Jeremy Neal
CMSC 461 - Databases
12/8/2014
Sleeman

Final Project Report

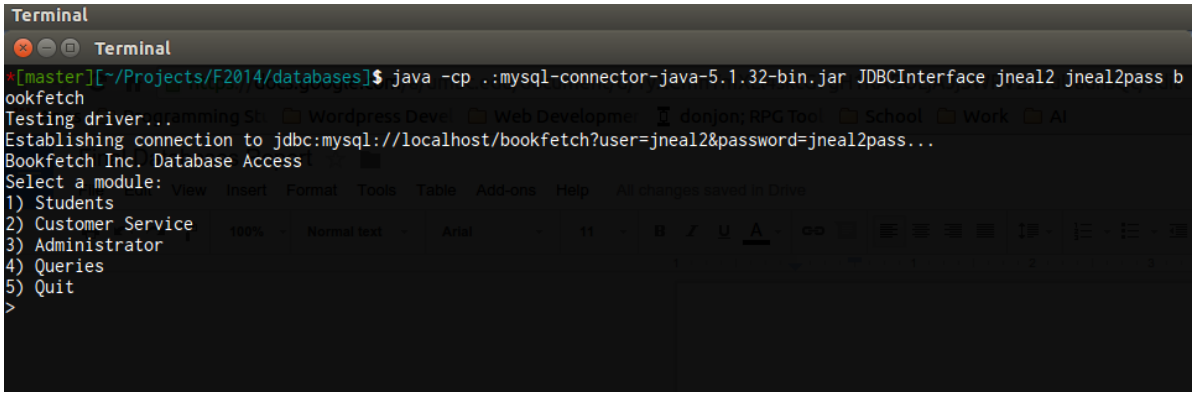
For the application portion of this project, I created a simple CLI application for interfacing with the database. MySQL is the underlying database, so the MYSQL-JDBC Driver is necessary. No additional libraries are included beyond the MySQL Driver. I considered using ncurses to provide a more robust interface while still staying on the command-line. However, no available bindings seemed to be well documented enough for the time table.

I did not change my previously created indices, though I did change some parts the database. Mostly, I changed fields that were being treated as numerics into varchars, primarily to maintain the proper number of zeroes. This is especially important in phone numbers, credit card numbers, SSNs and any other data that can have important leading zeroes. Some variable names were changed for ease of use, or to eliminate conflicts with keywords and any ambiguity.

The interface is a simple command line interface. If this were an actual application that would be distributed, additional work would have been put into packaging it into an executable, and providing more direct feedback from the command line.

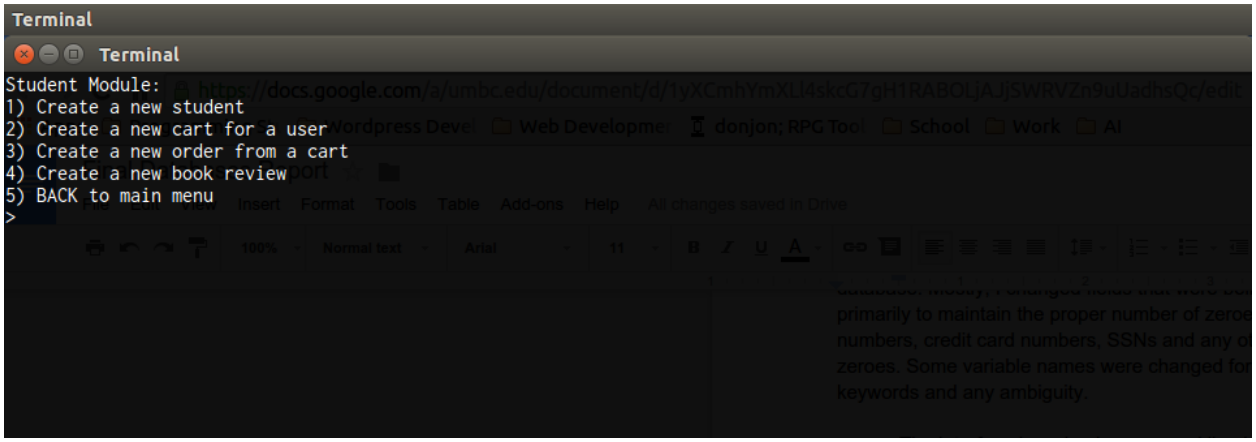
The application is called with: `java -cp .:mysql-jdbc-connector-5.1.32-bin.jar JDBCInterface username password database_name`. The class path may vary depending on where the .jar file is stored. Upon execution, the application first checks the command line arguments and, if they are sufficient, attempts to connect to the database.

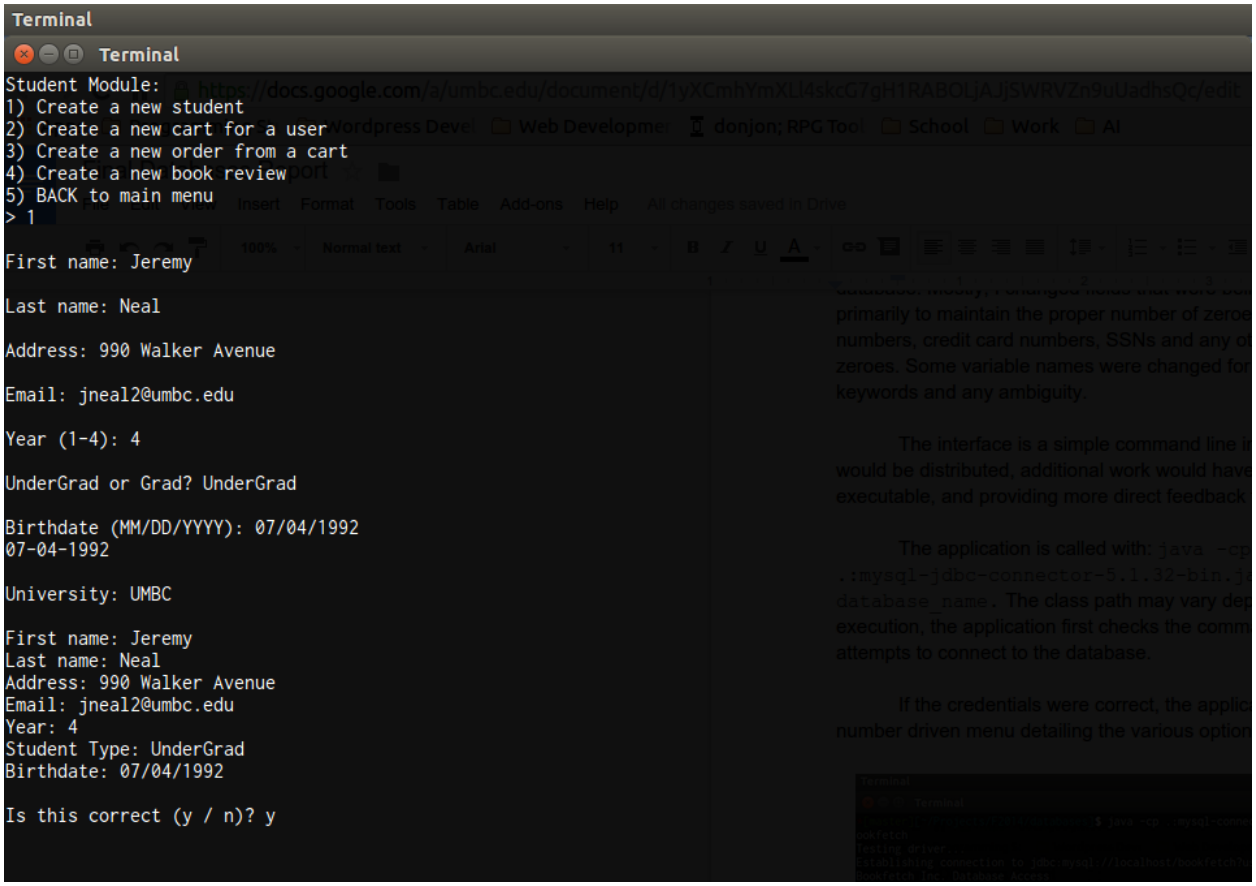
If the credentials were correct, the application will connect and present the user with a number driven menu detailing the various options they have.

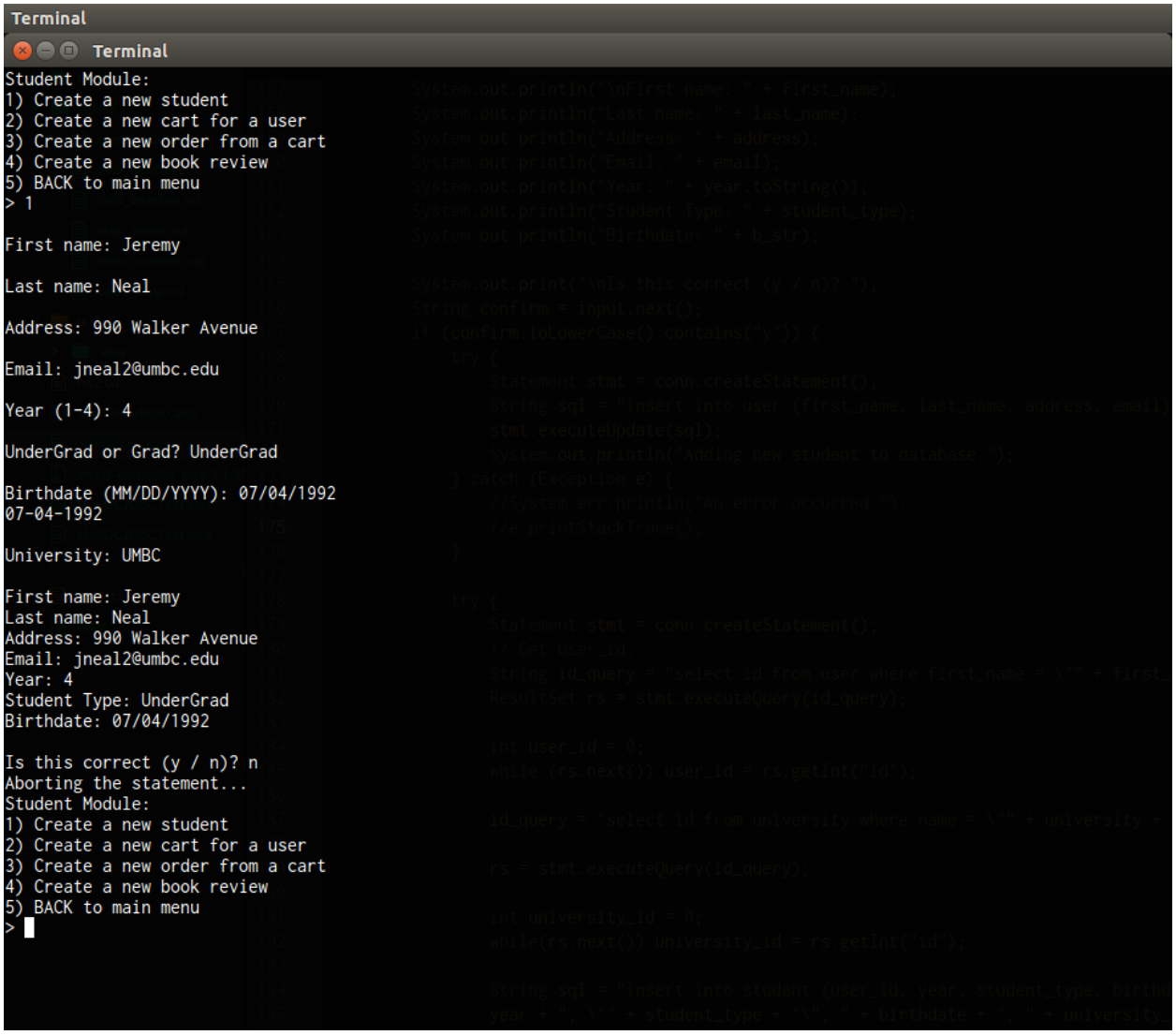


The first option contains the student module and requisite functionality. The second contains the customer service module, third is the administrator module, and so on. Within each module, there is another menu with the functions displayed.

Within the student module, the user (the person using the application, just to be clear), has the ability to create a new student add a new cart to a student, create an order from a cart, create a new book review, or go back to the main menu. With the exception of going back to the main menu, each option will ask for the required information to complete the given task. For example, to create a new student record, the user will be asked for a first name, last name, address, email, year, student type, birth date and university. This information will be used to query the database to see if the user already exists, and if not, create the user record to create the student record on top of. Other functions, like creating the cart, ask for a first name and last name to query for the user. The success or failure of certain functions is based off of the information given. For example, creating a new cart will fail if the student doesn't already exist in the system. It is considered beyond the scope of that particular function to generate a student record as well.




Terminal
Terminal
Student Module:
1) Create a new student
2) Create a new cart for a user
3) Create a new order from a cart
4) Create a new book review
5) BACK to main menu
> 1
First name: Jeremy
Last name: Neal
Address: 990 Walker Avenue
Email: jneal2@umbc.edu
Year (1-4): 4
UnderGrad or Grad? UnderGrad
Birthdate (MM/DD/YYYY): 07/04/1992
07-04-1992
University: UMBC
First name: Jeremy
Last name: Neal
Address: 990 Walker Avenue
Email: jneal2@umbc.edu
Year: 4
Student Type: UnderGrad
Birthdate: 07/04/1992
Is this correct (y / n)? y


Terminal
Terminal
Student Module:
1) Create a new student
2) Create a new cart for a user
3) Create a new order from a cart
4) Create a new book review
5) BACK to main menu
> 1
First name: Jeremy
Last name: Neal
Address: 990 Walker Avenue
Email: jneal2@umbc.edu
Year (1-4): 4
UnderGrad or Grad? UnderGrad
Birthdate (MM/DD/YYYY): 07/04/1992
07-04-1992
University: UMBC
First name: Jeremy
Last name: Neal
Address: 990 Walker Avenue
Email: jneal2@umbc.edu
Year: 4
Student Type: UnderGrad
Birthdate: 07/04/1992
Is this correct (y / n)? n
Aborting the statement...
Student Module:
1) Create a new student
2) Create a new cart for a user
3) Create a new order from a cart
4) Create a new book review
5) BACK to main menu
>

7 0 6
80

Jeremy Neal
CMSC 461 - Databases
11/10/2014
Sleeman

Project - Phase 3

The attached diagram shows the conversion of the entity-relationship model from Phase 2 to the relational model.

Several considerations were made in the design of this system. The strong entities from the ER diagram were taken and created into the main tables in the relational model. This included the User class, books, courses, universities, etc. In general, these items corresponded to objects of some kind, whether it was a person or a data point. Once these entities were accounted for, other entity sets were taken into consideration, and the relationships between them were implemented. While transforming our models from ER to relational, composite attributes were reduced to individual values as necessary. For example, names were often tracked as a composite attribute with a subattribute of first name and last name. These were translated to individual values.

In the ER diagram, there is specialization in the User model. To show this in the relational model, I chose to create a User base class that corresponded to all the users in the system, students and employees alike. The IDs of these entries were then used to link that user to their role, whether it was a student or employee. This seemed to provide a sufficient amount of specialization, and the use of more specific subclasses seemed unnecessary. The main values associated with these models don't change given further specificity; that is, we don't any more information about an admin or superadmin than we do about a customer service representative. Further specialization would only increase redundancy.

This is generally how relationships were described in the relational model. Relationships were shown using the primary key of a model (usually an ID value). For example, student users always have a cart. This cart can become an order. For the uses of our system, it worked well enough to retrieve these models using the ID of the student they are assigned to. A student doesn't need to have an order, but this relationship doesn't enforce such a condition.

One major design consideration was how to uniquely identify books. There were three attributes that, in conjunction, provide a satisfactory means of identifying books uniquely. These were the title, ISBN and ISBN-13 attributes. However, querying a database using three attributes can be a bit troublesome. It was decided to remove those values from the Book model, create a table called Book ID, and link those three values to a unique ID for each book. This way, those values are still accessible, but there is also a convenient means of uniquely identifying book selections.

An additional concern surrounding books was how to handle categories, subcategories and keywords. These are many-to-many relationships (books can have multiple values for each attribute, and these can be shared by different books). For each of these values, a table was created containing the corresponding book ID, and the associated value. This allows for multiple values for each. There would be some redundancy in the table itself as multiple books share values, but is a simple and efficient way of modeling the information. This doesn't include repeat information per book, which is a greater consideration.

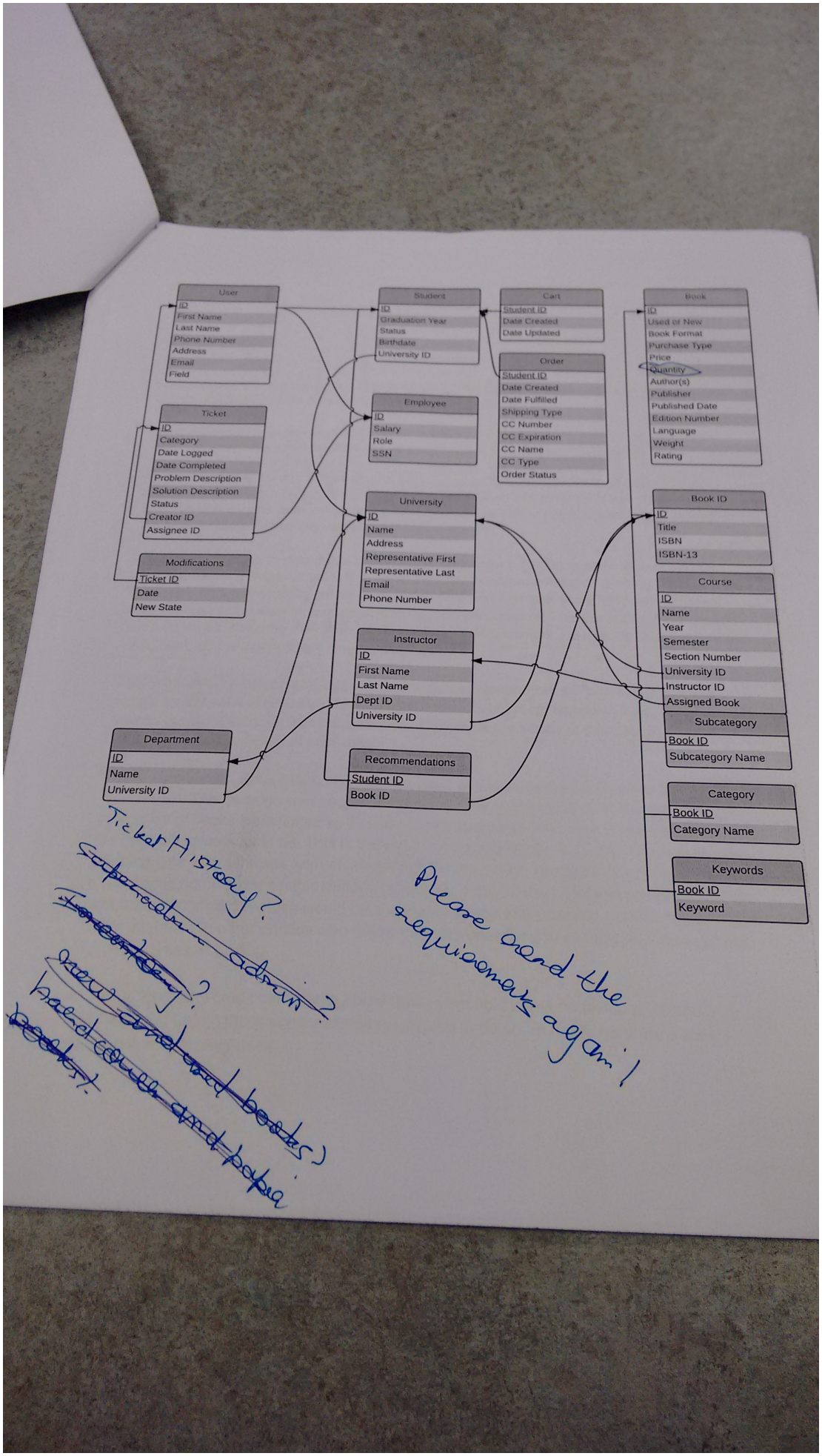
In regards to suggestions and criticisms on Phase 2, many of the recommendations felt inappropriate or incorrect. The first note was that a professor could work at two different universities and in two different departments. This seems extremely impractical and unlikely. The chances that a professor works at two different universities alone seems exceedingly rare. Moreover, given the academic specialization that professors often have, it seems equally odd that they would work in different departments as well. Given this judgment, I have chosen to disregard that note. In the rare case that this happened, the minimal redundancy incurred by creating a second, mostly redundant record would be negligible compared to the overhead involved in designing a system to handle such an extreme edge case. In any manner, there is a likelihood that professors at different universities might have the same name. The school and department that they teach at serve as discriminating factors.

It is completely possible that different books are used for different courses. Also, it is possible that different books are used for the same course (different professors could favor a different text). This criticism has been acknowledged and incorporated into the relational schema. The current schema acknowledges that the same course can have different book depending on which course section it is, who is teaching, what year, etc.

The role of an employee (admin, superadmin, customer service representative) is captured by the role attribute in the employee entity. As stated above, there is no additional information that needs to be stored for an employee based on their role, and this attribute would be sufficient for assigning privileges.

The condition and type of the book (e.g. used, hardcover) are handled under the book entity as well; the *used* and *format* attributes correspond to these values. They may need better names, but that was the intent. Additionally, the inventory count of a book is tracked under the *quantity* attribute.

In regards to missing relationships and the lack of a weak entity set, I've covered the only missing relationship I can, which is between a book and a course. I do not believe that a weak entity set is necessary to model the information accurately and efficiently.


User
ID
First Name
Last Name
Phone Number
Address
Email
Field
Student
ID
Graduation Year
Status
Birthdate
University ID
Cart
Student ID
Date Created
Date Updated
Book
ID
Used or New
Book Format
Purchase Type
Price
Quantity
Author(s)
Publisher
Published Date
Edition Number
Language
Weight
Rating
Order
Student ID
Date Created
Date Fulfilled
Shipping Type
CC Number
CC Expiration
CC Name
CC Type
Order Status
Ticket
ID
Category
Date Logged
Date Completed
Problem Description
Solution Description
Status
Creator ID
Assignee ID
Employee
ID
Salary
Role
SSN
University
ID
Name
Address
Representative First
Representative Last
Email
Phone Number
Book ID
ID
Title
ISBN
ISBN-13
Modifications
Ticket ID
Date
New State
Course
ID
Name
Year
Semester
Section Number
University ID
Instructor ID
Assigned Book
Instructor
ID
First Name
Last Name
Dept ID
University ID
Subcategory
Book ID
Subcategory Name
Department
ID
Name
University ID
Recommendations
Student ID
Book ID
Category
Book ID
Category Name
Keywords
Book ID
Keyword
Ticket History?
Please read the requirements again!

Grade: 91 \ 100

Jeremy Neal
CMSC 461 - Databases
10/27/2014
Sleeman

Project - Phase 2

The attached diagram lays out a high level design for the Book Fetch database application. Within, the relationships between various entities are defined.

A large part of the design was rigidly determined by the requirements listed in the document. However, there are some portions of this design which may warrant explanation.

One of the more obvious features is the generalization and specialization of users in the database. This was a rather straightforward way to reduce redundancy. While users have different abilities and responsibilities, the key attributes of most users overlap. This served as the best way to describe the users.

Additionally, it is worth noting that the distinction between a customer service user and an administrator is a single value, which could be either a numeric value or a string. The reasoning here is that while customer service users and administrators have very different responsibilities, the data stored about them is not very different. This serves as a simple and succinct way to describe the different users. Also, user privileges are outside the scope of this document, so the distinction is clear enough.

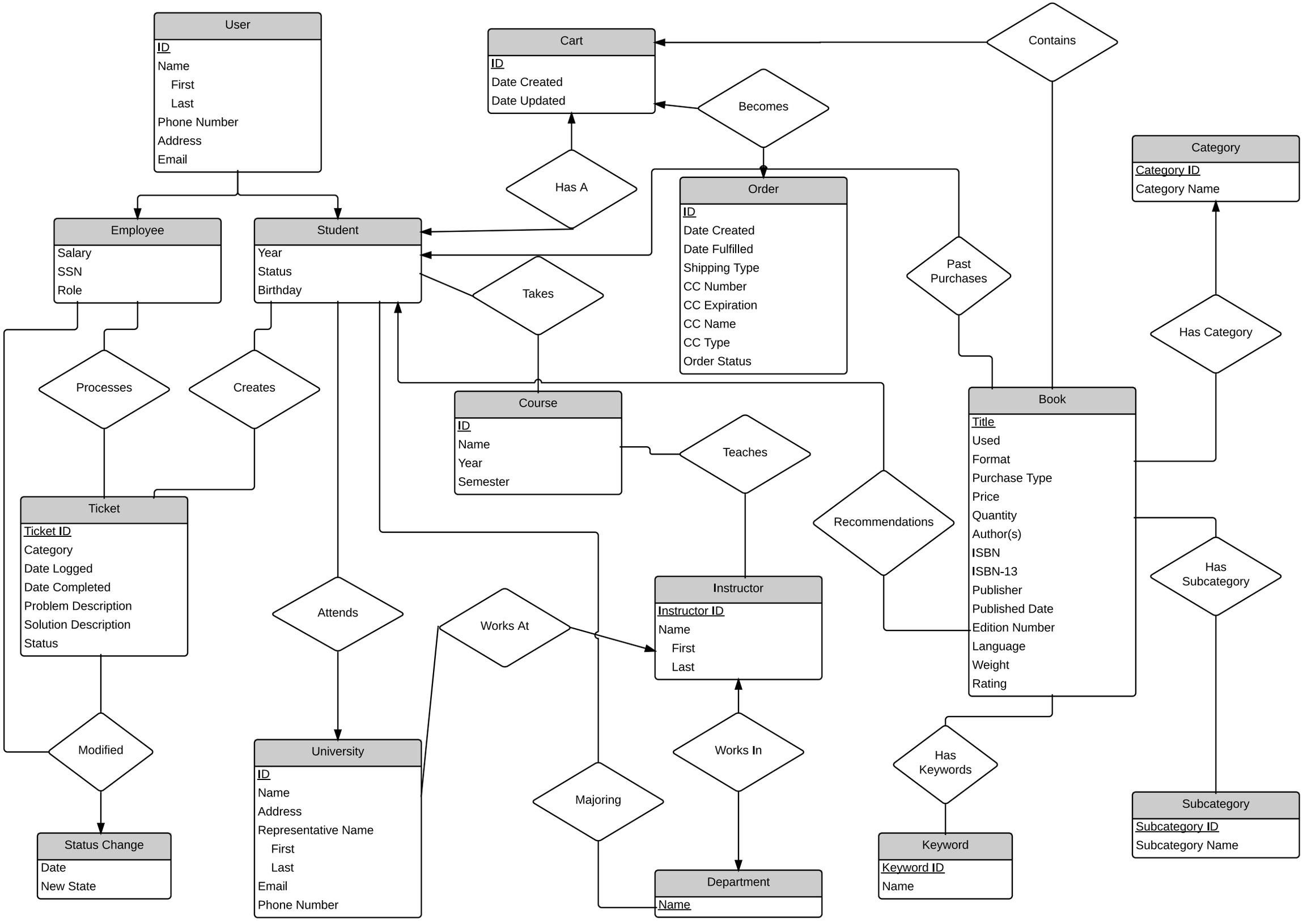
The relationship between employees and tickets was generalized to “processes”. This is mostly for the sake of simplicity. Employees have different responsibilities in regards to tickets, but the distinction was considered to be outside the scope of this design document. This ultimately is an issue of privileges.

There were some values that I chose to represent as values in the entity, while others used a relationship. This decision was based on the cardinality of the value itself. For example, categories of books were linked as a relation, because there is no way of knowing what all of the possible values could be. This is the same for subcategories of books. Conversely, the shipping type of orders was kept as an attribute of the entity. The reason for this is that we have a fixed number of possible options (standard, 1 day, 2 day). Creating another entity to hold these three static values would be a waste of space and possibly a performance issue. Other attributes in other entities also follow this convention, such as language, format and purchase types of books.

In the order entity, credit card name could have been split into a multivalued attribute, but for this particular entity, there seemed to be no reason to do so. In credit card transactions, the whole name is always used.

Book Fetch Database Application

Entity Relationship Diagram



9

Jeremy Neal
CMSC 461 - Sleeman
September 24, 2014

Project Phase 1 Report

Based off of the requirements provided, the database application for Book Fetch Inc. will require a number of features. The requirements have detailed data required in the system, as well as how that data will be used and who is able to interact with it.

At a high level, the workflow for this application will be as follows:

1. Build relation schemas using the information provided. Schemas are required for the following entities: student users, customer support users, administrators, books, universities, departments, instructors, courses, trouble tickets, carts, and orders Schemas which model people (students, customer support, etc.) have a number of elements in common and can be normalized to minimize any redundancy.

   Several schemas can take advantage of user-generated types to represent information more clearly. Examples of this include ticket status, book type, and credit card type. These could be enumerated using integers, but custom types will provide constraint and eliminate confusion and ambiguity.

   Recommendations do not require a schema. They are obtained by running operations on other data.

2. Construct the tables to hold the data. There will be several relationships between entities in the database. For example, tickets will be linked to the administrator they are assigned to via a foreign key.

3. With the database created, an interface for users will need to be created. Access to information must be restricted by user type. As shown in the functional requirements, students must be allowed to create an entry for themselves, create a cart, add items, convert a cart into an order, post reviews, etc.. These must all be strictly limited to student users.

   Customer service users need to respond to trouble tickets and handle order cancellations. As such, particular data, like trouble tickets, must be accessible by them alone, but only when they are new. Once they are assigned, the relevant administrator must be able to access that data.

   Administrators must be able to update inventory, add books and handle assigned trouble tickets. There must additionally be a single super-administrator.

Each user type must have an interface that limits their actions.

4. Using JDBC, functions must be written to handle the user's interactions with the database. These methods must be mapped to the interface to allow the users to perform all of their responsibilities and actions.