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. Candidate Classes and Responsibilities
   1. Resources
   2. Book
   3. Laptop
   4. DVD
   5. Copies
   6. User
   7. Librarian
   8. Account
3. UML Class Diagrams
   1. Summary
   2. Resources and copies
   3. Users
4. Database Design
5. UI Mock-ups

**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.

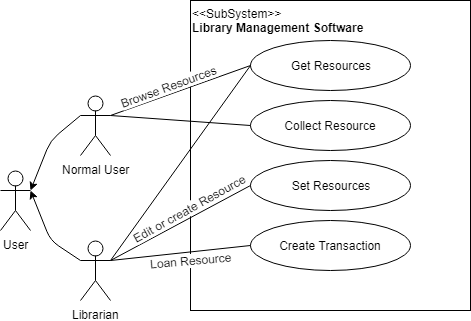
* 1. **Scope**

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** - Candidate Classes and Responsibilities

**2.1**

|  |  |
| --- | --- |
| **Resources** | |
| **Author:** Ryan Williams  **Super Class:** None  **Sub Classes:** DVD, Book, Laptop | |
| **Responsibilities**   * Provide a basic framework for the creation of other objects. * Unique resource ID. * Thumbnail image | **Collaborations**   * DVD * Book * Laptop * Copies |
| **Rough Description:** Abstract class. Contains the attributes held by all resources. | |

**2.2**

|  |  |
| --- | --- |
| **Book** | |
| **Author**: Ben Farrington  **Super Class:** Resources **Sub Classes:** None | |
| **Responsibilities**   * Get author string. * Get publisher string. * Get ISBN number. * Get genre string. * Get language string. | **Collaborations**   * Resource * Copies |
| **Rough Description**: A class to hold data on specifically the resource type of Book. A Book is a kind of resource. | |

**2.3**

|  |  |
| --- | --- |
| **Laptop** | |
| **Author:** Cristi Neacsu  **Super Class:** Resources  **Sub Classes:** None | |
| **Responsibilities**   * Get ID string. * Get Title/Name string. * Get Year number. * Get Manufacturer string. * Get Model string. * Get OS string. | **Collaborations**   * Resources * Copies |
| **Rough Description:** A child class of Resources, to store information about laptops available to rent in library. A Laptop is a kind of resource. | |

**2.4**

|  |  |
| --- | --- |
| **DVD** | |
| **Author:** Kieran Hughes  **Super Class:** Resources  **Sub Classes:** None | |
| **Responsibilities**   * Get Director. * Get Runtime. * Get Language. * Get Subtitle Language. | **Collaborations**   * Resources * Copies |
| **Rough Description:** Child class of Resources, stores attributes of DVDs available to rent from the library. A DVD is a kind of resource. | |

**2.5**

|  |  |
| --- | --- |
| **Copies** | |
| **Author:** Alex Moras  **Super Class:** None  **Sub Classes:** None | |
| **Responsibilities**   * Contains:   + Unique copy-ID (PK)   + Resource ID (FK)   + Issued to (User FK)   + Issue date   + Issued by (Librarian FK)   + Loan duration   + Expected return date   + Actual return date * Return date isn’t set until someone else requests the item and one isn’t available. * Each copy is a unique instance referencing a resource FK. * Each copy references the user (FK) borrowing it. | **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. On issue, return date isn’t set. If someone requests an unavailable item, the oldest “on loan” item is marked with a due date that is no-shorter than the loan-duration. | |

**2.6**

|  |  |
| --- | --- |
| **Users** | |
| **Author:** Christian Onishile  **Super Class:** Account  **Sub Classes:** None | |
| **Responsibilities**   * Username to identify instances of account * First name of instance of account * Last name of instance of account * Phone number on instance of account * Address of instance of account * Profile image of instance of account | **Collaborations**   * Accounts * Copy |
| **Rough Description:** This class models the characteristics of a valid user’s account within the Tawe-Lib. A user is a kind of account. | |

**2.7**

|  |  |
| --- | --- |
| **Librarian** | |
| **Author:** Christian Onishile  **Super Class:** Account  **Sub Classes:** None | |
| **Responsibilities**   * Username to identify instances of account * First name of instance of account * Last name of instance of account * Phone number on instance of account * Address of instance of account * Profile image of instance of account * Employment date of instance of account * Staff number of instance of account | **Collaborations**   * Accounts * Copy |
| **Rough Description:** This class models the characteristics of a valid librarian’s account within the Tawe-Lib. A librarian is a kind of account. | |

**2.8**

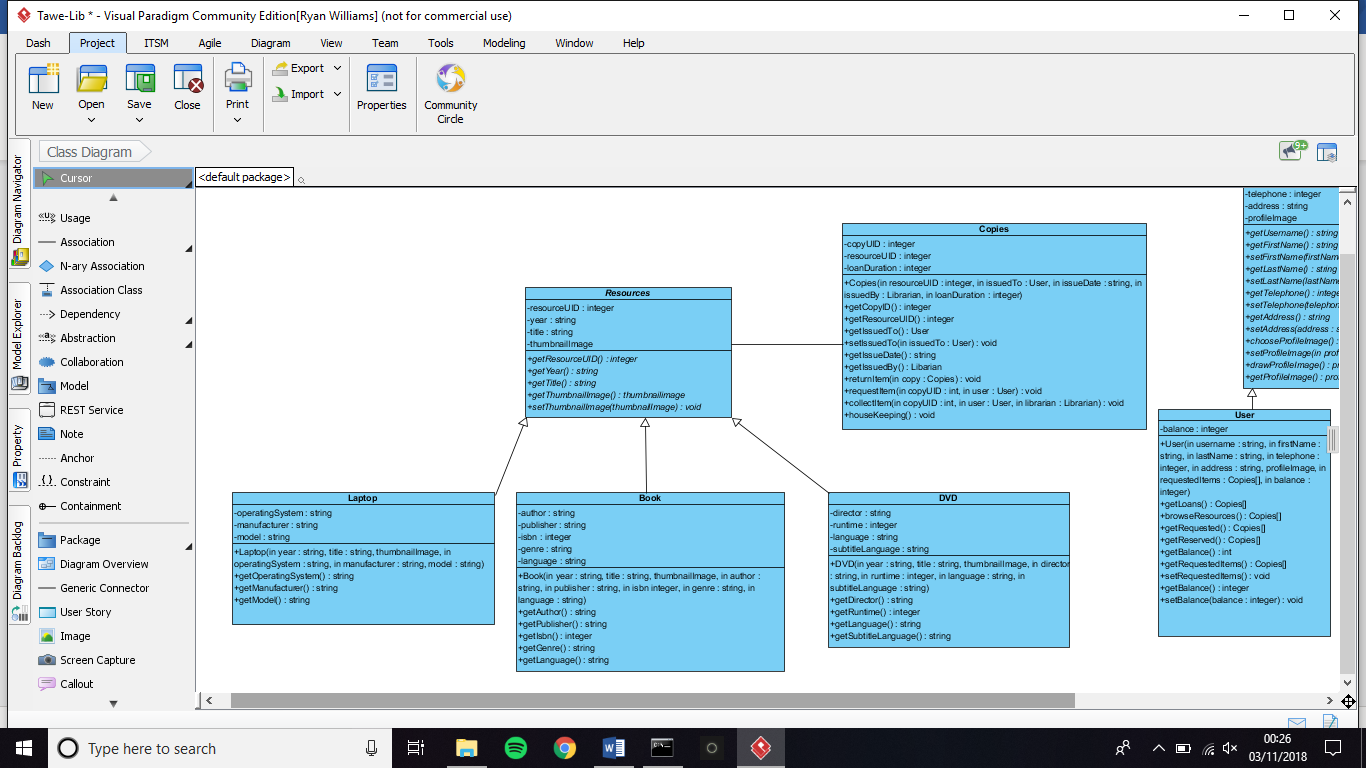
|  |  |
| --- | --- |
| **Account** | |
| **Author:** Christian Onishile  **Super Class:** None  **Sub Classes:** Librarian, User | |
| **Responsibilities**   * Username to identify instances of account * First name of instance of account * Last name of instance of account * Phone number on instance of account * Address of instance of account * Profile image of instance of account | **Collaborations** |
| **Rough Description:** An abstract class. With two sub-types of User, a standard User and a Librarian. | |

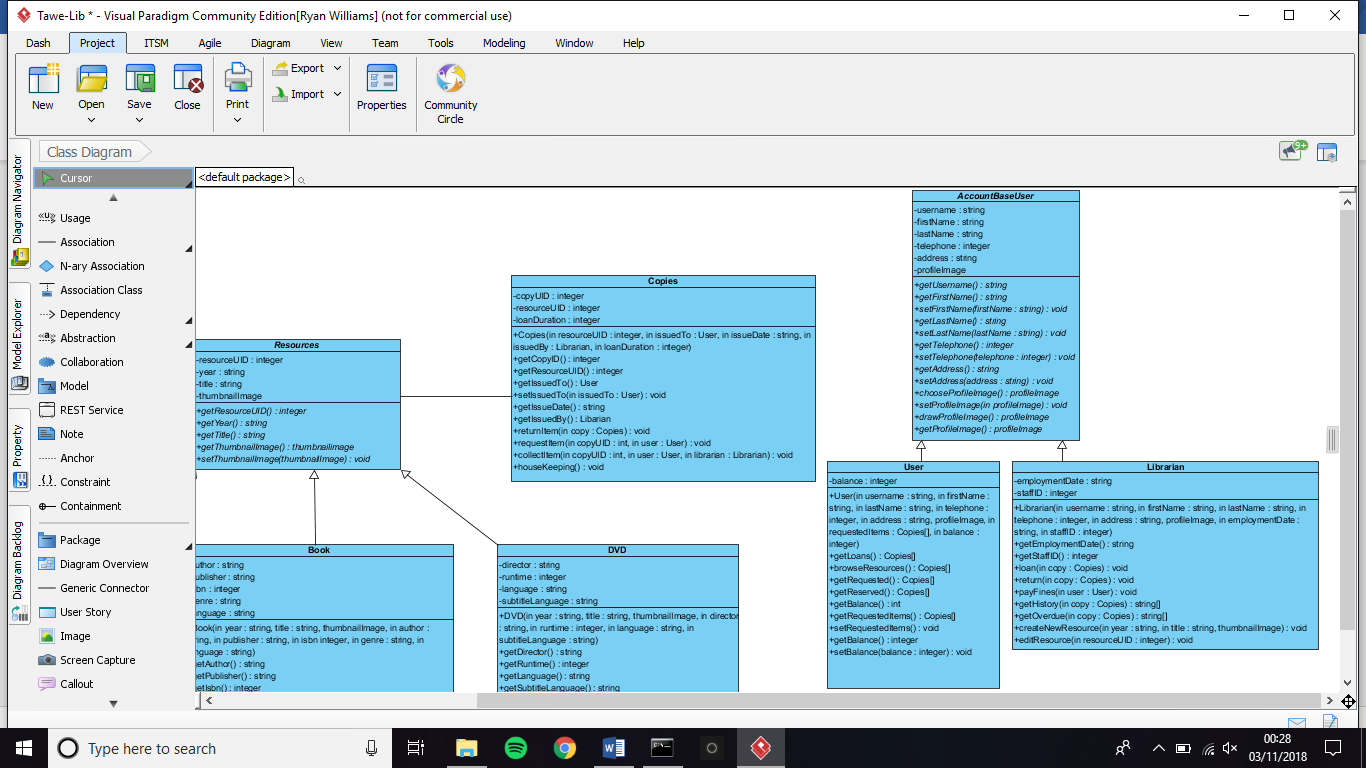
**Section 3** – UML Class Diagrams

**3.1**

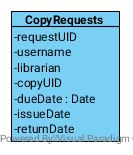
Below is the proposed structural class diagrams for the Tawe-lib software solution. This diagram shows the proposed classes with their respective relations, attributes and methods. This is only the proposed structure and is likely to be altered minimally within the implementation. An example of this 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.

**3.2**

**3.3**

**Section 4 – Database Design**

**4**

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.