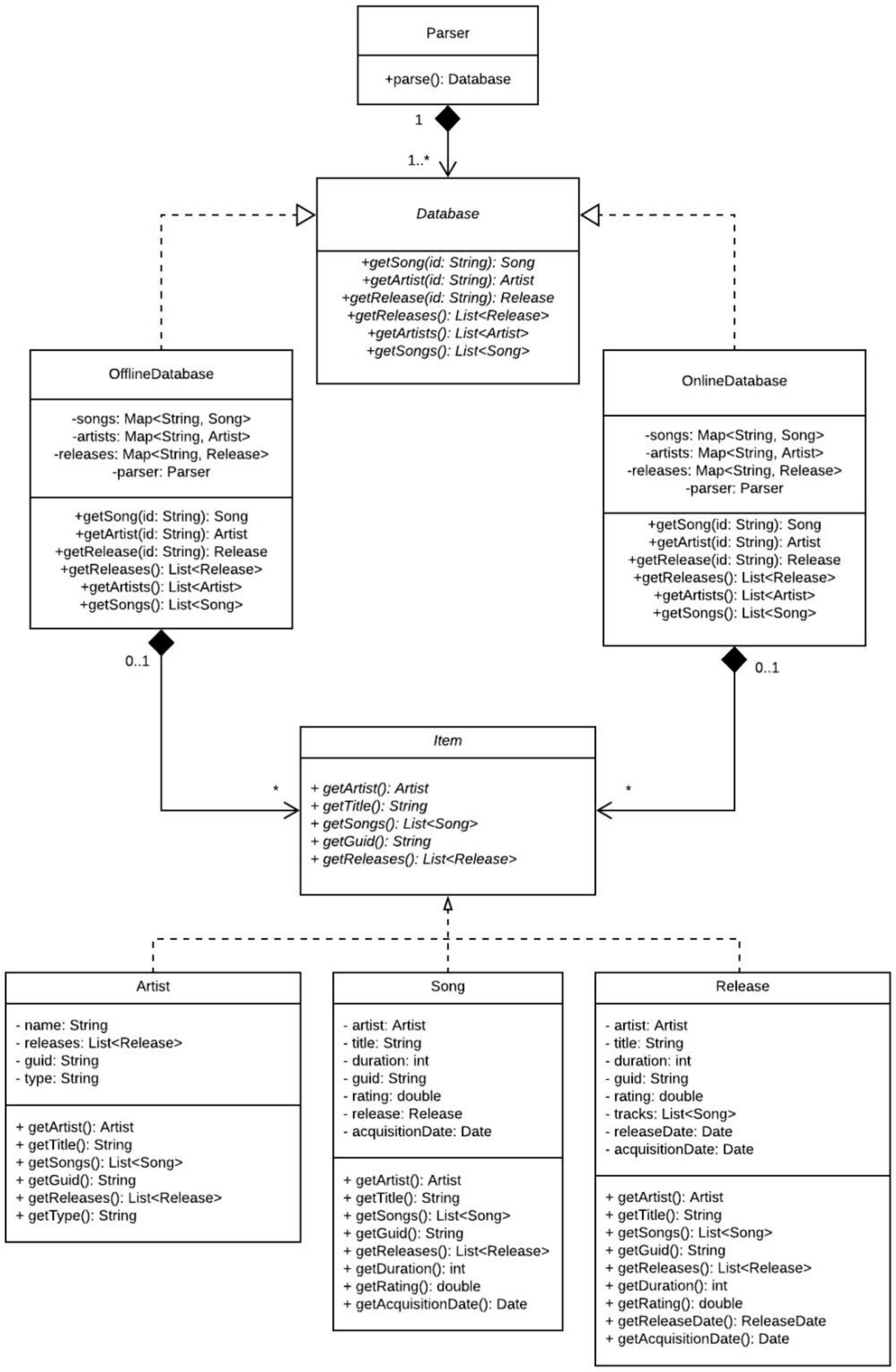
The database subsystem represents the general system pertaining to parsing, maintaining, and transmitting the offline and online databases. This system has three components:

- Parsing: Creating the offline database on start-up
- OfflineDatabase: Stores the database throughout the session
- OnlineDatabase: Contains the web-accessible database and dictates communication between it and the application

<u>Parsing Sequence Diagram</u>



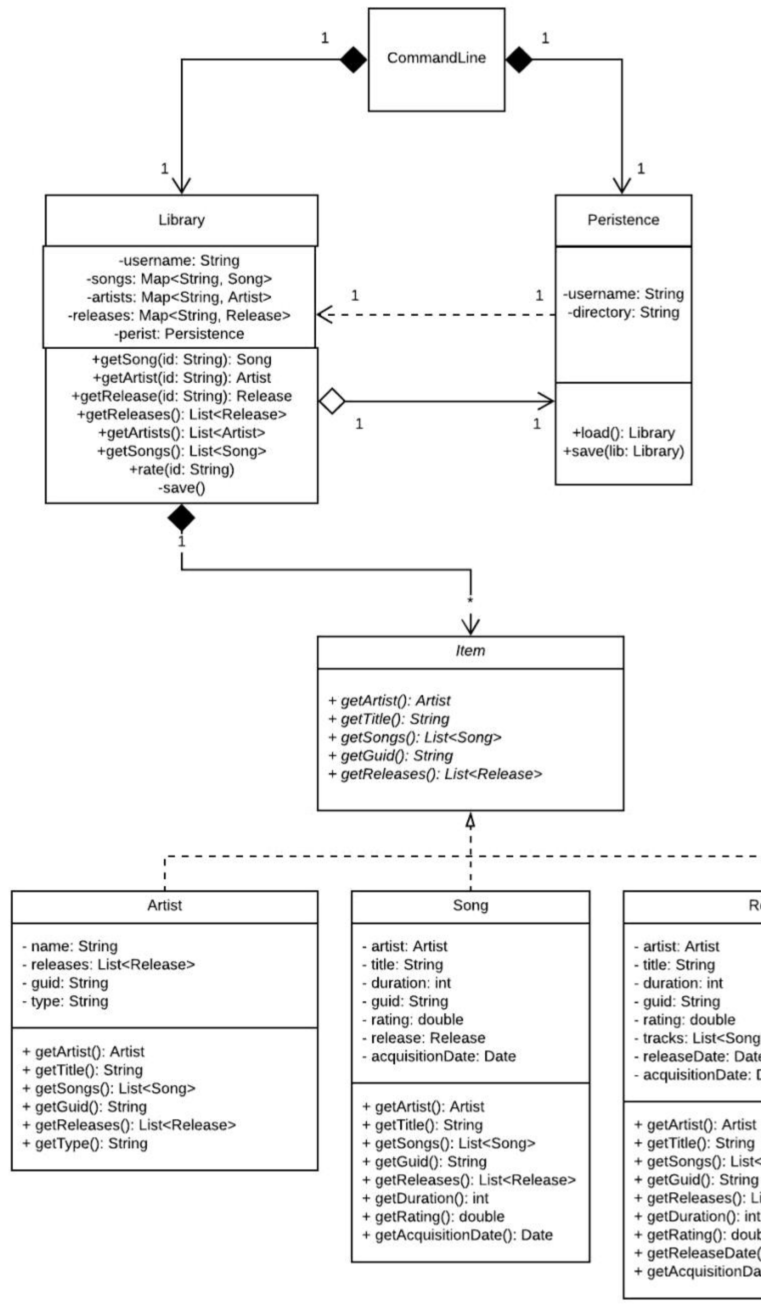
<u>Class Diagram</u>


Parser
+parse(): Database
1
1..*
Database
+getSong(id: String): Song
+getArtist(id: String): Artist
+getRelease(id: String): Release
+getReleases(): List<Release>
+getArtists(): List<Artist>
+getSongs(): List<Song>
OfflineDatabase
-songs: Map<String, Song>
-artists: Map<String, Artist>
-releases: Map<String, Release>
-parser: Parser
+getSong(id: String): Song
+getArtist(id: String): Artist
+getRelease(id: String): Release
+getReleases(): List<Release>
+getArtists(): List<Artist>
+getSongs(): List<Song>
OnlineDatabase
-songs: Map<String, Song>
-artists: Map<String, Artist>
-releases: Map<String, Release>
-parser: Parser
+getSong(id: String): Song
+getArtist(id: String): Artist
+getRelease(id: String): Release
+getReleases(): List<Release>
+getArtists(): List<Artist>
+getSongs(): List<Song>
0..1
0..1
*
*
Item
+ getArtist(): Artist
+ getTitle(): String
+ getSongs(): List<Song>
+ getGuid(): String
+ getReleases(): List<Release>
Artist
- name: String
- releases: List<Release>
- guid: String
- type: String
+ getArtist(): Artist
+ getTitle(): String
+ getSongs(): List<Song>
+ getGuid(): String
+ getReleases(): List<Release>
+ getType(): String
Song
- artist: Artist
- title: String
- duration: int
- guid: String
- rating: double
- release: Release
- acquisitionDate: Date
+ getArtist(): Artist
+ getTitle(): String
+ getSongs(): List<Song>
+ getGuid(): String
+ getReleases(): List<Release>
+ getDuration(): int
+ getRating(): double
+ getAcquisitionDate(): Date
Release
- artist: Artist
- title: String
- duration: int
- guid: String
- rating: double
- tracks: List<Song>
- releaseDate: Date
- acquisitionDate: Date
+ getArtist(): Artist
+ getTitle(): String
+ getSongs(): List<Song>
+ getGuid(): String
+ getReleases(): List<Release>
+ getDuration(): int
+ getRating(): double
+ getReleaseDate(): ReleaseDate
+ getAcquisitionDate(): Date

User Profile

This subsystem controls all processes related to storing, saving, and loading the users personalized data. All of the users library data is saved after every modification. The library is loaded and unloaded based when users log in. Finally, the library component is the object that stores the users information, and persists the users data.

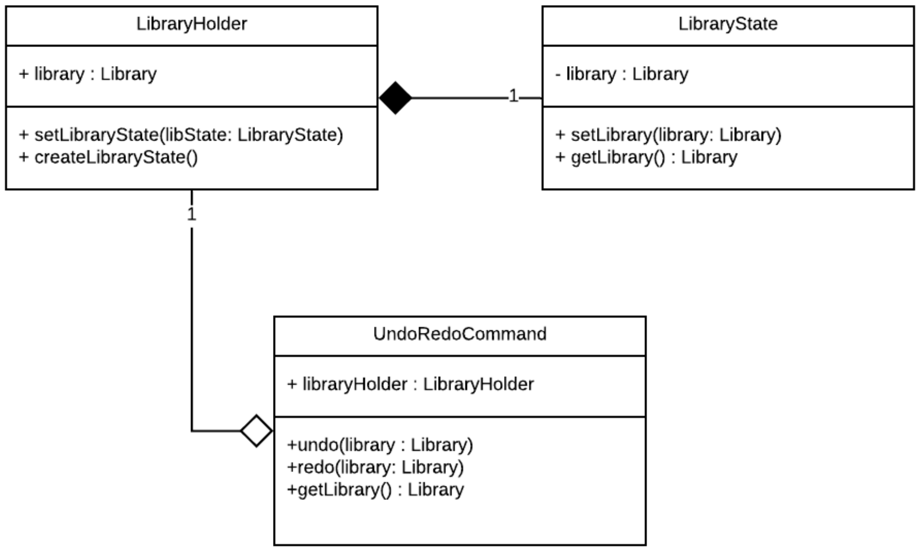
UML Class Diagram



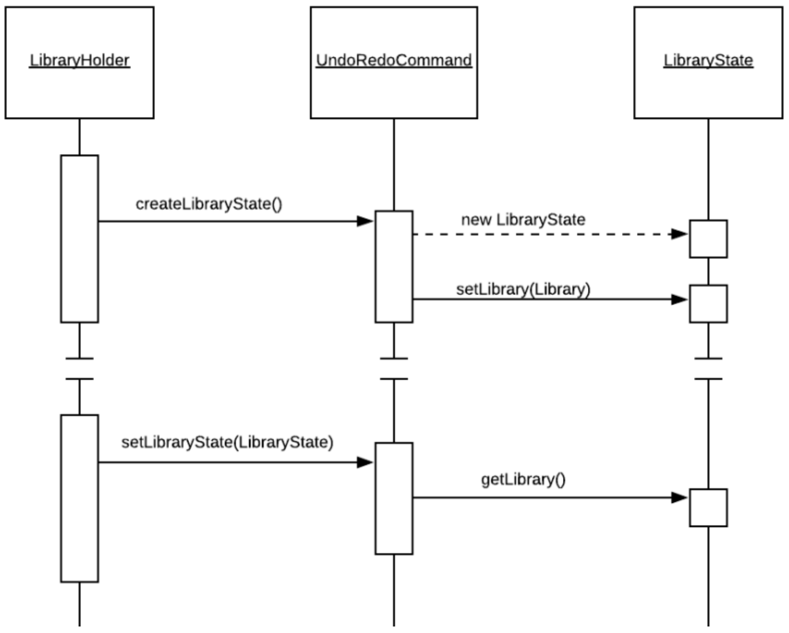
Undo/Redo

- Users are allowed to undo or redo operations made on the user library provided the command being undone or redone is an add, remove, or rate command. The LibraryHolder requests the user library from the LibraryHolder class as a side-effect of a library operation. The LibraryHolder then creates and returns an instance of the LibraryState which contains the user library. When the user then wants to undo the command, the UndoRedoCommand gives the LibraryState back to the LibraryHolder which then updates its internal library.

UML Class Diagram



Sequence Diagram



Design Pattern: Memento

Role

The Memento pattern allows the user to invoke an undo or a redo operation on add, remove, and rate commands that edit the library. When the user makes one of these commands, the library will be stored in a memento (*LibraryState*) before the command is invoked on the library allowing the user to undo the previous command such as unadding a song from the library. Using the Memento pattern prevents any class from modifying the library once it is sent to the memento since only the originator (*LibraryHolder*) has access to it.

Design Considerations

The undo and redo feature of this release can be implemented in any number of ways. The most simple way to achieve this is to have a separate library object in the class that invokes the add, remove, and rate commands. Then whenever one of those commands is entered by the user it can store the current library before the command is invoked. While simple and easy to implement, it adds more functionality to the class that invokes commands and does not separate the concerns.

In order to achieve low coupling in the system the best solution is to use the Memento pattern. The applicability of the memento pattern in the GoF book says to use this pattern when "a snapshot of (some portion of) an object's state must be saved so that it can be restored to that state later, and a direct interface to obtaining the state would expose implementation details and break the object's encapsulation". The system must store a snapshot of the user library in order to restore it later when the undo command is invoked and if our design used the simple approach described above, the user library would be open for modification and therefore not properly protected like it needs to be.

Class	**Role in GoF pattern**	**Participant's contribution in the context of the application**
LibraryHolder	**Originator**	Creates a LibraryState containing a snapshot of the current user library. When the user makes an undo or redo request it uses the LibraryState to restore the user library.
LibraryState	**Memento**	This class contains the internal state of the LibraryHolder which is the user library.
UndoRedoCommand	**Caretaker**	When a user makes an add, remove, or rate request on the library, this class requests the library from the LibraryHolder.
Deviations from the standard pattern: None		
Requirements being covered: • The system supports undo and redo of the following operations, add song/release, remove song/release, and rate song.		

Appendix

This section provides fine-grained design details for all of the classes in your design. You will capture this information using the CRC (Class-Responsibilities-Collaborators) card format below.

Class: `Database`	
Responsibilities: This class contains the collections of songs, artists, and releases that the user and the music library has access to. The collections are stored as HashMaps where the key is the GUID of the respective entity and the value is that entity's object. Contains getters and setters for each.	
Collaborators: Parser	
Uses: Song, Artist, Release, Item	**Used by:** Parser, Song, Release, CommandFactory, LibraryCommand, AddCommand, RemoveCommand, RateCommand, ExploreCommand, DatabaseSearchCommand, DatabaseSearchSongCommand, DatabaseSearchReleaseCommand
Author: Jarred Moyer, Shane Burke	

Class: `Parser`	
Responsibilities: This class uses the provided CSV files to parse each line correctly into the database class. Each line in the file is converted to its respective object (Song, Artist, Release). The files are parsed at every program start.	
Collaborators: Database	
Uses: Database, Artist, Song, Release, Item	**Used by:** Command Line
Author: Jarred Moyer, Cameron Riu	

Class: `Item`	
Responsibilities: This abstract class has the primary responsibility of providing the getters and setters for its subclasses which are Song, Artist, and Release.	
Collaborators:	
Uses:	**Used by:** Song, Artist, Release, LibrarySearchSongCommand, LibrarySearchReleaseCommand, LibrarySearchArtistCommand, DatabaseSearchSongCommand, DatabaseSearchReleaseCommand
Author: Shane Burke, Ryan Borger, Jarred Moyer	

Class: `Song`	
Responsibilities: This class holds the data for a song from one line of the song CSV file. It contains the artist of the song, the duration, and the GUID of the song.	
Collaborators: Release, Artist	
Uses: Artist, Database, Item	**Used by:** Database, Library, Parser, Release, AddCommand, RemoveCommand, RateCommand, ExploreCommand, LibrarySearchSongCommand, DatabaseSearchSongCommand
Author: Jarred Moyer, Shane Burke, Cameron Riu	

Class: `Artist`	
Responsibilities: This class holds the data for an artist from one line of the artist CSV file. It contains the name, the GUID, and the type if applicable.	
Collaborators: Song, Release	
Uses: Item	**Used by:** Database, Library, Parser, Song, Release, ExploreCommand, LibrarySearchArtistCommand
Author: Jarred Moyer, Shane Burke, Cameron Riu	

Class: `Release`	
Responsibilities: This class holds the data for an artist from one line of the Release CSV file.It contains the GUID, the duration, the artist GUID, the artist, the title, the issue date, the medium, a list of songs that are the track list, and the GUID's for that song list.	
Collaborators: Song, Artist	
Uses: Item, Song, Artist, Database	**Used by:** Database, Library, Parser, AddCommand, Remove Command, ExploreCommand, LibrarySearchSongCommand, LibrarySearchReleaseCommand, DatabaseSearchReleaseCommand
Author: Jarred Moyer, Ryan Borger, Cameron Riu, Shane Burke	

Class: `Library`	
Responsibilities: This class holds the users personal database data. Songs, releases, and artists are organized into hashmaps and are populated and their acquisition date stored. Also handles initiating save requests to persist help.	
Collaborators: ...	
Uses: Song, Release, Artist, PersistHelp, Date	**Used by:** CommandFactory, LibraryCommand, AddCommand, RemoveCommand, RateCommand, ExploreCommand, LibrarySearchCommand, LibrarySearchSongCommand,

	LibrarySearchArtistCommand, LibrarySearchReleaseCommand
Author: Jarred Moyer, Shane Burke	

Class: `PersistHelp`	
Responsibilities: Handles the loading and saving required to ensure the persistence of the library object.	
Collaborators: CommandLine	
Uses: Library	**Used by:** CommandLine, Library
Author: Jarred Moyer	

Class: `CommandLine`	
Responsibilities: The command line is the front end of the program. It calls the database parsing and then when the user inputs a command then creates the command factory to execute the command.	
Collaborators: Gui	
Uses: Database, CommandFactory, ResultObserver	**Used by:** ResultSubject
Author: Jarred Moyer, Shane Burke, Cameron Riu	

Class: `ResultSorter`	
Responsibilities: Dictates which sorting algorithm to use depending on what the user specifies in the command line.	
Collaborators: Gui	
Uses: Comparator	**Used by:** LibrarySearchArtistCommand LibrarySearchCommand LibrarySearchReleaseCommand LibrarySearchSongCommand
Author: Shane Burke	

Class: `SortAlphabetically`	
Responsibilities: Comparator that takes in items and compares their names for sorting purposes. Results will be presented in alphabetical order.	
Collaborators: SortByRating, SortByReleaseDate, SortByAcquisitionDate	
Uses: Comparator	**Used by:** ResultSorter
Author: Ryan Borger	

Class: `SortByAcquisitionDate`	

Responsibilities: Comparator that takes in items and compares their acquisition date for sorting purposes. Results will be presented by the most recent acquisition date.	
Collaborators: SortByRating, SortByReleaseDate, SortAlphabetically	
Uses: Comparator	**Used by:** ResultSorter
Author: Ryan Borger	

Class: `SortByRating`	
Responsibilities: Comparator that takes in items and compares their rating for sorting purposes. Results will be presented by highest to lowest rating.	
Collaborators: SortAlphabetically, SortByReleaseDate, SortByAcquisitionDate	
Uses: Comparator	**Used by:** ResultSorter
Author: Ryan Borger, Shane Burke	

Class: `SortByReleaseDate`	
Responsibilities: Comparator that compares two item’s release date for sorting purposes. Results will be presented by the most recent release date.	
Collaborators: SortByRating, SortAlphabetically, SortByAcquisitionDate	
Uses: Comparator	**Used by:** ResultSorter
Author: Ryan Borger	

Class: `ResultFilterer`	
Responsibilities: This interface is responsible for representing the objects that accept, filter, then return a list of items. The objects can carry any number of decorators.	
Collaborators: AddedFilter, BaseFilter	
Uses: Item	**Used by:** UserCommand, CommandCreator
Author: ...	

Class: `BaseFilter`	
Responsibilities: This class is a filter that removes no results from the specified list and acts as the base component upon which decorative filters are applied. It is used in cases where no filtering is required to promote uniformity with results that require filtering.	
Collaborators: AddedFilter, ResultFilterer	

Uses: Item	**Used by:** CommandCreator
Author: ...	

Class: `AddedFilter`	
Responsibilities: This class represents a filter that adds additional filtering behavior to an existing BaseFilter or already decorated filtering object. It is supplied another Component so that its existing filtering behavior may be performed in addition to the added behavior.	
Collaborators: ResultFilterer, BaseFilter	
Uses: Item	**Used by:** CommandCreator, ArtistNameFilter, MinRatingFilter, TitleFilter, MaxDurationFilter, MinDurationFilter, etc.
Author: ...	

Class: `ArtistNameFilter`	
Responsibilities: When this class is supplied a list of items, returns a new list that only includes the items for which the artist name matches the one specified in the filter. Used when searching for items by a specific artist.	
Collaborators: MinRatingFilter, TitleFilter, MaxDurationFilter, MinDurationFilter, etc.	
Uses: AddedFilter, ResultFilterer	**Used by:** UserCommand, CommandCreator
Author: ...	

Class: `MinRatingFilter`	
Responsibilities: When this class is supplied a list of items, returns a new list that only includes the items for which the minimum rating is equal to or greater than the one specified in the filter. Used when searching for library items that have a rating above a minimum threshold.	
Collaborators: ArtistNameFilter, TitleFilter, MaxDurationFilter, MinDurationFilter, etc.	
Uses: AddedFilter, ResultFilterer	**Used by:** UserCommand, CommandCreator
Author: ...	

Class: `ArtistGuidFilter`	
Responsibilities: When this class is supplied a list of items, returns a new list that only includes the items for	

which the artist GUID matches the one specified in the filter. Used when searching for items by a specific artist.	
Collaborators: AddedFilter, ResultFilterer	
Uses: AddedFilter, ResultFilterer	**Used by:** UserCommand, CommandCreator
Author: ...	

Class: `TrackNameFilter`	
Responsibilities: When this class is supplied a list of items, returns a new list that only includes the items for which an included track’s name matches the one specified in the filter. Used when searching for releases containing a specific track.	
Collaborators: ArtistNameFilter, TitleFilter, MaxDurationFilter, MinDurationFilter, etc.	
Uses: AddedFilter, ResultFilterer	**Used by:** UserCommand, CommandCreator
Author: ...	

Class: `TrackGuidFilter`	
Responsibilities: When this class is supplied a list of items, returns a new list that only includes the items for which an included track’s GUID matches the one specified in the filter. Used when searching for releases containing a specific track.	
Collaborators: ArtistNameFilter, TitleFilter, MaxDurationFilter, MinDurationFilter, etc.	
Uses: AddedFilter, ResultFilterer	**Used by:** UserCommand, CommandCreator
Author: ...	

Class: `TitleFilter`	
Responsibilities: When this class is supplied a list of items, returns a new list that only includes the items for which the title matches the one specified in the filter. Used when searching for items with a specific title.	
Collaborators: ArtistNameFilter, TitleFilter, MaxDurationFilter, MinDurationFilter, etc.	
Uses: AddedFilter, ResultFilterer	**Used by:** UserCommand, CommandCreator
Author: ...	

Class: `MinDurationFilter`	

Responsibilities: When this class is supplied a list of items, returns a new list that only includes the items for which the total duration exceeds the specified minimum. Used when searching for songs or releases with a minimum duration.	
Collaborators: ArtistNameFilter, TitleFilter, MaxDurationFilter, MinDurationFilter, etc.	
Uses: AddedFilter, ResultFilterer	**Used by:** UserCommand, CommandCreator
Author: ...	

Class: `MaxDurationFilter`	
Responsibilities: When this class is supplied a list of items, returns a new list that only includes the items for which the total duration is lower than the specified maximum. Used when searching for songs or releases with a maximum duration.	
Collaborators: ArtistNameFilter, TitleFilter, MinDurationFilter, etc.	
Uses: AddedFilter, ResultFilterer	**Used by:** UserCommand, CommandCreator
Author: ...	

Class: `ArtistTypeFilter`	
Responsibilities: When this class is supplied a list of items, returns a new list that only includes the items for which the artist type matches the specified type. Used when searching for artists of a specific type.	
Collaborators: ArtistNameFilter, TitleFilter, MaxDurationFilter, MinDurationFilter, etc.	
Uses: AddedFilter, ResultFilterer	**Used by:** UserCommand, CommandCreator
Author: ...	

Class: `ReleaseTitleFilter`	
Responsibilities: When this class is supplied a list of items, returns a new list that only includes the items for which the release title matches the specified title. Used when searching for songs belonging to a specific release.	
Collaborators: ArtistNameFilter, TitleFilter, MaxDurationFilter, MinDurationFilter, etc.	
Uses: AddedFilter, ResultFilterer	**Used by:** UserCommand, CommandCreator
Author: ...	

Class: `ReleaseGuidFilter`	
Responsibilities: When this class is supplied a list of items, returns a new list that only includes the items for which the release GUID matches the specified GUID. Used when searching for songs belonging to a specific release.	
Collaborators: ArtistNameFilter, TitleFilter, MaxDurationFilter, MinDurationFilter, etc.	
Uses: AddedFilter, ResultFilterer	**Used by:** UserCommand, CommandCreator
Author: ...	

Class: `MinReleaseDateFilter`	
Responsibilities: When this class is supplied a list of items, returns a new list that only includes the items for which the release date is later than the specified date. Used when searching for releases with release dates after a certain date.	
Collaborators: ArtistNameFilter, TitleFilter, MaxDurationFilter, MinDurationFilter, etc.	
Uses: AddedFilter, ResultFilterer	**Used by:** UserCommand, CommandCreator
Author: ...	

Class: `MaxReleaseDateFilter`	
Responsibilities: When this class is supplied a list of items, returns a new list that only includes the items for which the release date is earlier than the specified date. Used when searching for releases with release dates before a certain date.	
Collaborators: ArtistNameFilter, TitleFilter, MaxDurationFilter, MinDurationFilter, etc.	
Uses: AddedFilter, ResultFilterer	**Used by:** UserCommand, CommandCreator
Author: ...	

Class: `SortFilter`	
Responsibilities: When this class is instantiated, the *SortFilter* is supplied a *Comparator* object. This object is used to sort the results given by the *filter(...)* method.	
Collaborators: ArtistNameFilter, TitleFilter, MaxDurationFilter, MinDurationFilter, etc.	
Uses: AddedFilter, ResultFilterer	**Used by:** UserCommand, CommandCreator

Author: ...	

Class: `Gui`	
Responsibilities: This class is used to make a query (through a button press, etc.), processes the request through the Client. Invokes Commands once created.	
Collaborators: CommandLine	
Uses: ResultObserver, CommandCreator, UserCommand, CommandScene, Scene	**Used by:** ResultSubject, MusicDatabaseScene, LibraryScene, RatingScene, LoginScene, CreateAccountScene
Author: ...	

Class: `CommandCreator`	
Responsibilities: This class Orchestrates the creation of a particular Command. Defines abstract methods for creating command line (Cli) or GUI commands. For CLI commands, the user is prompted for all parameters before the Command object is created. Returns the created Concrete Command object to the Invoker.	
Collaborators: UserCommand	
Uses: CommandScene, Library, Database, ResultProvider, ResultHandler	**Used by:** CommandLine, Gui, MusicDatabaseScene, LibraryScene, RatingScene, LoginScene, CreateAccountScene
Author: ...	

Note: All command classes below have a CommandCreator class of their own, [class name]Creator, that use the abstract methods above.

Class: `UserCommand`	
Responsibilities: This class describes all user commands. Provides a common interface, used by the Invoker, to execute commands. Hides the implementation and type of a particular command from the Invoker.	
Collaborators: CommandCreator	
Uses: Library, Database, ResultHandler, ResultProvider, ResultFilterer	**Used by:** CommandLine, Gui
Author: ...	

Class: `RateCommand`	
Responsibilities: This class, when executed, assigns a rating to an item in the user’s library. Invoked by the Invoker after the user requests to rate an item.	

Collaborators: CommandCreator	
Uses: Library, ResultProvider	**Used by:** CommandLine, Gui
Author: ...	

Class: `ListCommand`	
Responsibilities: This class, when executed, provides the list of artists whose music resides in the user's library. If in command line mode, formats and prints the list to the command line	
Collaborators: CommandCreator	
Uses: Library, ResultHandler	**Used by:** LibraryScene, CommandLine
Author: ...	

Class: `LibrarySearchReleaseCommand`	
Responsibilities: This class, when executed, searches the library for releases matching the specified criteria. Results are passed to the ResultHandler that was supplied by the Client.	
Collaborators: CommandCreator	
Uses: Library, ResultHandler	**Used by:** LibraryScene, CommandLine
Author: ...	

Class: `LibrarySearchArtistCommand`	
Responsibilities: This class, when executed, searches the library for artists matching the specified criteria. Results are passed to the ResultHandler that was supplied by the Client.	
Collaborators: CommandCreator	
Uses: Library, ResultHandler	**Used by:** LibraryScene, CommandLine
Author: ...	

Class: `LibrarySearchSongCommand`	
Responsibilities: This class, when executed, searches the library for songs matching the specified criteria. Results are passed to the ResultHandler that was supplied by the Client.	
Collaborators: CommandCreator	
Uses: Library, ResultHandler	**Used by:** CommandLine, LibraryScene
Author: ...	

Class: `DatabaseSearchReleaseCommand`	
Responsibilities: This class, when executed, searches the current database for releases matching the specified criteria. Results are passed to the ResultHandler that was supplied by the Client.	
Collaborators: CommandCreator	
Uses: Database, ResultHandler	**Used by:** MusicDatabaseScene, ResultsHandler
Author: ...	

Class: `DatabaseSearchArtistCommand`	
Responsibilities: This class, when executed, searches the current database for artists matching the specified criteria. Results are passed to the ResultHandler that was supplied by the Client.	
Collaborators: CommandCreator	
Uses: Database, ResultHandler	**Used by:** MusicDatabaseScene, CommandLine
Author: ...	

Class: `DatabaseSearchSongCommand`	
Responsibilities: This class, when executed, searches the current database for songs matching the specified criteria. Results are passed to the ResultHandler that was supplied by the Client.	
Collaborators: CommandCreator	
Uses: CommandScene	**Used by:** CommandLine, MusicDatabaseScene
Author: ...	

Class: `ExploreCommand`	
Responsibilities: This class, when executed (and the most recent item results are of artists in the library), the artists' releases and other songs are listed for the user. If another ExploreCommand is executed while in this state, the corresponding release's songs will be displayed for the user.	
Collaborators: CommandCreator	
Uses: CommandScene	**Used by:** CommandLine, MusicLibraryScene
Author: ...	

Class: `AddCommand`	
Responsibilities: This class, when executed, adds the item corresponding to a search result to the user's library. The identifier of the search result to add is	

specified when the command is created through the Builder pattern.	
Collaborators: CommandCreator	
Uses: CommandScene	**Used by:** CommandLine, MusicLibraryScene
Author: ...	

Class: `RemoveCommand`	
Responsibilities: This class, when executed, removes the item corresponding to a search result from the user's library. The identifier of the search result to add is specified when the command is created through the Builder pattern.	
Collaborators: CommandCreator	
Uses: CommandScene	**Used by:** CommandLine, Gui
Author: ...	

Class: `HelpCommand`	
Responsibilities: This class, when executed, displays a help message instructing the user on the use of the application.	
Collaborators: CommandCreator	
Uses: CommandScene	**Used by:** CommandLine, Gui
Author: ...	

Class: `BackCommand`	
Responsibilities: This class, when executed (and the user is currently exploring an artist in the library), the list of artists in the library will be displayed to the user again.	
Collaborators: CommandCreator	
Uses: CommandScene	**Used by:** CommandLine, Gui
Author: ...	

Class: `LoginCommand`	
Responsibilities: This class, when executed with a username and password, checks for a matching username and password in the user database. If the credentials match, the user is logged in and taken to the home screen	
Collaborators: CommandCreator	
Uses: CommandScene	**Used by:** CommandLine, Gui
Author: ...	

Class: `CreateAccountCommand`	
Responsibilities: This class, when executed with a password and a username that is not already in the database, a new account is created in the user database with these credentials.	
Collaborators: CommandCreator	
Uses: CommandScene	**Used by:** ...
Author: ...	

Class: `ResultSubject`	
Responsibilities: This class defines the interface for registering and notifying observers. This reusable interface may be implemented by any class that produces results and desires to inform observers of them.	
Collaborators: ResultObserver	
Uses:	**Used by:** ResultManager
Author: ...	

Class: `ResultObserver`	
Responsibilities: This class defines the interface for notifying an observer of new results in the Subject. It includes a single abstract method *updateResults()*. It requires that the Concrete Observer “pulls” the results from the Concrete Subject.	
Collaborators: ResultSubject	
Uses:	**Used by:** CommandLine, Gui
Author: ...	

Class: `ResultManager`	
Responsibilities: This class is the primary implementation of the ResultSubject interface. Implements the tracking and notifying of observers. Also provides public methods *getAvailableCommands()* and *getItems()*, which are used by Concrete Observers to “pull” the updated state once notified. Stores the most recent search or explore results. Provides interfaces for storing/retrieving results as well as retrieving the list of available commands.	
Collaborators: Gui, CommandLine	
Uses: ResultSubject, ResultProvider, ResultHandler	**Used by:** CommandCreator, UserCommand
Author: ...	

Class: `ResultState`	
Responsibilities: This class defines the list of available commands for a given state. Updates the Context's internal state upon receiving new results. Provides a common interface for all Concrete States. Hides the implementation and type of the current state from the Context.	
Collaborators: ResultManager, Concrete States	
Uses: Item	**Used by:** ResultManager
Author: ...	

Class: `NoResultsState`	
Responsibilities: This class defines the list of available commands before any result-producing command has been executed. Handles the Context's state transitions upon receiving new results while in this state.	
Collaborators: ResultManager, other Concrete States	
Uses: Item	**Used by:** ResultManager
Author: ...	

Class: `LibraryReleaseResultState`	
Responsibilities: This class defines the list of available commands when the most recent search results are from a release search in the library. Handles the Context's state transitions upon receiving new results while in this state.	
Collaborators: ResultManager, other Concrete States	
Uses: Item	**Used by:** ResultManager
Author: ...	

Class: `LibrarySongResultState`	
Responsibilities: This class defines the list of available commands when the most recent search results are from a song search in the library. Handles the Context's state transitions upon receiving new results while in this state.	
Collaborators: ResultManager, other Concrete States	
Uses: Item	**Used by:** ResultManager
Author: ...	

Class: `LibraryArtistResultState`	

Responsibilities: This class defines the list of available commands when the most recent search results are from an artist search in the library. Handles the Context's state transitions upon receiving new results while in this state.	
Collaborators: ResultManager, other Concrete States	
Uses: Item	**Used by:** ResultManager
Author: ...	

Class: `DatabaseReleaseResultState`	
Responsibilities: This class defines the list of available commands when the most recent search results are from a release search in the database. Handles the Context's state transitions upon receiving new results while in this state.	
Collaborators: ResultManager, other Concrete States	
Uses: Item	**Used by:** ResultManager
Author: ...	

Class: `DatabaseSongResultState`	
Responsibilities: This class defines the list of available commands when the most recent search results are from a song search in the database. Handles the Context's state transitions upon receiving new results while in this state.	
Collaborators: ResultManager, other Concrete States	
Uses: Item	**Used by:** ResultManager
Author: ...	

Class: `DatabaseArtistResultState`	
Responsibilities: This class defines the list of available commands when the most recent search results are from an artist search in the database. Handles the Context's state transitions upon receiving new results while in this state.	
Collaborators: ResultManager, other Concrete States	
Uses: Item	**Used by:** ResultManager
Author: ...	

Class: `ExploreResultState`	
Responsibilities: This class defines the list of available commands when the most recent results are from the "explore" command. Handles the Context's	

state transitions upon receiving new results while in this state.	
Collaborators: ResultManager, other Concrete States	
Uses: ResultProvider, Item	**Used by:** ResultManager
Author:	

Class: `LibraryHolder`	
Responsibilities: This class creates a LibraryState containing a snapshot of the current user library. When the user makes an undo or redo request it uses the LibraryState to restore the user library.	
Collaborators: UndoRedoCommand	
Uses: Library, LibraryState	**Used by:** CommandLine, Gui
Author: ...	

Class: `LibraryState`	
Responsibilities: This class contains the internal state of the LibraryHolder which is the user library.	
Collaborators:	
Uses: Library	**Used by:** LibraryHolder
Author: ...	

Class: `UndoRedoCommand`	
Responsibilities: When a user makes an add, remove, or rate request on the library, this class requests the library from the LibraryHolder.	
Collaborators: LibraryHolder	
Uses: Library	**Used by:** CommandLine, Gui
Author: ...	