Library Book Management System

Team B: SWEN-262 Design Project R2

Charles Barber, Nicholas Feldman, Christopher Lim,

Anthony Palumbo, Edward Wong

Table of Contents

SUMMARY

REVISION TABLE

Revision Number	Revision Date	Summary	Author
0.1	*02/14/2017*	*Domain Model*	*Anthony Palumbo*
0.2	*02/16/2017*	*Nouns & Verbs, Knowns & Unknowns*	*Charles Barber, Edward Wong*
0.3	*02/21/2017*	*Design Pattern Usage*	*Nicholas Feldman*
1.0	*02/22/2017*	*Initial Creation and Changes*	*Christopher Lim*
1.1	*02/03/2017*	*State vs. Strategy*	*Nicholas Feldman*
1.2	*03/02/2017*	*Design Evaluation*	*Anthony Palumbo*
1.3	*03/03/2017*	*Added UML Class Diagrams*	*Nicholas Feldman*
1.4	*03/10/2017*	*Updated UML Class Diagrams*	*Anthony Palumbo*
1.5	*03/10/2017*	*Added Feature Requirements*	*Christopher Lim*
1.6	*03/11/2017*	*Added Subsystem Design*	*Charles Barber*
1.7	*03/11/2017*	*Added Design Pattern Usage to Subsystems*	*Nicholas Feldman*
1.8	*03/15/2017*	*Updated UML Class Diagrams*	*Anthony Palumbo*
1.9	*03/15/2017*	*Added Architecture Model*	*Christopher Lim*
1.10	*03/15/2017*	*Added CRC Cards*	*Edward Wong*
1.11	*03/17/2017*	*Added Sequence Diagrams*	*Anthony Palumbo, Charles Barber*
1.12	*03/17/2017*	*Updated Domain Model*	*Anthony Palumbo*
1.13	*03/19/2017*	*Completed CRC Cards*	*Anthony Palumbo, Edward Wong*
1.14	*03/20/2017*	*Formatted Document*	*Christopher Lim*

2.0	*04/03/2017*	*Fixed Grammatical Issues*	*Charles Barber*
2.1	*04/05/2017*	*Updated Requirements for R2 and Domain Model*	*Christopher Lim*
2.2	*04/07/2017*	*Removed ViewState*	*Charles Barber*
2.3	*04/08/2017*	*Updated UML Class Diagrams*	*Anthony Palumbo, Nicholas Feldman*
2.4	*04/08/2017*	*Updated UML Sequence Diagram*	*Anthony Palumbo, Charles Barber*
2.5	*04/15/2017*	*Added New CRC Cards*	*Edward Wong*
2.6	*04/15/2017*	*Added Description of Library State Implementation*	*Edward Wong*
2.7	*04/16/2017*	*Added Description of Proxy Implementation*	*Nicholas Feldman*
2.8	*04/16/2017*	*Added new design patterns to document*	*Anthony Palumbo*
2.9	*04/17/2017*	*Corrected and Expanded Prose*	*Christopher Lim, Anthony Palumbo*
3.0	*04/18/2017*	*Formatted Document*	*Christopher Lim*

PROBLEM STATEMENT

Design and implement the Library Book Management System (LBMS). The LBMS is Book Worm Library's (BWL) system for providing book information to users, tracking library visitor statistics for a library statistics report, tracking checked out books, and allowing the library inventory to be updated. It is the server-side system that provides an API used by client-side interfaces that BWL employees use.

SYSTEM REQUIREMENTS

At a high-level this project will be source controlled on GitHub, in a private repository until after the project is over, and implemented in Java as a desktop application. It will be compatible with the standard Java 1.8 SDK installed on the RIT SE lab machines. The system does not require or use any form of external database, persisting only in standard Java constructs. The system will be delivered as an executable jar file and require no network connection to function. A batch file, start.bat, will be provided to set any required environment variables, perform any program specific initialization, and execute the graphical user interface for the program. Another file, startAPI.bat, will be provided to run only the API version of the project that takes in requests and prints out the responses. This version is particularly useful for other developers who may need to run a specific part of the project without the

graphical user interface. On a clean exit the program will either produce or update a file named data.ser, this contains all the serialized data for the system and can be deleted to get a clean start of the program.

FEATURE REQUIREMENTS

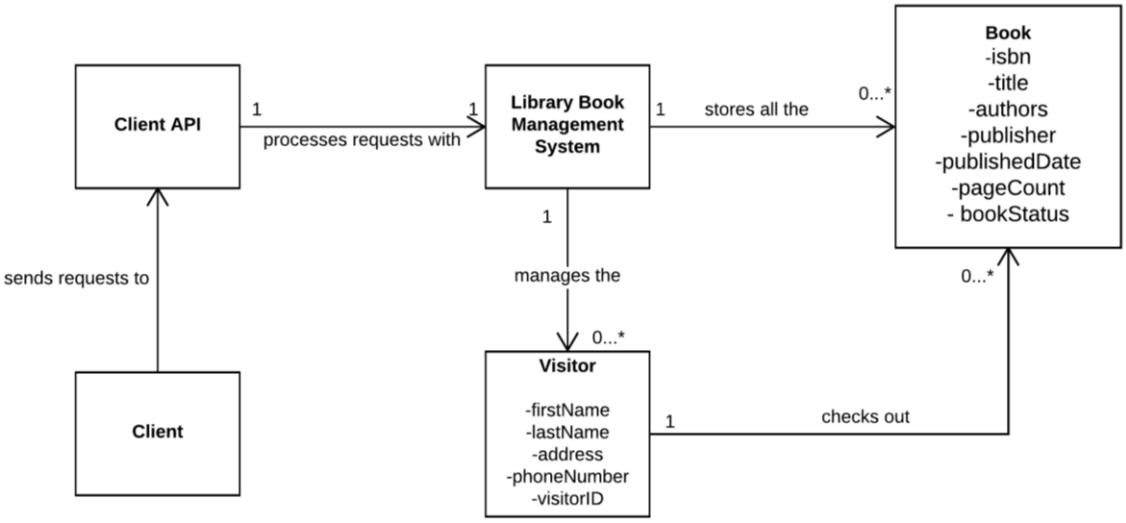
No.	*User Story Name*	*Description*
1	API	The LBMS shall use text-based requests and responses. An LBMS exchange consists of one text string sent by a client followed by one text string sent by the system. The system shall receive requests from a client as text strings. A client shall terminate request strings with a semicolon (;) character. If the exchange is a partial client request, i.e. does not end with a termination character, the system response shall indicate that it received the partial client request and the system shall wait to receive the remainder of the request in the next one or more exchanges. If the exchange completes a client request, the system shall perform the requested operation, and provide a response for the request according to the LBMS server reply format specification.
2	Visitor Registration	The LBMS will require that first time visitors to the library register. The system will store the following information for each visitor: first name, last name, address, phone number, and a visitor ID (a unique 10-digit ID generated upon visitor registration).
3	Visits	The LBMS shall keep track of visits by visitors. The system shall keep track of the time each visitor spends at the library during each visit. (Information will be used for statistical data in library reports.)
4	Operational Hours	The library opens at 08:00 every day. All visits in progress are automatically ended when the library closes at 19:00 when visitors remaining in the library are asked to leave. Visits do not extend over multiple days.
5	Searching	The LBMS shall respond to queries for book information. The system shall store book data for all books currently in BWL's possession. Book data shall consist of: isbn, title, author (can be multiple authors), publisher, published date, page count, number of copies, and number of copies currently checked out. The system shall respond with all information matching the provided search parameters in the order requested in the query. The client can request an ordering by title, publish date, and book status (i.e. not checked out). The system shall respond with an empty string when there are no books matching the query.
6	Checking out	The LBMS shall track checked out books by visitors. The system shall allow each visitor to checkout a maximum of 5 books at a time. Each book may be checked out for a maximum of 7 days. The system will store the data of the book check out transaction with the following information: isbn, visitor ID, date checked out, due date.

7	Fines	The LBMS shall apply an initial $10.00 fine to all books 1 day overdue. Subsequently, $2.00 will be added to the initial fee for each additional week overdue, up to a maximum fine of $30.00.
8	Statistics Report	The LBMS shall respond to queries for an informational report of the library. The system shall respond to a statistical query with the following information about a queried month at the library: i. The number of books currently owned by the library. ii. The number of visitors registered at library. iii. The average amount of time spent at the library for a visit. iv. The books purchased for the specified month. v. The amount of money collected through checked out book fines
9	Advance Time	The LBMS shall support a feature to track and advance time. On initial startup, the LBMS system will record the date and time. The LBMS system shall track the number of days that have passed while the system is in operation. The LBMS system shall allow users to move the date forward by a specified number of days. The time of day will remain unaffected. The system will also allow for the advancement of the time by a specified number of hours. Upon each date change the system will generate a report of any overdue books (checked out by users past the due date).
10	Clean Shutdown	The LBMS system shall provide a mechanism for a “clean” shutdown of the system. The system shall end any visits in progress at system shutdown and persist all data at system shutdown.
11	Startup	The LBMS system shall restore persistent state on startup. Any state previously stored will be restored on startup.
12	Concurrent Client Connections	The LBMS shall support operations provided by multiple, concurrent clients. A client will be prompted to log in immediately upon establishing a connection and will be automatically logged out upon disconnecting. Each client connection will be separate in the sense that the library will handle all request and response exchanges for each client, individually. Actions taken by individual clients that change the state of the system will affect other clients (e.g. borrowing books will affect book availability for all clients).
13	Client Accounts	The system shall store client account for visitors and employees. An account represents the username, password, and role for an individual visitor. Each account will have different access permissions depending on the type (either visitor or employee). Employees will have access to the entire system while visitors who are not employees will only be able to begin a visit, end a visit, search the library, and borrow a book in addition to basic system tasks (e.g. connect, log in, log out, etc.).

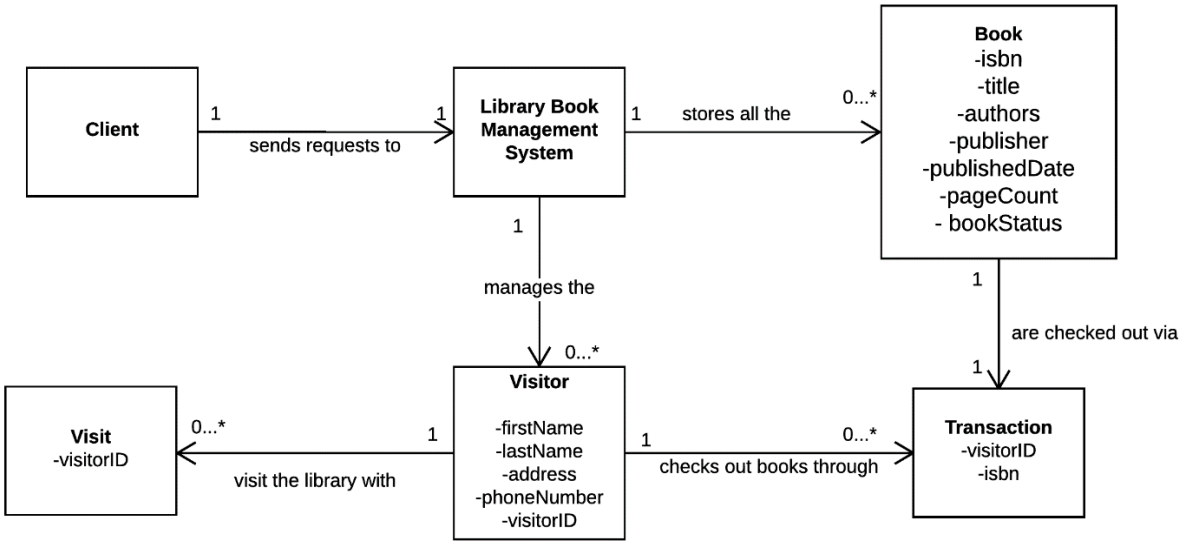
14	Undo/Redo	The LBMS system shall support the ability to undo and redo actions by any user. The actions that support the undo/redo functionality are the following: Begin Visit, End Visit, Purchase Book, Borrow Book, Return Book, and Pay Fine.
15	Purchase Book Source	The LBMS will now allow employees to choose between a provided books.txt file and Google Books as a source of books available for the library to purchase. In the case of using Google Books as a source of books, only books labeled as for sale in the US will be available for purchase.

DOMAIN MODEL

ORIGINAL



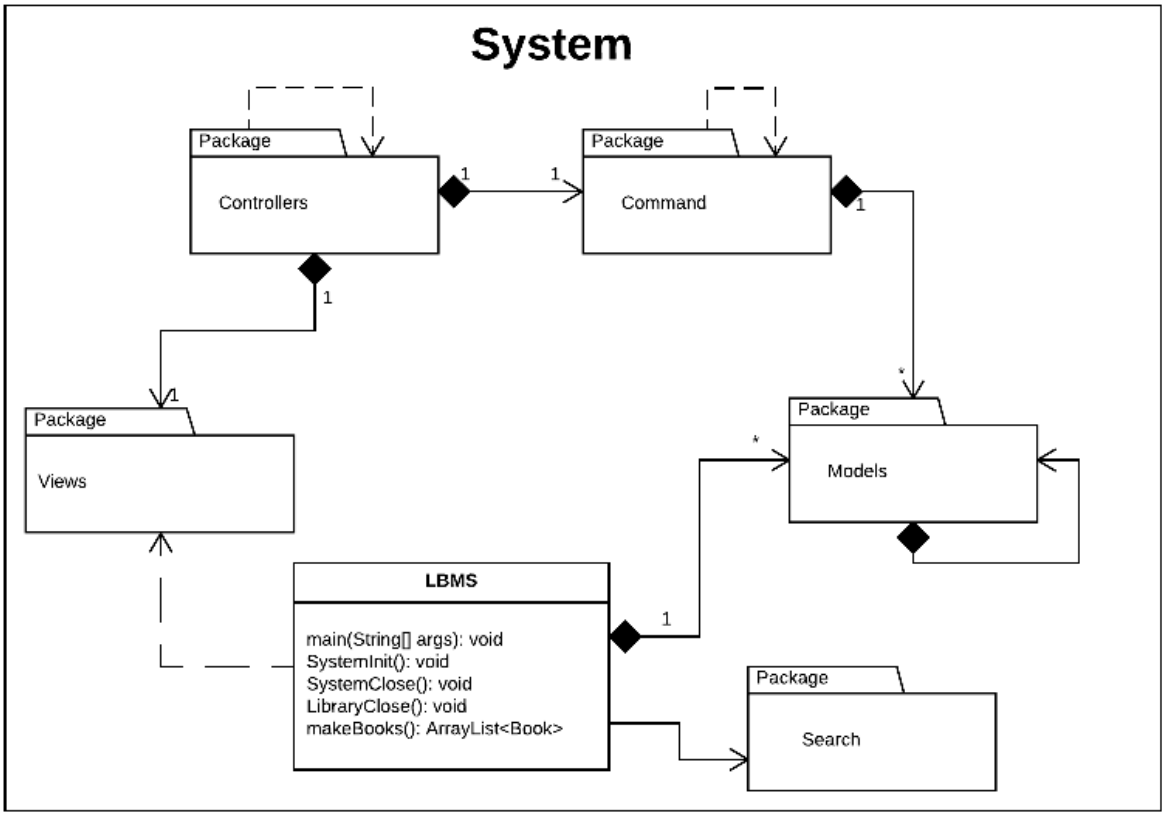
UPDATED



ARCHITECTURAL MODEL

The following model displays the interactions between each main package forming the entire system. We tried to organize the subsystems by package, however each package does not directly correlate to a subsystem. The entry point into the system is through LBMS, which is dependent on the views package as it is displayed to the user as a view. User input is sent to the controllers which interprets and handles the input and converts it into a specific command within the Command Subsystem, contained in the command package. The controllers update the view with the appropriate view once the command is executed. The Command Subsystem interacts with the models modifying and/or accessing the data. Model data are stored in memory within LBMS. This memory can be queried using the Search Subsystem, also contained within the search package, which is used for things such as a search command.

The advantages of this design include separation of concerns using a model-view-controller architecture, allowing the system to update the view at any time while keeping the interaction with the model standardized and encapsulated, and the ability to run two different mode, API and GUI, while reusing much of the code with common functionality. With the structure we have setup, it is easy to add features to the system by following the MVC pattern. The disadvantage of our system is the large number of packages and classes within them we used in order to follow design patterns.

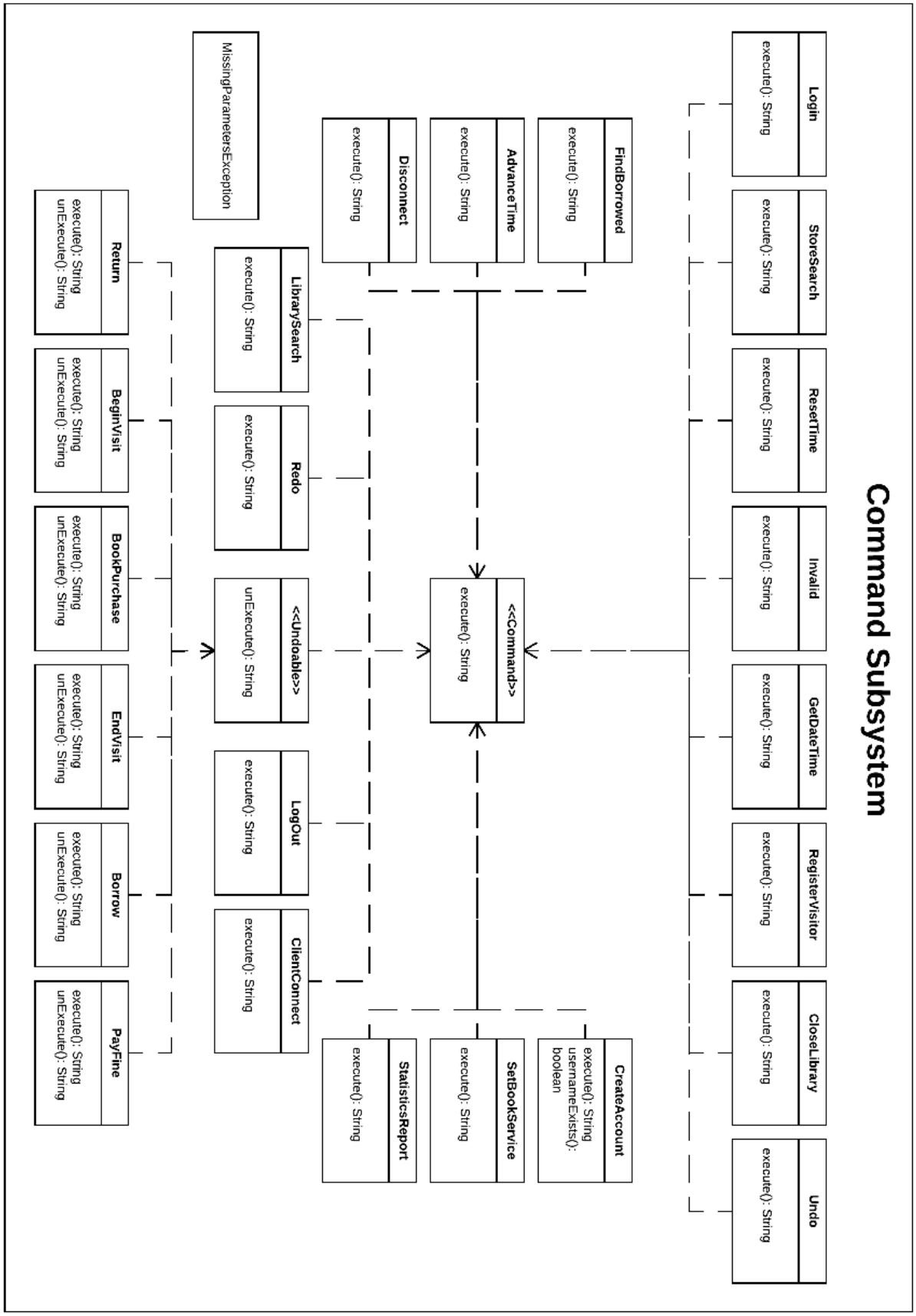


SUBSYSTEM DESIGN

COMMAND SUBSYSTEM

The Command Subsystem implements the command design pattern and is contained to the command package. The main interface, Command, requires that all classes that implement it have an execute() method that returns a String. The way we setup our classes the constructors all take in a request String, and possibly an additional parameter for simplifying the String parsing, and store the data in the command. Then when execute() is called the command interacts with the LBMS class to receive and/or modify its data and generates a response string that follows the request/response format that was given to us for this project. If the original request String is not properly formatted the contructors may throw a MissingParametersException and the CommandController class will handle it. For R2 we added another interface, Undoable, that extends the Command interface and adds a required method called, unExecute(). Commands that implement this interface are able to be “undone” and “redone” by a user of the system, the unExecute() method essentially reverses what was done in execute() and is used when a command is “undone”.

There are several advantages to using this design for commands. First, a new developer can see the direct correlation between each command and the accompanying request requirement from the project website which makes it really simply to add commands to the system. Additionally each command’s implementation is separated into its own class which follows the idea of separation of concerns. The only disadvantage of using this pattern is the requirement that all commands must implement the Command interface, which is not really an issue. Also it passed the responsibility of handling the request formats to the CommandController by throwing a MissingParametersException.


Command Subsystem
Login
execute(): String
StoreSearch
execute(): String
ResetTime
execute(): String
Invalid
execute(): String
GetDateTime
execute(): String
RegisterVisitor
execute(): String
CloseLibrary
execute(): String
Undo
execute(): String
FindBorrowed
execute(): String
AdvanceTime
execute(): String
Disconnect
execute(): String
<<Command>>
execute(): String
CreateAccount
execute(): String
usernameExists(): boolean
SetBookService
execute(): String
StatisticsReport
execute(): String
MissingParametersException
LibrarySearch
execute(): String
Redo
execute(): String
<<Undoable>>
unExecute(): String
LogOut
execute(): String
ClientConnect
execute(): String
Return
execute(): String
unExecute(): String
BeginVisit
execute(): String
unExecute(): String
BookPurchase
execute(): String
unExecute(): String
EndVisit
execute(): String
unExecute(): String
Borrow
execute(): String
unExecute(): String
PayFine
execute(): String
unExecute(): String

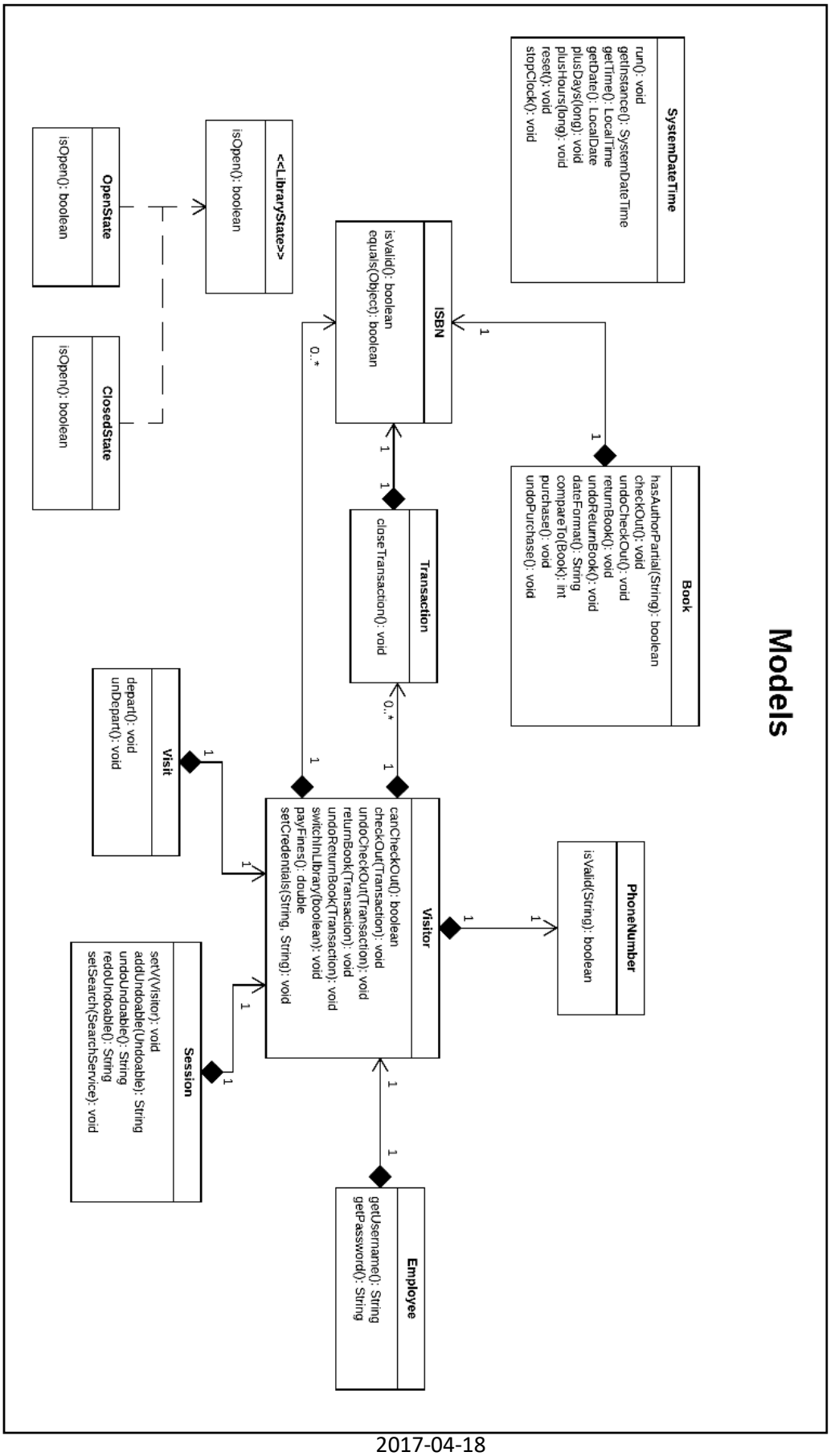
Name: Library Commands		**GoF pattern:** Command
Participants		
Class	**Role in GoF pattern**	**Participant's contribution in the context of the application**
Command	Interface	This interface is the template for the concrete commands. It declares and requires each command to implement the execute() method. This method is common to all commands and does not share a common implementation. Each concrete command initializes by retrieving relevant information from an input string.
Undoable	Interface	This interface extends the Command interface. It adds functionality to a command by requiring the unExecute() method to reverse the execute() method when it is “undone”.
MissingParameters Exception	N/A	Custom exception used by the constructors of commands.
AdvanceTime	ConcreteCommand	This class defines the steps specific to advancing the system time by some number of days and hours.
BeginVisit	ConcreteCommand	This class defines the steps specific to having a visitor begin a visit at the library.
BookPurchase	ConcreteCommand	This class defines the steps specific to purchasing a book from the bookstore to add to the library’s collection.
Borrow	ConcreteCommand	This class defines the steps specific to having a visitor borrow an available book from the library’s collection.
ClientConnect	ConcreteCommand	This class defines the steps specific to connecting a client instance to the system.
CloseLibrary	ConcreteCommand	This class defines the steps specific to closing the library at closing time.
CreateAccount	ConcreteCommand	This class defines the steps specific to creating an account for a user to log in. Specifically, this process creates username and password credentials for an existing visitor.
Disconnect	ConcreteCommand	This class defines the steps specific to disconnecting a client instance from the system.
EndVisit	ConcreteCommand	This class defines the steps specific to having a visitor end his or her visit to the library.

FindBorrowed	ConcreteCommand	This class defines the steps specific to finding the books currently borrowed from the library by a particular visitor.
GetDateTime	ConcreteCommand	This class defines the steps specific to retrieving the system date and time (which may be different from the current date and time).
Invalid	ConcreteCommand	This class defines the steps specific to informing the user an invalid command string was entered.
LibrarySearch	ConcreteCommand	This class defines the steps specific to searching the library's collection of books for books matching input criteria.
LogIn	ConcreteCommand	This class defines the steps specific to logging in a visitor to a client instance.
LogOut	ConcreteCommand	This class defines the steps specific to logging out a visitor to a client instance.
PayFine	ConcreteCommand	This class defines the steps specific to having a visitor who owes overdue book fines pay those fines.
Redo	ConcreteCommand	This class defines the steps specific to redoing an undoable command.
RegisterVisitor	ConcreteCommand	This class defines the steps specific to having a new visitor register with the system.
ResetTime	ConcreteCommand	This class defines the steps specific to resetting the system time to the current date and time.
Return	ConcreteCommand	This class defines the steps specific to having a visitor return a book they have borrowed.
SetBookService	ConcreteCommand	This class defines the steps specific to setting the service that will be providing the responses for the book purchase search (either book.txt [local] or GoogleBooks [google]).
StatisticsReport	ConcreteCommand	This class defines the steps specific to generating a report of the current state of the library.
StoreSearch	ConcreteCommand	This class defines the steps specific to searching the bookstore for books available for purchase by the library which match input criteria.
Undo	ConcreteCommand	This class defines the steps specific to undoing an undoable command.

LBMS	Receiver	This class contains the state of the system and therefore receives the actions executed by the commands.
Deviations from the standard pattern: • Addition of a second interface to implement commands that can be “undone”. • Custom exception that is thrown in constructors to signal an improper request format that it was given.		
Requirements being covered: • The application must perform several different actions relating to application content. • All commands can be treated identically from the outside since they all have a similar pattern of creation and execution. The execution is always handled in a method named execute() and contains all steps required to properly perform the action.		

MODELS SUBSYSTEM

The Models Subsystem is just a grouping of all the models we implemented and their interactions. These classes were separated into their own package to follow the MVC pattern and to make it clear what was part of the system data. Each class stores different data required by the system and all inherit from the Serializable interface, allowing them to persist across startups. One class, SystemDateTime, implements the Singleton pattern, because there will only ever be one instance of SystemDateTime. The State pattern is also implemented in the models as the library itself is either in a closed or open state depending on the time of day. The state the library is in changes the actions that can be performed.

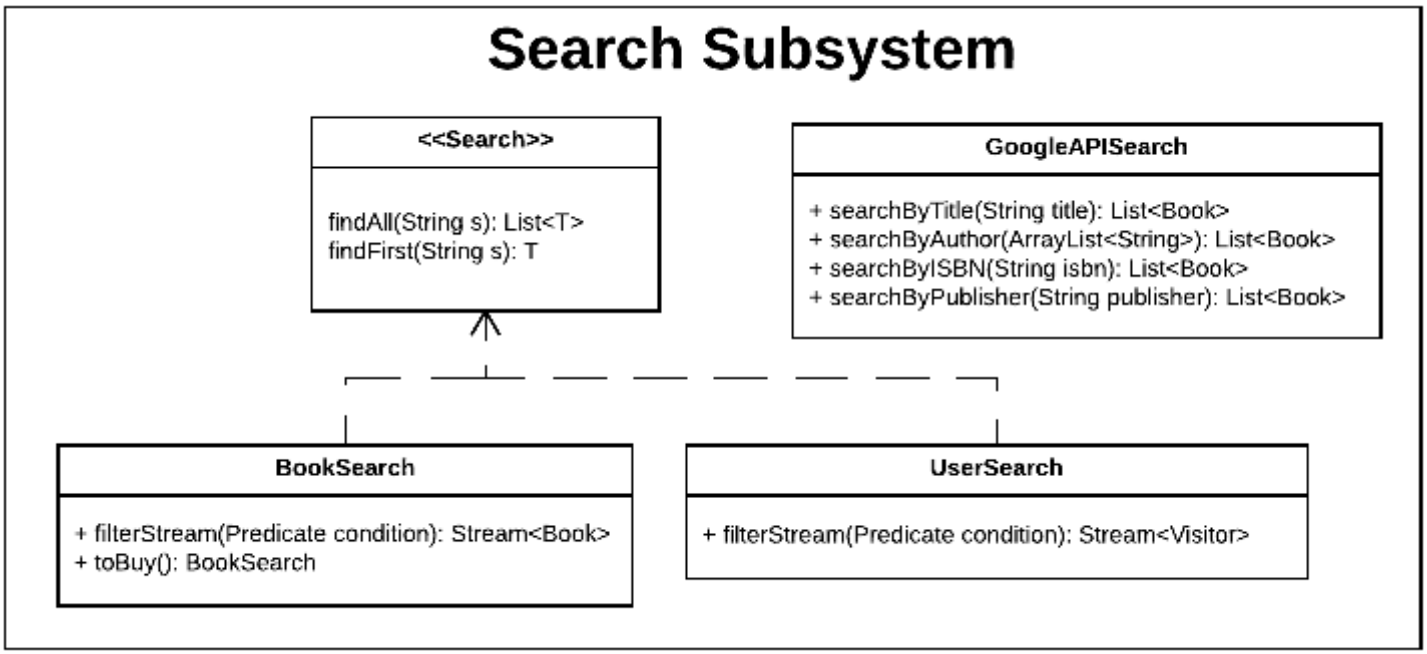

Models
SystemDateTime
run(): void
getInstance(): SystemDateTime
getTime(): LocalTime
getDate(): LocalDate
plusDays(long): void
plusHours(long): void
reset(): void
stopClock(): void
Book
hasAuthorPartial(String): boolean
checkOut(): void
undoCheckOut(): void
returnBook(): void
undoReturnBook(): void
dateFormat(): String
compareTo(Book): int
purchase(): void
undoPurchase(): void
ISBN
isValid(): boolean
equals(Object): boolean
Transaction
closeTransaction(): void
<<LibraryState>>
isOpen(): boolean
OpenState
isOpen(): boolean
ClosedState
isOpen(): boolean
PhoneNumber
isValid(String): boolean
Visit
depart(): void
unDepart(): void
Visitor
canCheckOut(): boolean
checkOut(Transaction): void
undoCheckOut(Transaction): void
returnBook(Transaction): void
undoReturnBook(Transaction): void
switchInLibrary(boolean): void
payFines(): double
setCredentials(String, String): void
Session
setV(Visitor): void
addUndoable(Undoable): String
undoUndoable(): String
redoUndoable(): String
setSearch(SearchService): void
Employee
getUsername(): String
getPassword(): String
1
0..*

Name: System Clock		**GoF pattern:** Singleton
Participants		
Class	**Role in GoF pattern**	**Participant's contribution in the context of the application**
SystemDateTime	Singleton	This class is responsible for keeping track of the system time for the library book management system. It is run on a separate thread to avoid an incorrect time due to processing of the rest of the program. When the system is started a new system date time object is created, after that the instance of the first created one is returned to follow the singleton pattern.
Deviations from the standard pattern: No deviations.		
Requirements being covered: Only one instance of a class can be given at any time, there can only be one clock for the system.		

Name: Library State		**GoF pattern:** State
Participants		
Class	**Role in GoF pattern**	**Participant's contribution in the context of the application**
LibraryState	Interface	This is the interface for all the state classes. It requires a state class to have an isOpen() method.
OpenState	ConcreteState	This class is used to represent the state of the library between the hours of 0800 and 1900 (System Time).
ClosedState	ConcreteState	This class is used to represent the state of the library between the hours of 1900 and 0800 the following day (System Time). When in this state, the library restricts access to commands.
Deviations from the standard pattern: None		
Requirements being covered: The system only has one given state at a time and the system can only complete certain actions from any given state.		

SEARCH SUBSYSTEM

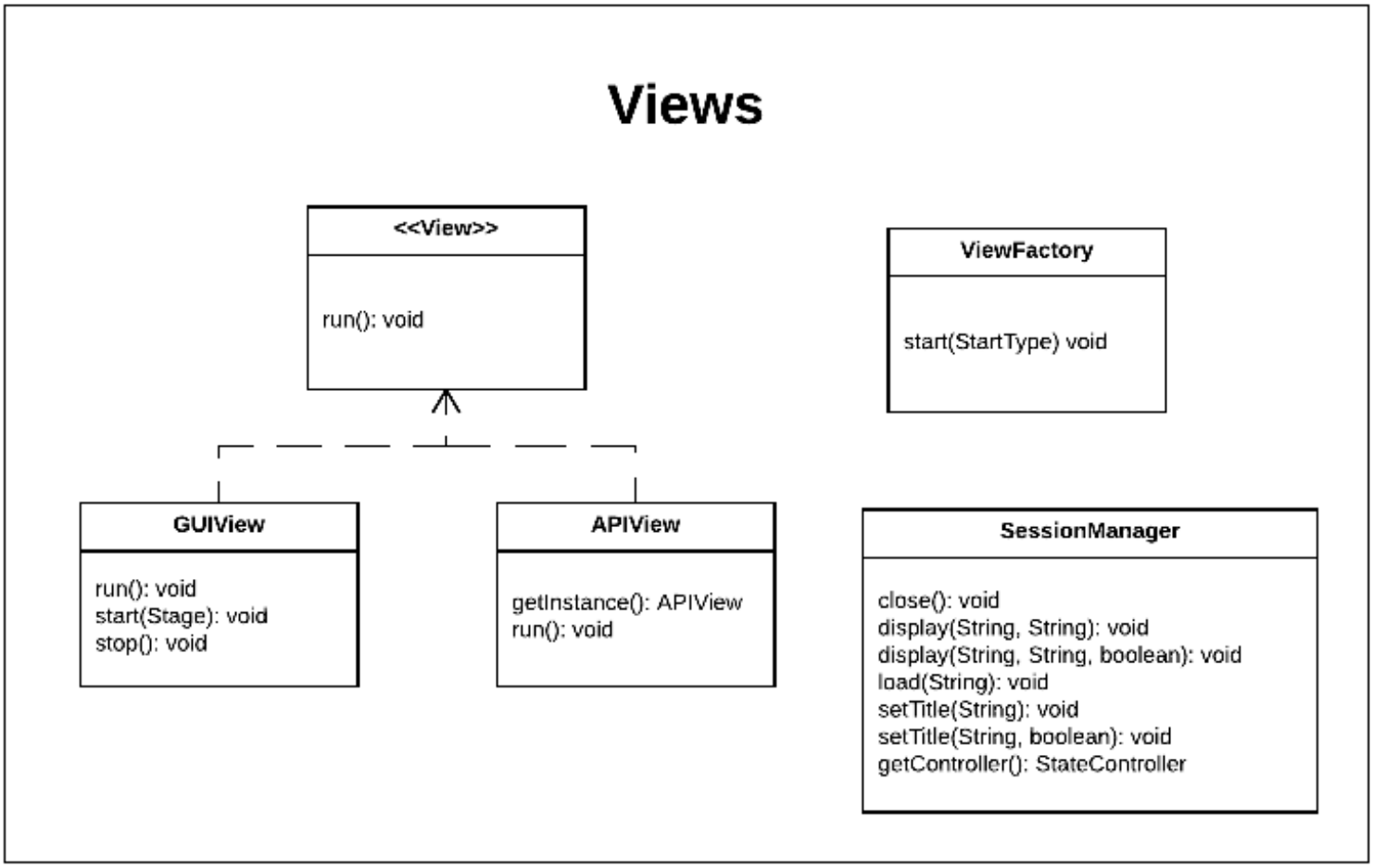
The Search Subsystem implements the Strategy design pattern, inheriting from a single interface, Search, and implementing different algorithms in each subclass. The strategy design patterns allow us to add additional searching algorithms quickly and efficiently, without having to modify the rest of the system. We used two main classes, BookSearch and UserSearch, to search the system. Both of those classes searched the local data of the program. For R2 we added another class, GoogleAPISearch, which uses Google Books to find additional books that can be purchased. However, it does not implement the Search interface and is not really part of the design pattern.



Name: Search		**GoF pattern:** Strategy
Participants		
Class	**Role in GoF pattern**	**Participant's contribution in the context of the application**
Search	Interface	This interface is used to declare the search() and findFirst() methods to be implemented in the concrete implementations. These methods are common to all searches and contain the ability to find all and find one object that matches given criteria, respectively.
BookSearch	ConcreteStrategy	This class contains the steps specific to searching for a book object in the system.
UserSearch	ConcreteStrategy	This class contains the steps specific to searching for a user object in the system.
Deviations from the standard pattern: • This implementation made use of Java enums to contain different ways of searching for a particular object type within the same class, while keeping the search for different types of objects in separate classes.		
Requirements being covered: • The application must provide the ability to search for objects in different ways, depending on user input. • Each class implementation provides different ways of searching while representing a specific instance of the general action: search.		

VIEWS SUBSYSTEM

The Views Subsystem plays two crucial roles in our system. It acts as the View element from our MVC architecture, but also implements the State and Factory patterns. The Library Book Management System can only have one mode at a time that is decided on with program arguments when run. The two available modes of the system are API or GUI, API sends requests to the system and prints responses, GUI launches the graphical user interface. The mode of the system is controlled by the ViewFactory class, which uses the Factory pattern.



Name: View State		**GoF pattern:** State
Participants		
Class	**Role in GoF pattern**	**Participant's contribution in the context of the application**
View	Interface	This is the interface for all the state classes, it requires a state class to have the run() method.
APIView	ConcreteState	This class is used for starting the API mode of the Library Book Management System.
GUIView	ConcreteState	This class is used for starting the graphical user interface of the Library Book Management System.
Deviations from the standard pattern: None		
Requirements being covered: The system can only run one mode at a time, either API or GUI.		

Name: ViewFactory		**GoF pattern:** Factory Method
Participants		
Class	**Role in GoF pattern**	**Participant's contribution in the context of the application**
ViewFactory	ConcreteCreator	This class selects the appropriate View and runs it.
View	Product	This is the interface the ConcreteProducts must implement.
APIView	ConcreteProduct	This class is used for starting the API mode of the Library Book Management System.
GUIView	ConcreteProduct	This class is used for starting the graphical user interface of the Library Book Management System.
Deviations from the standard pattern: No interface used for the creator.		
Requirements being covered: The system can only run one mode at a time, either API or GUI.		

CONTROLLERS SUBSYSTEM

The Controllers Subsystem is contained in the controllers package and contains two sub packages, commandproxy and guicontrollers. The commandproxy package is used to control the Command Subsystem and incorporates the Proxy pattern. This allows us to determine if a command can be executed based on user permissions, allowing visitors and employees to have different abilities within the system. It also contains the ParseResponseUtility class which is not part of the design pattern, but is used to take the information outputted by a command and format it for use with the GUI. The guicontrollers package contains several sub-packages and classes that interact with the GUI, following the MVC pattern. These controllers are the part of the system that connect the JavaFX code to the Command Subsystem.

ADD CONTROLLERS UML HERE OR NEXT PAGE PREFERABLY

Name: Proxy Command		**GoF pattern:** Proxy
Participants		
Class	**Role in GoF pattern**	**Participant's contribution in the context of the application**
ICommandController	Subject	Interface for the command controller, requires the processRequest method that takes a string as a parameter and returns a string.
ProxyCommandController	Proxy	This class implements the ICommandController interface. It checks for the user's status (Visitor or Employee) and calls processRequest in the CommandController if it allowable.
CommandController	RealSubject	This class also implements the ICommandController interface. It creates the appropriate command and executes it, directly interacting with the Command Subsystem.
Deviations from the standard pattern: None		
Requirements being covered: Uses the Proxy design pattern to control how commands are processed by the system based on the user and state of the library.		

APPENDIX

ADD CRC CARDS HERE AND ORGANIZE BY PACKAGE

ALSO ANNOTATE THE DOCUMENT SOMEHOW TO INDICATE CHANGES FROM R1

ADD SEQUENCE DIAGRAMS AND STATE DIAGRAMS TOO