PLATO

Final Report

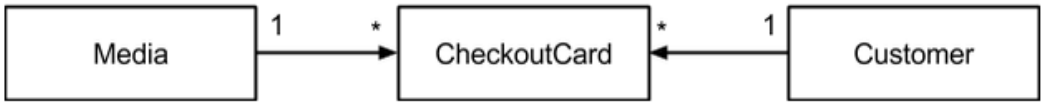
By: Shelby Hockey, Alex Pogue, Taylor Welter

A. Research

A lot of time was spent learning about the best ways to utilize a database. We started out using JDBC as a simple way to communicate with a database. Then we discovered JPA implementations such as EclipseLink and Hibernate. These tools allowed us to treat my database entities as actual objects. Lastly, we found a way to describe complex relationships between objects in the database using JPA annotations.

First I learned about setting up and using PostgreSQL and JDBC. I learned how to create and execute queries safely using prepared statements. Prepared statements are an alternative to concatenating user input directly into the query string. Without prepared statements, the user could directly type wicked SQL query commands to do evil things to the database, such as drop all tables. Another big benefit of JDBC was its backend independence. A change of just a few lines of code would make the code work with MySQL, for example.

Once that was working with a few persisted objects, we saw we had a few more objects to persist which had more complicated relationships with each other. Below is a drawing of the relationships between Media, Customer, and CheckoutCard. Each media has multiple CheckoutCards associated with it, one for each time it was checked out. Customer also has multiple CheckoutCards, one for each piece of media that the customer checked out.



This would have been a challenge to implement using JDBC.

Another problem is the code length. For example, it took 35 lines of code to simply get a book object from the database (see Iteration One DatabaseSupport.gfetBook()). It was difficult to add new objects to the database as we had to write all this code, and much of the code was copy and pasted from other database access methods.

We then discovered (through Dr. Leslie Miller's suggestion) Java Persistence API (JPA). JPA is an API that assists management of relational data in a Java program. Implementations of JPA include Hibernate and EclipseLink, among others. We chose to use EclipseLink because it adheres closely to the JPA standard, and we did not want to use any Hibernate-specific functionality.

Basically, JPA allows the programmer mark a class with an @Entity annotation and then store objects of that type into the database using its members. Then, any time the programmer wants access to a stored object, the object can be easily loaded back into memory. This relieves the programmer from mapping database columns to their associated object fields.

Using JPA, it only takes 2 lines (previously 35 lines using JDBC) of code to retrieve a book object from the database. In fact, the source code commit in which I converted the JDBC database accesses to JPA accesses removed 473 lines of JDBC code, while only adding 269 lines of JPA code.

Using JPA, we mapped the Media, Customer, and CheckoutCard using a @OneToMany mapping from Media to CheckoutCard, and another and another @OneToMany mapping from Customer to CheckoutCard. We used two @ManyToOne annotations in the CheckoutCard class which point to its corresponding Media and Customer. With JPA, it was reasonably simple to implement that relationship.

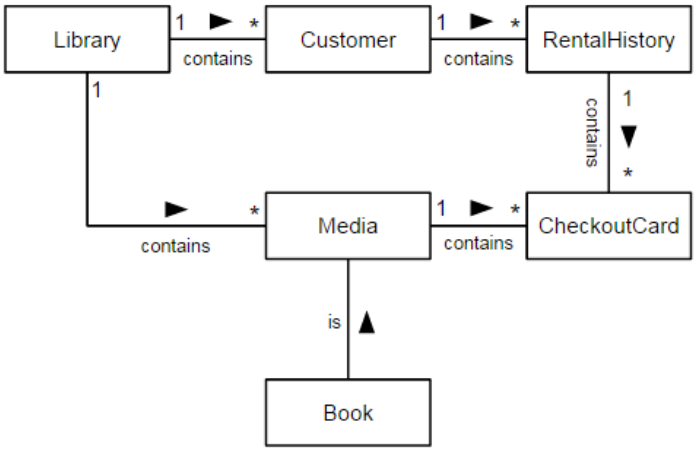
B. A section describing the target audience for the project and how you identified that you met their needs.

The intended audiences for our project were the librarians and, to a lesser extent, the patrons of a library. We would create a library management system that would track the inventory and the loans of various forms of media to its patrons, thus simplifying things for the librarians. To start, we visualized what libraries were like before/without computer systems: with card catalogs and a checkout slip within a book (or another type of media) to receive a signature and date stamp—we essentially wanted to capture that system and make it slightly more efficient, for the convenience of librarians and patrons alike.

C. Description of the project design.

The objective of this project was to create a library management system. The Admin will have privileges to add books, delete books, edit the books (i.e. change the call number), and have all of the options available to the Librarian. The Librarian should be able to check in, check out, and renew books. Librarians can also get customer information (i.e. see how many books a Customer has checked out and any fees) and all information about a book including book checkout history, number of copies of this book, location, and other book information. The Customer will have the opportunity to search for a book and see what books they currently have checked out with due dates, and any fees.

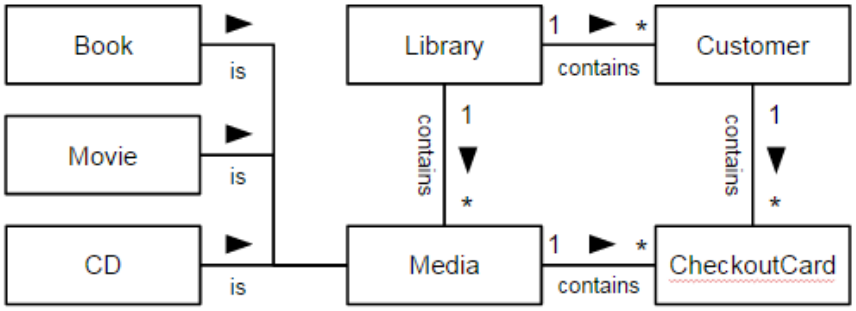
To implement this we started with the UML Diagram showed below.



First attempt at a Domain Model UML diagram

We wanted to keep a comprehensive list of items a customer had checked out, past and present. The object to hold this historical data was to be called rental history. Within the rental history we would have a list of "check out cards". These would have the media title, date of check out, date it was checked back in and any fees the customer may have obtained. Then Media would also contain a list of all checkout cards, for all customers, and media.

After some database experimentation, we discovered that we could get rid of RentalHistory and simply allow the customer to own its own CheckoutCards. We also added multiple types of media into the domain model. To integrate these design changes, we arrived at the following domain model:

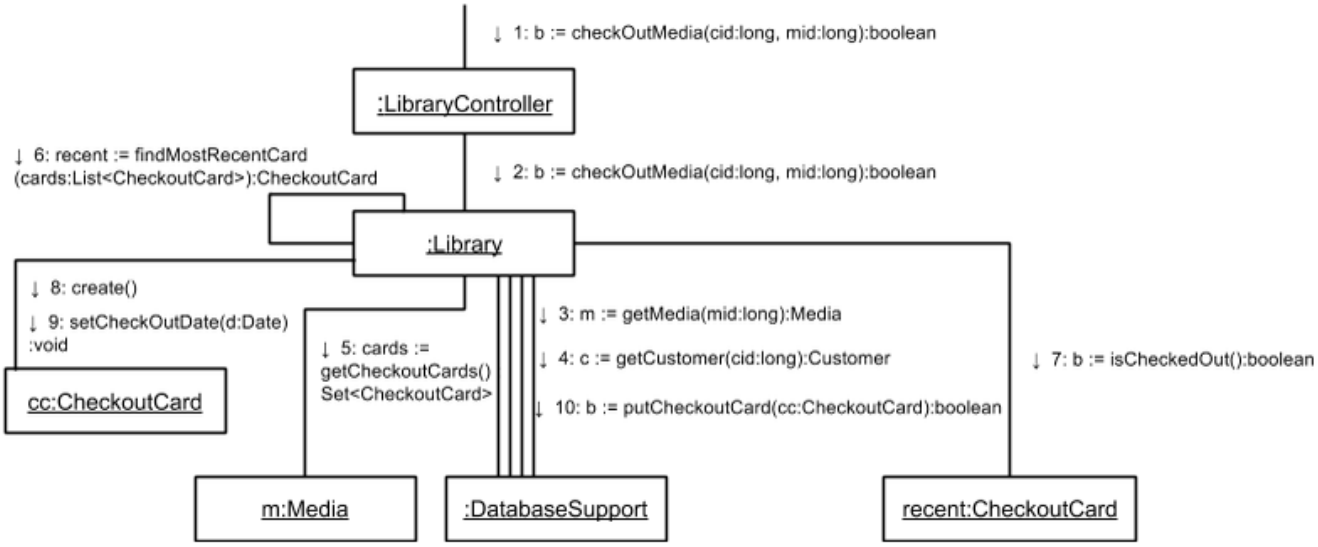


The final domain model with RentalHistory removed and media types added.

D. A section describing the system implementation.

PLATO is organized using a variation of model-view-controller. The model is DatabaseSupport, the controller is LibraryController, and the view is platoUI.

The application is data-oriented. Most of the functionality is in storage and retrieval of media, checkout cards, and customers. Therefore much of the implementation is in the database and making sure to store and retrieve data in the correct way. For example, one of the more complex use cases is checking out of a piece of media. The only computational operation in this use case is finding the most recent checkout card of the media, and most of the code is database reads and writes. The interaction diagram is given below:



Interaction diagram for Check Out Media use case

As can be seen from the interaction diagram, most of the work is done interacting with the database and reading objects retrieved from the database.

E. Interval breakdown.

Iteration One

For the first iteration, we set up the basics for database. We implemented storage and retrieval of Media, Customers, and CheckoutCards. We also implemented some basic functionality of a library. We decided to implement functions like check in and checkout directly to media. Then the “add” and “edit”

functions would be implemented by each individual media type. We focused our implementation efforts on the book media type. In this we had use cases for add book, delete book, and edit book. Originally, we had edit book as one use case but later decided to divide it into 3 different edit methods; edit title, edit author and edit publisher. Once we had added books, we needed to create customers. To implement customers, we started with use cases for add, delete, and edit customers. By the end of the iteration we had implemented books and customers. We also have set up the check in, check out, and set the late policy for books. We believed this to be the essential functionality to set up and begin to implement the library application.

Iteration Two

For Iteration Two, we implemented functions in order to create a functional simulation. To begin the iteration, we finalized the checkout process in the database. In order to check out the media, we must determine if the media is already checked out. Then when the media is checked in we needed to determine if it was turned in on time, if it wasn't we would calculate the late fee. If a customer is being charged a late fee, the librarian would then have to have the ability to let the customer pay the fee.

Now that the customers can return their books and pay the appropriate fees, we then had to set up the search function. The search function uses a field identifier and the search string. For technical simplicity, searches only identified entries with the exact same letters as the search query.

Lastly, we implemented log in and log out functionality. We also implemented functionality to view customer checkouts. We decided that only library staff will have that privilege. This allowed the customer's information to be viewed by appropriate audiences. At this point we also set up the functionality to view various book information such as the author, title, publisher, and title of each book within the system.

Iteration 3

As time progressed we discovered we were falling behind schedule, and we decided to scale back our original plans. As a team we decided that having the option to have CD's and movies was more important than being able to view a book's complete rental history. All of the functions of books were added to movies and CD's.

F. References.

EclipseLink API (https://www.eclipse.org/eclipselink/api/2.6)
JPA 2.0 EclipseLink tutorial (http://www.vogella.com/tutorials/JavaPersistenceAPI/article.html)
JPA One To Many tutorial (http://examples.javacodegeeks.com/enterprise-java/jpa/one-to-many-bidirectional-mapping-in-jpa/)
JPA JoinColumn vs mappedBy (http://stackoverflow.com/questions/11938253/jpa-joincolumn-vs-mappedby)

G. A user's guide as an appendix.

The PLATO system is a command line application which is broken up into menus. Open a command line shell and navigate to the bin folder of the project (CodeAtTimeOfDemo folder). Next run the following command:

```
java platoUI
```

and the application will begin running. It will first query the username and password. We have already set up a username "admin" and password "coms362" for you to try (pro tip: there are also other user type names available – all with the same password (e.g. "customer"). Logging in as different user types will give you access to different parts of the system.

Once you're inside the system, there are a set of commands. Many of them lead to other menus with more commands. Use of all the "new" commands should work fine, but, since you cannot query for a list of books in the current version, commands like "check out" will be confusing because you do not know specific id numbers of books or customers. This functionality has been tested in the demo previously.

Feel free to look around the application and try things out.

H. Appendix

A. Alex – Alex focused his attention on the database. Throughout the project, he researched various database types and their performance. He also evaluated code clarity within the database code. He made decisions of technologies to use – started with JDBC, and later moved to JPA and EclipseLink. As well as being involved in each iteration's main use case implementation, he also helped moderate decision-making discussions with regard to how each choice will integrate into the database setup.

B. Taylor – Taylor fulfilled a bit of a support role: first, getting everyone on the same page regarding the use cases and other methodology used in the class, then coding Shelby and his use cases during Iteration One, while providing backup to Alex as he figured out how to put JDBC in place within DatabaseSupport. During Iteration Two, he got Shelby up to speed with regards to the software system, and did some more coding both in the regular system and a method in the database. Then, during Iteration Three, Taylor suffered from a time crunch, and couldn't devote as much time toward the project, until just before the demo, where he created the User Interface.

C. Shelby – Along with the team stuff Shelby was responsible for doing her use cases for Iteration One and Two. Along with this she helped with the Java interfaces, class diagrams, and in Iteration Two she programmed her use cases above the database level, but no further, as she did not have any knowledge of creating databases. In Iteration Three, Shelby did all of the use cases, interaction diagrams, Java interfaces, class diagrams, and programmed all of the use cases again with the exception of those that required database support.

D. Together – As a team we decided to simulate a library system, adding, deleting and checking in/out different media. From this we developed a domain model and decided on what use case we would implement and which iteration these would be completed during. We also worked on the final report as a team, by delegating sections for each person to complete.

For Iteration One, we split the nine use cases evenly. Soon after we discovered some use cases where not relevant and others needed to be broken up into multiple different use cases. Then, when it came to implementing the code for Iteration One, Alex and Taylor did a majority of the work as they set up the database with JDBC. With Iteration Two, we divided the use cases evenly again, and since it was already mostly established, there wasn't a whole lot of database work. We each implemented our own use cases, class diagrams, Java interfaces, and implemented the code. Between Iterations Two and Three, Alex

revamped the database, switching the existing functionality from JDBC to JPA, Alex debugged and made sure the Database was working correctly and efficiently. Taylor created the user interface, and debugged. Shelby did all of the use cases, Java interface, class diagrams, and implemented the code up that did not interfere with the database.