A1

CS230 Group 47

**Tawe-Lib Design Document**

**Contents**

1. Introduction
   1. Purpose
   2. Scope
   3. Design Overview
      1. Description of problem
      2. Use Case Diagram
      3. Architecture
2. Descriptions
   1. Hierarchy Descriptions
   2. Collaborations Descriptions
   3. Operations Descriptions
3. Candidate Classes and Responsibilities
   1. Resources
   2. Book
   3. Laptop
   4. DVD
   5. Copies
   6. User
   7. Librarian
   8. Account
4. UML Class Diagrams
   1. Summary
   2. Resources and copies
   3. Users
5. Database Design
6. UI Mock-ups
   1. Summary
   2. User interface
   3. Librarian interface

**Section 1 –** Introduction

* 1. **Purpose**

The purpose of this document is to describe the implementation of the Tawe-Lib Specification given to us in assignment one of CS230. The Tawe-Lib software is a library management system.

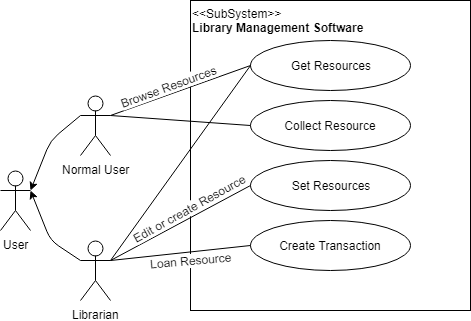
* 1. **Scope**

The Tawe-Lib software is a rudimental library system and is designed to keep track of all library resources, the check-in and check-out of these resources, fines for overdue loans, and multiple copies. All managed through a simple GUI with two distinct types of user.

* 1. **Design Overview**
     1. **Description of problem**

Users of the library must be able to browse all available library resources, get information on a resource and be able to check-out the resource if a copy of it is available. Or be Queued until one is available. Librarians must be able to add or remove resources or copies from the library and facilitate the payment of fines alongside the check-in and check-out of resources by users.

* + 1. **Use Case Diagram**



* + 1. **Architecture**



**Section 2 –** Descriptions

**2.1 Hierarchy Descriptions**

Within the class diagram, shown on 3.2, there are two inheritance hierarchies involved. The first hierarchy is the resources hierarchy, which generalizes the shared attributes of the various resources. Laptops, Books, and DVDs are all a kind of resource. As such, the subclasses specialize the attributes and behaviours of the different types of resources, including: Laptops, Books, and DVDs. As indicated in the class diagram, Resources is an abstract class containing multiple abstract methods. With the use of abstract methods in the superclass, each of the subclasses can have their own methods of implementation. The second hierarchy involved with the class diagram is the accounts hierarchy. Here the attributes of an account are generalized in the abstract class called AccountBaseUser. User and Librarian are both a kind of account. The specialized attributes and behaviours of different types of accounts, users and librarians, are then introduced within subclasses User and Librarian.

**2.2 Collaborations Descriptions**

In our method of implementing the design, we decided to make associations between classes. As a result, there are no aggregation or composition relationships demonstrated within our class diagrams.

Users are capable of viewing and requesting copies via their interface, while they browse through the library’s resources. This search will query the database for all the available copies and present the user with the details of his request.

Librarians are capable of creating and editing resources and copies through their interface. They can also make changes to both Resources and all Copies databases. Those changes will be done by querying the Copies or Resources tables and submitting the required information.

**2.3 Operations Descriptions**

**Section 3** - Candidate Classes and Responsibilities

**3.1**

|  |  |
| --- | --- |
| **Resources** | |
| **Super Class:** None  **Sub Classes:** DVD, Book, Laptop | |
| **Responsibilities**   * Store unique resource ID * Store year * Store title * Store thumbnail image path * Get unique resource ID * Get year * Get title * Get thumbnail Image path * Set thumbnail Image path * Edit Resource | **Collaborations**   * DVD * Book * Laptop * Copies |
| **Rough Description:** Abstract class which holds data which the subclasses have in common, along with abstract operations which can be used by all the subclasses. | |

**3.2**

|  |  |
| --- | --- |
| **Book** | |
| **Super Class:** Resources **Sub Classes:** None | |
| **Responsibilities**   * Create new book instance * Stores author * Stores publisher * Stores isbn * Stores genre * Stores language * Get author * Get publisher * Get ISBN number * Get genre * Get language | **Collaborations**   * Resource * Copies |
| **Rough Description**: A subclass of the superclass Resources, which holds the data and methods relevant to only book and no other Resources subclasses. This class models the physical object that is a real book owned by Tawe-Lib. | |

**3.3**

|  |  |
| --- | --- |
| **Laptop** | |
| **Super Class:** Resources  **Sub Classes:** None | |
| **Responsibilities**   * Create a new Laptop instance * Store operating system * Store manufacturer * Store model * Get operating system * Get manufacturer * Get model | **Collaborations**   * Resources * Copies |
| **Rough Description:** A subclass of the superclass Resources, which holds the data and methods relevant to only laptop and no other Resources subclasses. This class models the physical object of a real laptop owned by Tawe-Lib. | |

**3.4**

|  |  |
| --- | --- |
| **DVD** | |
| **Super Class:** Resources  **Sub Classes:** None | |
| **Responsibilities**   * Create a new DVD instance * Store director * Store runtime * Store language * Store subtitle language * Get Director * Get Runtime * Get Language * Get Subtitle language | **Collaborations**   * Resources * Copies |
| **Rough Description:** A subclass of the superclass Resources, which holds the data and methods relevant to only DVD and no other Resources subclasses. This class models the physical object of a real DVD owned by Tawe-Lib. | |

**3.5**

|  |  |
| --- | --- |
| **Copies** | |
| **Super Class:** None  **Sub Classes:** None | |
| **Responsibilities**   * Stores a unique copy ID * Stores the associated unique resource ID * Stores a loan duration * Create a new instance of Copies * Get unique copy ID * Get unique resource ID * Get issued to * Get issued date * Get issued by * Set issued to * Return copy * Request item * Collect item * Housekeeping | **Collaborations**   * User * Book * Laptop * DVD |
| **Rough Description:** This class is standalone and doesn’t have any super/sub classes.  Instead, it references other objects using a foreign key. A librarian can issue a copy to a User which requires them to specify which resource (i.e. type of book/DVD/laptop) through a foreign key for that object. | |

**3.6**

|  |  |
| --- | --- |
| **AccountBaseUser** | |
| **Super Class:** User, Librarian **Sub Classes:** None | |
| **Responsibilities**   * Stores unique username * Stores first name * Stores last name * Stores telephone number * Stores address * Stores a path to a profile image * Get unique username * Get first name * Get last name * Get telephone number * Get address * Get profile image * Set address * Set telephone number * Set first name * Set last name * Choose profile image * Draw profile image | **Collaborations**   * User * Librarian |
| **Rough Description:** This abstract class is used to store attributes and abstract methods used within both of its subclasses: User and Librarian. | |

**3.7**

|  |  |
| --- | --- |
| **Librarian** | |
| **Super Class:** Account  **Sub Classes:** None | |
| **Responsibilities**   * Stores employment date * Stores a unique staff ID number * Can create an instance of librarian * Get employment date * Get unique staff ID * Get history of transactions * Get list of overdue resources | **Collaborations**   * AccountBaseUser |
| **Rough Description:** This class models the characteristics of a valid librarian’s account within the Tawe-Lib. This is a subclass of AccountBaseUser | |

**3.8**

|  |  |
| --- | --- |
| **User** | |
| **Super Class:** None  **Sub Classes:** Librarian, User | |
| **Responsibilities**   * Stores current account balance * Create a new instance of user * Get balance * Set Balance * Pay fines | **Collaborations**   * AccountBaseUser |
| **Rough Description:** A subclass of AccountBaseUser holding the attributes and operations unique to the User class, this class models the intended normal user of the system. | |

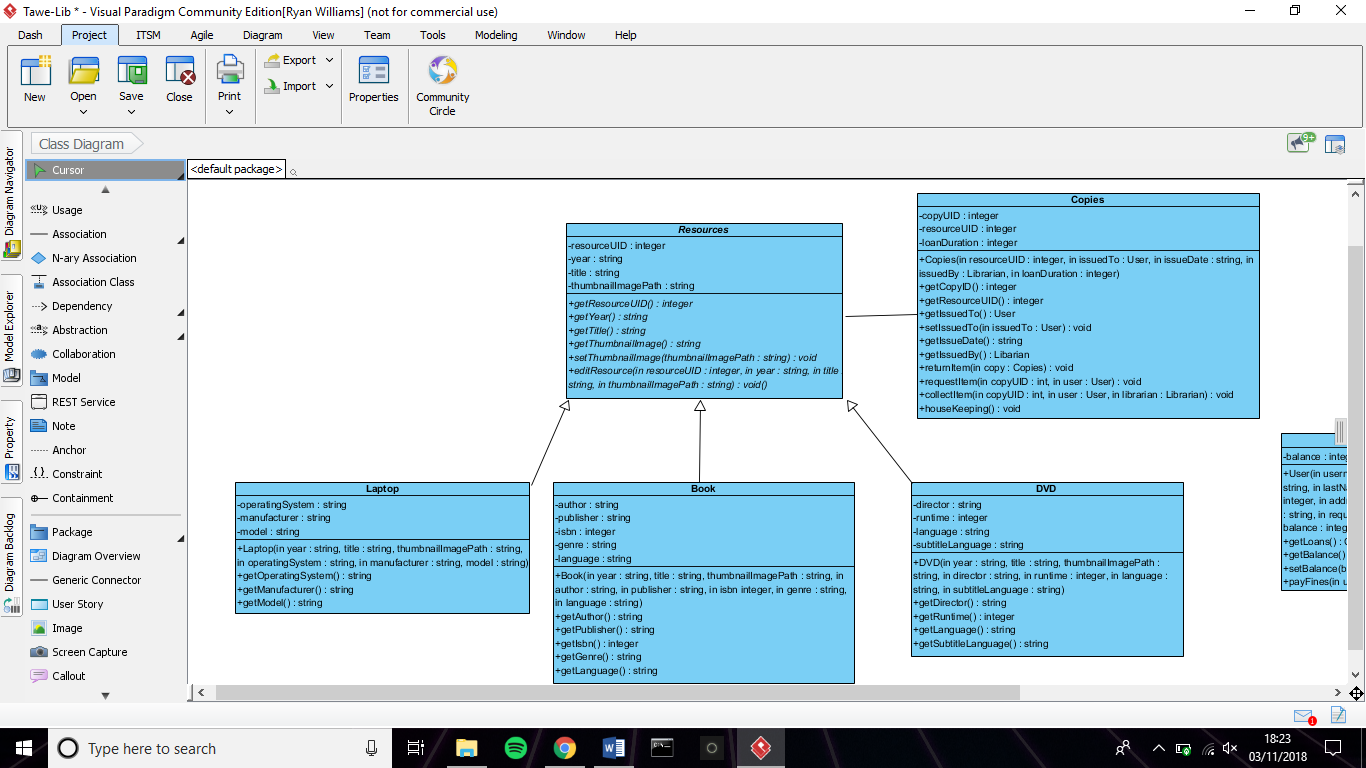
**Section 4** – UML Class Diagrams

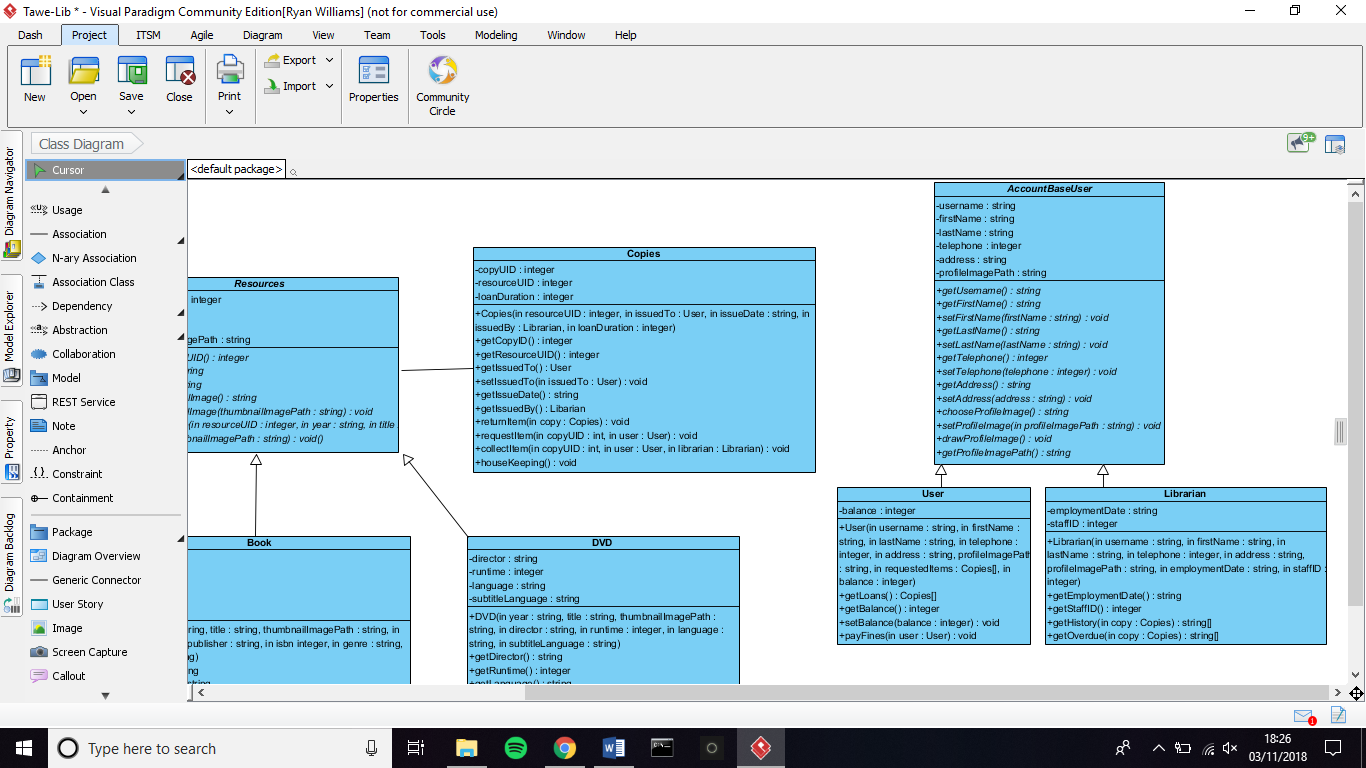
**4.1**

Below is the proposed structural class diagrams for the Tawe-lib software solution. This diagram shows the proposed classes with their respective relations, attributes and methods. This is only the proposed structure and is likely to be altered minimally within the implementation. An example of this would be the sates of the methods and attributes, in respect of their visibility and whether they are static or not. These two properties are currently shown as all are public and non-static, this is likely to change during the implementation stage.

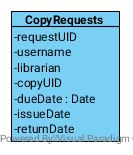
These diagrams have been split into the two main relational groups proposed for Tawe-lib.

**4.2**



**4.3**

**Section 5 –** Database Design



Each class has it’s own table in the database. Each resource type’s table (i.e. Book, Laptop, DVD) is referentially linked to the master Resource table through the resourceUID primary key. The same principle applies for the BaseUser class with the standard User and Librarian. However, things get different when we start to work with Copies. Each copy is stored in the Copy table but all the requests for these are stored in a CopyRequests table not represented as a class. Every request for a copy is stored here and is filtered using the copyUID to list all requests for a copy, or the username to find all historical transactions for a user. This also allows the librarian to search for all overdue items with ease as only one table is being searched. The CopyRequests table functions thanks to the following operations running within the Copies class:

+returnItem:

* Searches the table for all objects with copyUID.
* Last record in the view set with an issueDate is selected.
* The return date is set to either today or a date passed as a parameter.
* If the view set has a “next record” (meaning there is a pending request), set the next record’s dueDate date.
* Return whether a fine is necessary.

+requestItem:

* Search the table for all objects with copyUID.
* If the last record in the view set has a returnDate set, create a new record and set the issueDate as today and dueDate as null.
* Else if the last record in the view set has an issueDate set and the requesting user is not the user in the last record, update their dueDate to the minimum length specified in duration.
* Create a new record for the current request.

+collectItem:

* Search the table for all objects with copyUID.
* If the dueDate has not passed and the requesting user is the correct one, set the issueDate as today.
* If the view set has a “next record” set the dueDate to the minimum duration.

+housekeeping: (This is run nightly to remove users that didn’t collect the copy.)

* Search the table for all objects with copyUID.
* If the dueDate has expired and there is no issueDate, delete the record.
* Select the next record on from the deleted one and set the dueDate to the collection time.

As can be seen above, there are a couple issues with this database schema. One such issue is that all the data is stored in one long table when it could be split based as one table per copy. For ease of development, we’ll stick to this design. We have to use a lot of SQL logic to get the data we want, however in the real world this won’t cause a significant slowdown of the application. The biggest issue however is that we require a housekeeping function. When an item is returned or requested it automatically sets the collection dates and due dates, however if no one collects the item the system can’t automatically expire it as no function is being run. In an ideal world, we would have this running as an asynchronous operation constantly looking for expired collections and then remove that record. In this case, we run a task at midnight to clear the uncollected requests and set the next request ready to collect. This could be done through a Cron Job on the host computer.

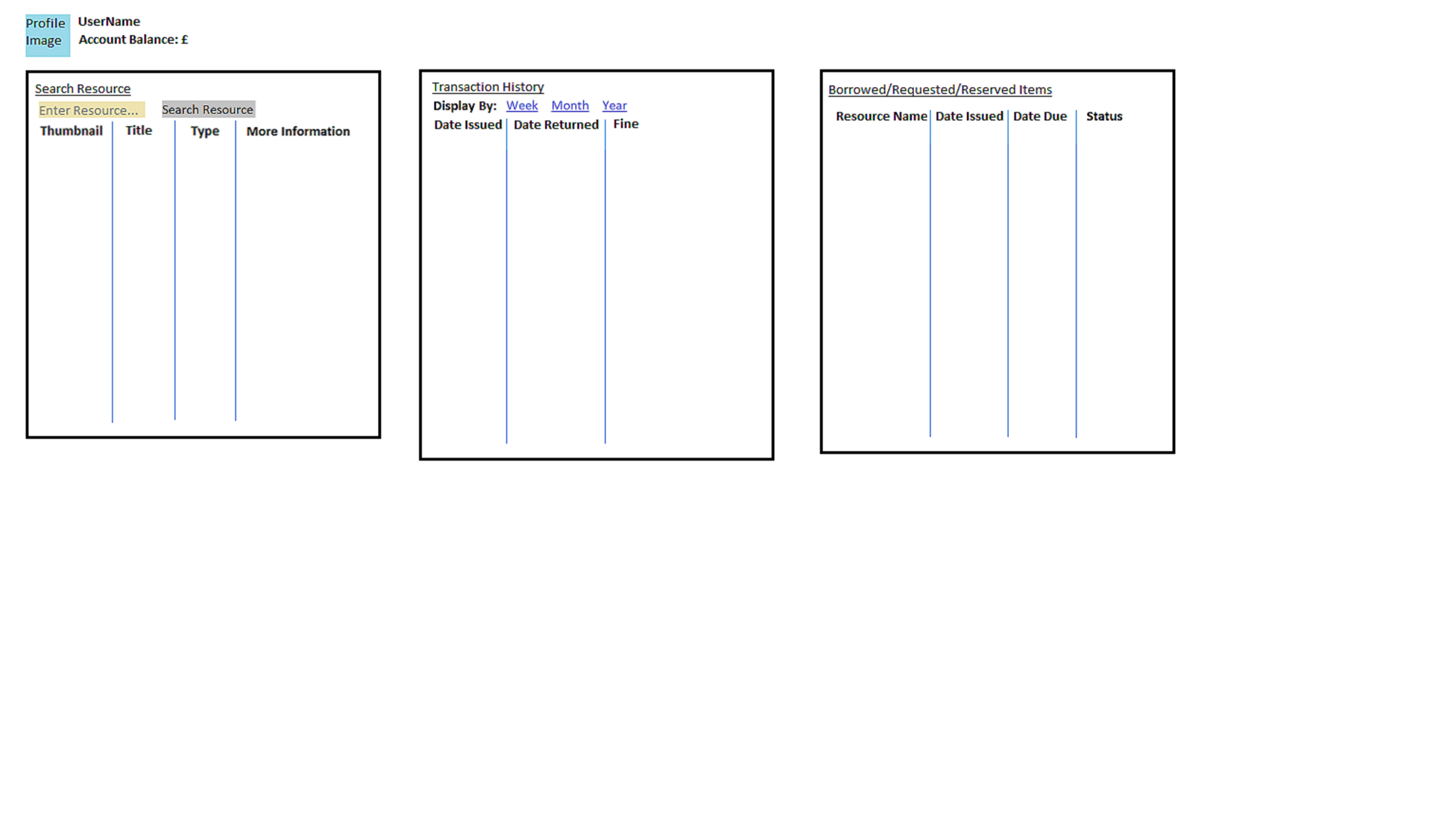
The software we will use to host the database will be SQLite3. We chose this over other options such as MariaDB, MongoDB, mySQL and others simply because of how lightweight it is as well as the integrations it offers for many IDE’s and languages. We can install the relevant library and spin it up within the program itself without requiring a separate service to host the database. We recognise that SQLite does have a disadvantage in that it’s slower compared to the alternatives and can’t handle as much data. Although in this case that doesn’t matter as we won’t be dealing with hundreds-of-thousands of records. It does have the issue whereby if the program crashes, so does the database – although in all the tests I’ve seen the data has been safe.

**Section 6** – UI Mock-ups

**6.1** Summary

Some rudimentary GUI mock-ups to show a basic layout of how the dashboards should work for each user. These two interfaces minus a few secondary interfaces are the only way a user should interface with the system. Users can browse all Resources, their transaction history and statistics. Librarians can browse, edit or create resources, and also administer resource collections, loans or overdue copies.

**6.2** User Interface



**6.3** Librarian Interface

