<PROJECT & RELEASE>

Design Documentation

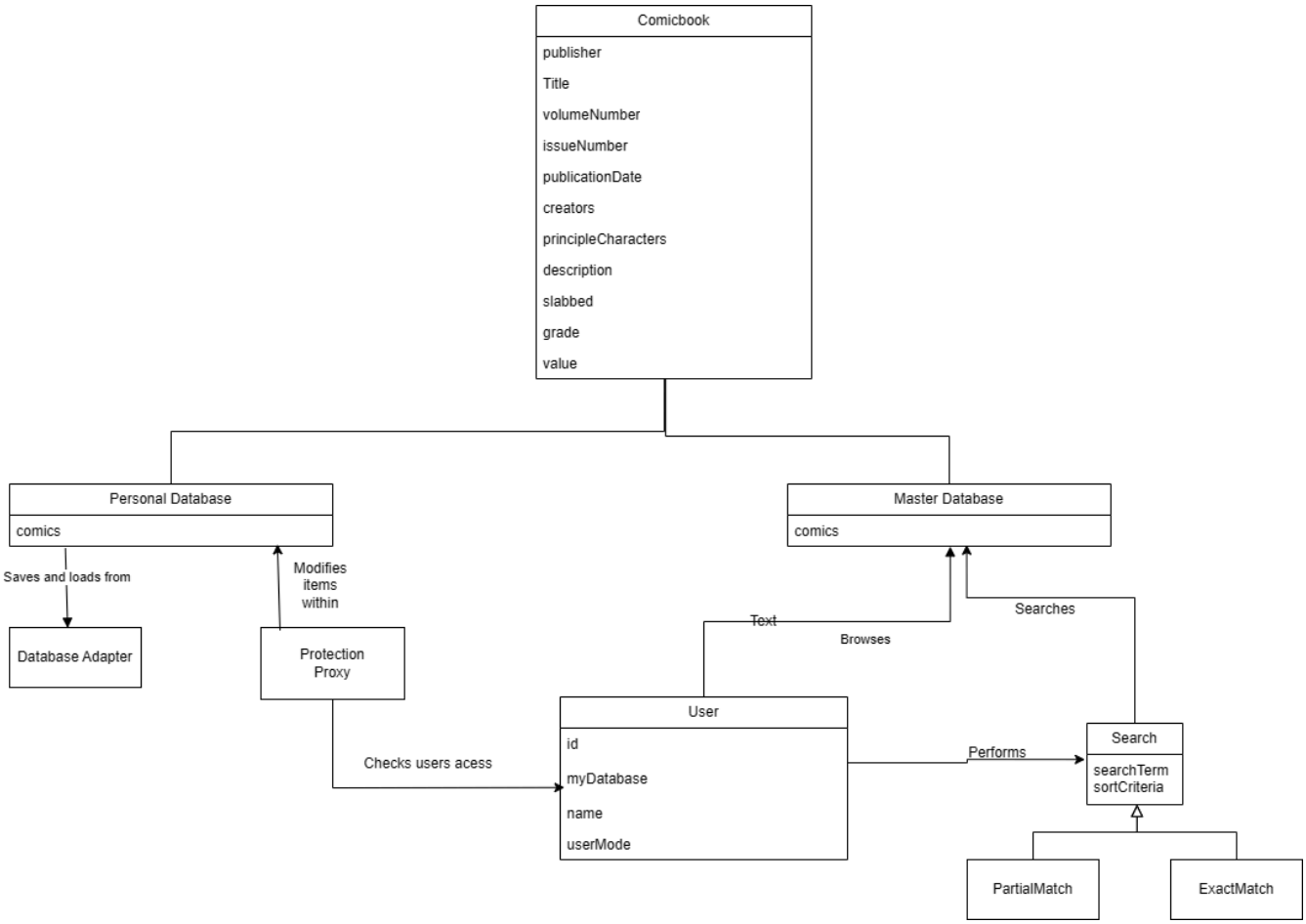
Table of Contents:

Summary

Comix is an app for users to have their own collection of comics. These comics exist in a Master Database of comics that a user can search through to find their desired comic. From their browsing they can add these comics to their own personal collection of comics. Users can manage this personal collection by adding and removing comics as well as grading,slabbing, signing and authenticating them which changes their value.

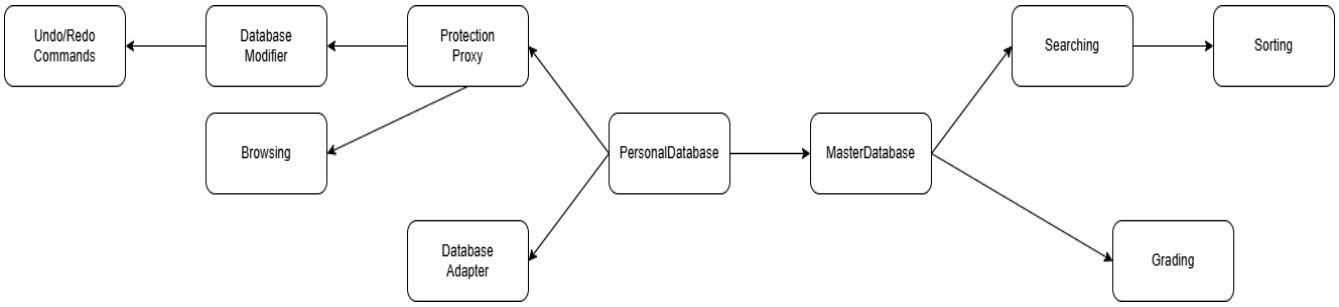
Domain Model

This section provides a domain model for the project. It should follow the guidelines discussed in class and the design project activity sheets. For it to be readable, you may need to turn this page into landscape mode.



System Architecture

The personal database uses the protection poxy and database adapter to see what permissions the current user has and what type of database is being read. The protection proxy uses the database and browning subsystems to add edit and look through all comics that a user has depending on if they are a guest or verified user. The personal database gets its comics from the master database that holds all of the comics. The master base can be searched through with the searching subsystems and further sorted with that one. It also uses the grading subsystem for individual comics that are in the database.



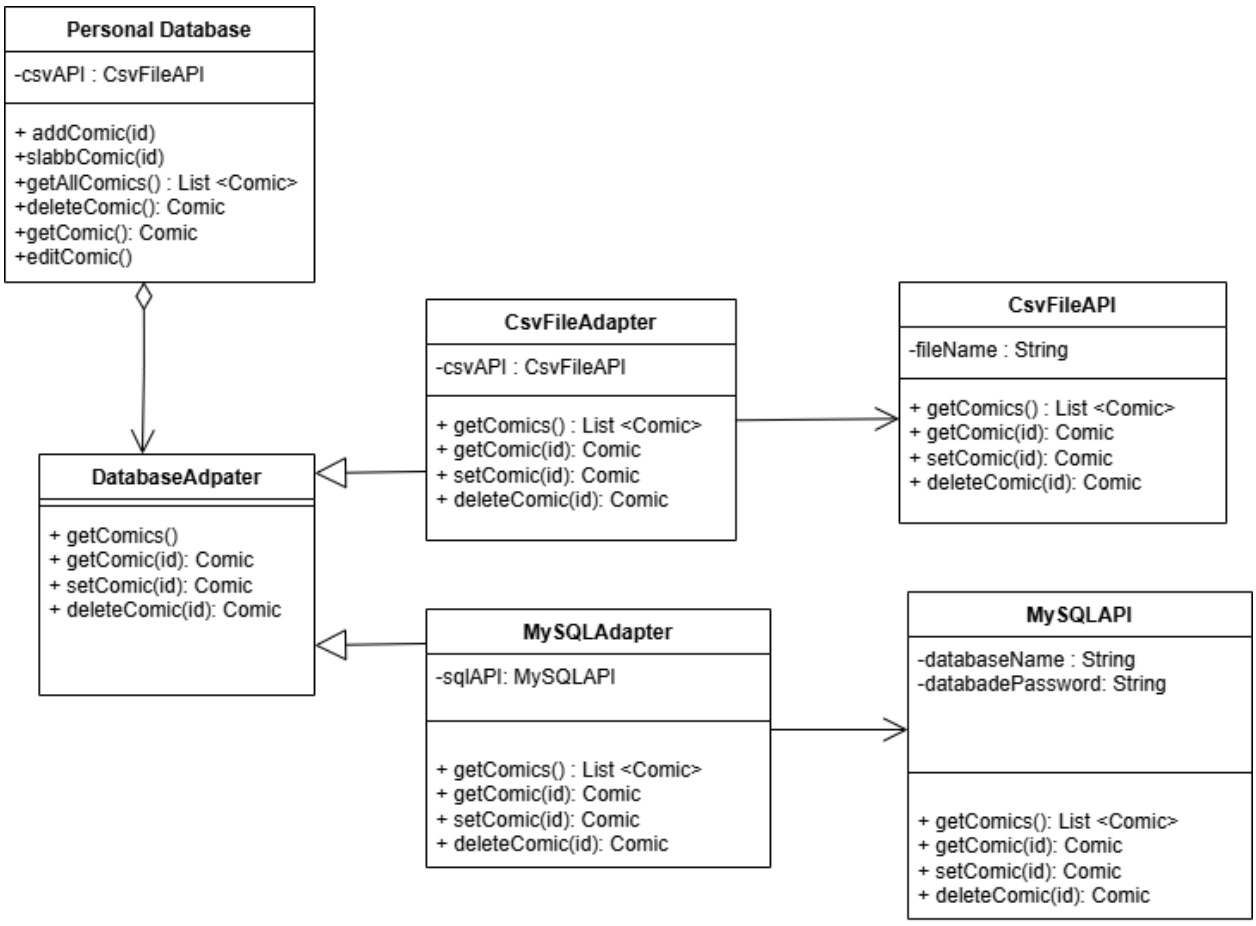
Subsystems

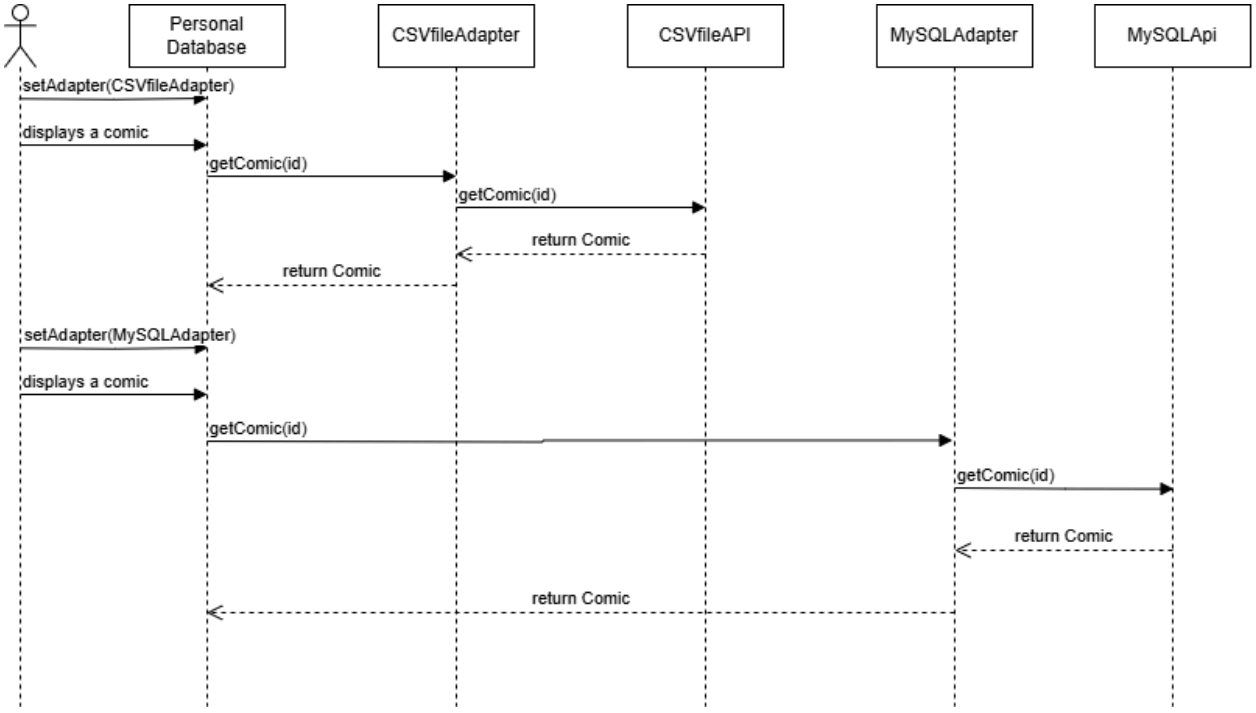
Database Adapter

In part 2 of the non-functional requirements for COMIC part two it states that while right now the software “needs to only export to/import from files” future versions of this application will “require the ability to save to or load from a real database (such as MySQL, PostgreSQL, Oracle, etc.).” So we have to design the personal database to not only be able to take and send information to a CSV file but also in the future keep it open and easy for you to get information from a SQL database instead and that is where the Adapter design pattern comes in.

The adapter design pattern makes it so the client doesn’t have to access any of the different file types directly. Instead, it creates a general Adapter, that all adaptees inherit allowing for all different types of databases to be able to be switched out easily without changing the class calling it at all. This makes it an ideal class to add for a system where we know we will have to add further database options down the line, but aren’t entirely sure what they’ll be. This comes down to the open/closed principle which states that systems shouldn’t be open for extension but closed for modification. With this extra class, we won’t have to modify any parts of our existing classes to change what database we’re sourcing from.

This will impact the system by increasing the cohesion and lowering the coupling. This is because each class will have fewer responsibilities due to there being more of them, but it will still decrease the coupling because you have fewer classes that rely directly on each other. The adapter is an example of pure fabrication so by being ago-between for these two classes it can achieve both.



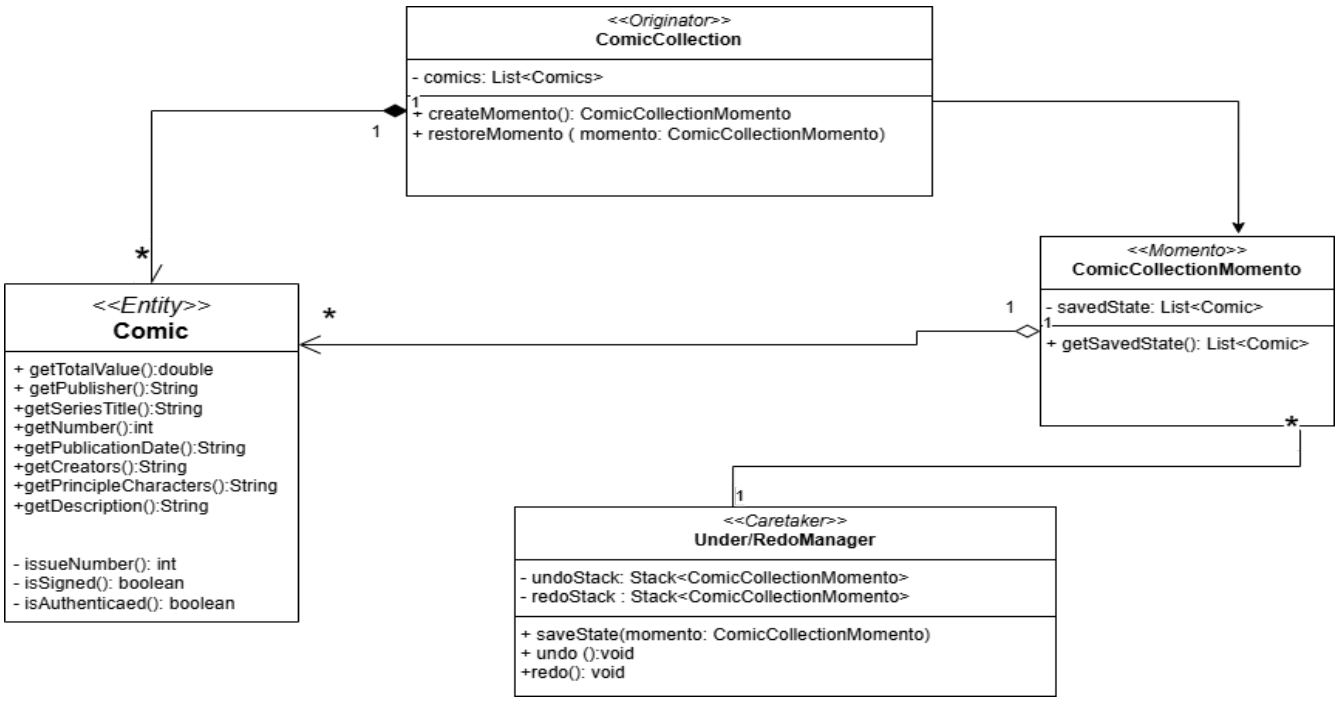


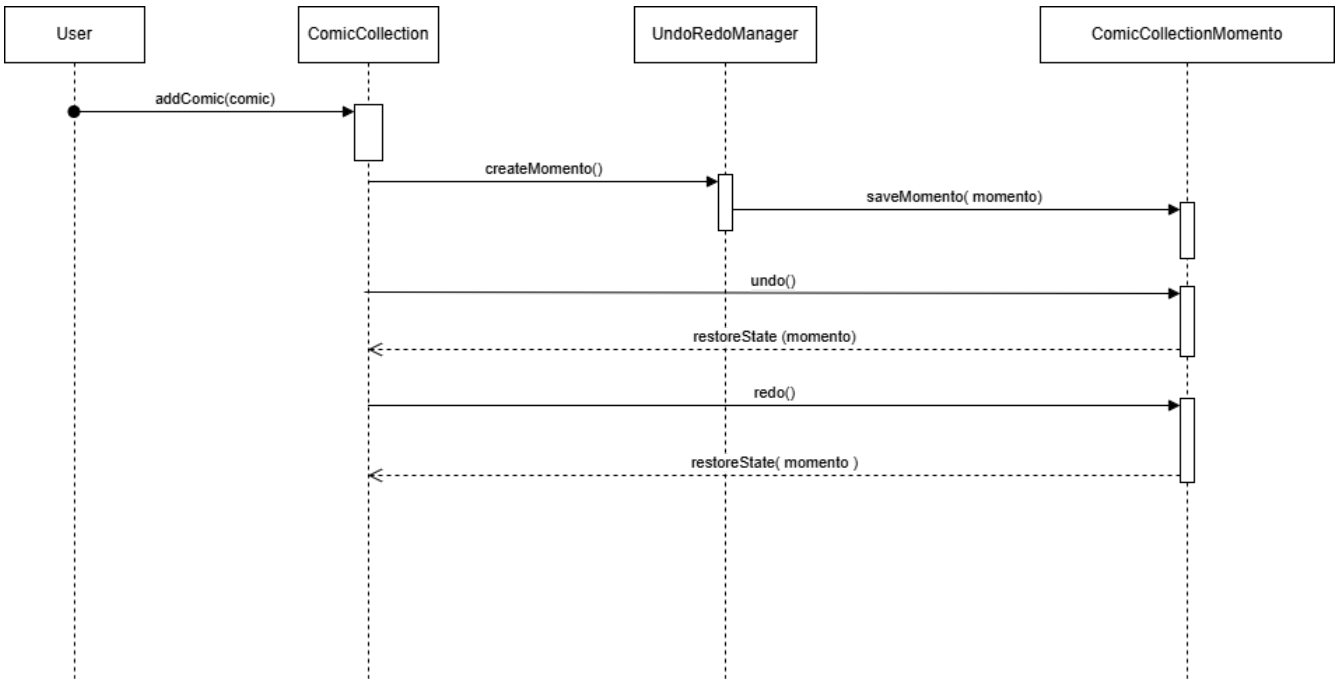
Subsystem Name: Database Adapter	GoF Pattern: Adapter

Participants		
Class	Pattern Stereotype	Participant's contribution in the context of the application
DatabaseAdapter	Target Adapter	A basic implementation of the adapter that all the other database adapters inherit. It has methods to get, set, and delete comics.
CSVFileAdapter	Adapter	An implementation of the database adapter class for writing intCSV CSV files. It contains a csvfileAPI and calls into it to get all the info for its methods.
MySQLAdapter	Adapter	An implementation of the database adapter class for accessing a MySQL database. It contains a mySQLAPI and calls into it to get all the info for its methods.
CSVFileAPI	Adaptee	An API that writes and reads into a CSV to store information about comics in a personal database. It has to get, use, and delete methods.
MySQL API	Adaptee	An API that gets and sets information from a MySQL database to store information about comics. It has get, set, and delete methods.
Deviations from the standard pattern: None		
Requirements being covered: Non-functional requirements, part 2,		

Undo/redo

In the requirements it states, “the user shall be able to undo commands that have modified the contents of their personal collection during an active session.” The memento pattern is perfect for this implementation by capturing a snippet of a comics state before modifications. Each time a user changes their personal collection (add, removing, or signed) a memento object stores the previous state. If a user wishes to undo an action, it can be restored from the last saved state from a stack data structure. This pattern adheres to the single responsibility principle by creating an originator and then a memento maneges any changes. A trade off is that even though this approach increases the memory usage since a memento stores a full snapshot of the state of the collection, this is still acceptable because this pattern is straightforward without tracking individual operations.


<<Originator>>
ComicCollection
- comics: List<Comics>
+ createMomento(): ComicCollectionMomento
+ restoreMomento (momento: ComicCollectionMomento)
<<Momento>>
ComicCollectionMomento
- savedState: List<Comic>
+ getSavedState(): List<Comic>
<<Entity>>
Comic
+ getTotalValue():double
+ getPublisher():String
+getSeriesTitle():String
+getNumber():int
+getPublicationDate():String
+getCreators():String
+getPrincipleCharacters():String
+getDescription():String
- issueNumber(): int
- isSigned(): boolean
- isAuthenticaed(): boolean
<<Caretaker>>
Under/RedoManager
- undoStack: Stack<ComicCollectionMomento>
- redoStack : Stack<ComicCollectionMomento>
+ saveState(momento: ComicCollectionMomento)
+ undo ():void
+redo(): void


User
ComicCollection
UndoRedoManager
ComicCollectionMomento
addComic(comic)
createMomento()
saveMomento(momento)
undo()
restoreState (momento)
redo()
restoreState(momento)

Name: UndoRedo System		GoF Pattern: Memento
Participants		
Class	Role in Pattern	Participant's Contribution in the context of the application
comicCollection	originator	The ComicCollection class is responsible for managing the user's personal collection of comics. It creates and restores states by interacting with the memento. When changes are made, it generates a comic collectionMomento to capture the state of the collection at that point.
UndoRedoManager	Caretaker	The UndoRedoManager manages the history of mementos. It is responsible for pushing and popping the mementos from its stack. This enables the class to revert or reapply changes.
ComicCollection/momento	Memento	The ComicCollectionMementos class captures and restores the state of the ComicCollection at a given moment. This class does not modify the collection, however it allows the collection to be restored to a previous state.
Deviations from the standard pattern: the UndoRedoManager supports an unlimited number of undo and redo operations.This implementation stores multiple mementos to support continuous undo/redo actions throughout a session.		
Requirements being covered: 2. Undo and redo commands that modify the personal collection.		

Status of the Implementation

Everything from R1 is implemented and works except for the command subsystem which still needs work.