Library Manager <> Design and Execution Notes CIS-35B

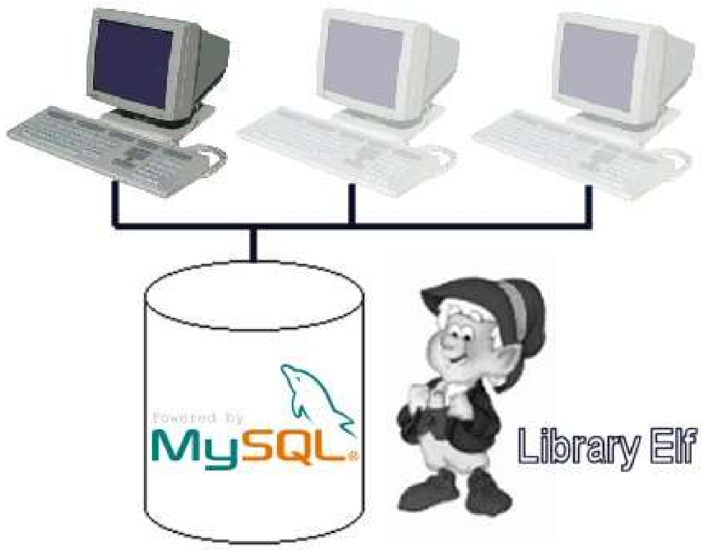
T. Fox, T. Nguyen, T. Pham

Overview

The Library Manager (LM) System shall consist of one or more data terminals accessing a common database.

A terminal for LM shall be an application written in Java executing on a personal computer. At this time the preferred PC platform is any desktop or laptop running Windows2000 or XP, with at least 500 Megs of RAM. The PC shall have jre 1.4.x installed.

The preferred database is presently MySQL, version 4.0 or higher. It may be installed in one of the PCs that hosts the LM terminal application, or in a separate dedicated host to which LM data terminals can connect via TCP/IP.



The “Library Elf” represents whatever functionality LM may need that is not defined for or implemented in the terminal applications. It may be a daemon program or ‘phantom’ terminal running when all the other terminals are shut down, doing database cleanup. ***TBD***.

The overall functionality of the LM data terminal is to conduct transactions for users with the database. Internally, the functionality divides into three categories:

- The GUI: The classes that present to a user via graphic screens the kinds of transactions that are available. Examples: the login screen, the catalog search screen, etc.
- The transactions: The classes that model the way a user and the database interact to perform a transaction. Examples: check out a book, check in a book, create a new user account, etc.
- The data modelers: The classes that represent records in the tables of the database.

T. Fox, T. Nguyen, T. Pham

The database tables

users

Users include patrons, librarians, and administrators. The first character of the `user_id` field indicates whether the user is a patron ('P'), librarian ('L'), or administrator ('A')[1]. The record structure of the ***users*** table is:

```
user_id CHAR(10) NOT NULL
password VARCHAR(12) NOT NULL UNIQUE
date_added DATE NOT NULL
date_retired DATE
contact_id INTEGER
PRIMARY KEY(user_id)
```

The `contact_id` field identifies a record in the **contactinfo** table (below).

branches

Branches are separate locations where the public can have access to library resources. Books are shelved at branches. Each branch has its own record in the database so that the library system can have a way of identifying the location of books that are not checked out. The record structure of the ***branches*** table is:

```
branch_id CHAR(10) NOT NULL
policy_id CHAR(3) NOT NULL
date_added DATE NOT NULL
date_retired DATE
contact_id INTEGER
PRIMARY KEY (branch_id)
```

contactinfo

'Contact info' are the data items that name and locate a branch, or the residence of a user. The record structure of the ***contactinfo*** table is:

```
contact_id INTEGER NOT NULL
last_name VARCHAR(15)
first_name VARCHAR(15)
m_i CHAR(1)
street_address VARCHAR(20)
city VARCHAR(15)
state CHAR(2)
zip VARCHAR(10)
phone VARCHAR(15)
email VARCHAR(20)
PRIMARY KEY (contact_id)
```

1 Letting the categories of user IDs be recognizable by eye was a choice made while prototyping the database design using a CSV (Comma Separated Values) database, before we got MySQL running properly. CSV tables are all ASCII text, so having the various kinds of key fields be easily classified by the first character of the field was intended to make it easier to verify correct key composition when manually editing the CSV tables.

Library Manager <> Design and Execution Notes CIS-35B

T. Fox, T. Nguyen, T. Pham

catalog

The ***catalog*** table is a list of book descriptions. Its record structure is:

```
catalog_id CHAR(10) NOT NULL
title VARCHAR(15) NOT NULL
author VARCHAR(15) NOT NULL
publisher VARCHAR(15) NOT NULL
pub_year VARCHAR(4)
isbn VARCHAR(10)
other_key VARCHAR(8)
keywords VARCHAR(10)
PRIMARY KEY (catalog_id)
```

inventory

An ***inventory*** item is a specific copy of a book that is described in the ***catalog*** table. The record structure of the ***inventory*** table is:

```
item_id CHAR(10) NOT NULL
catalog_id CHAR(10) NOT NULL
cost VARCHAR(6)
date_added DATE NOT NULL
date_removed DATE
why_removed VARCHAR(10)
PRIMARY KEY (item_id)
```

invloc

The ***invloc*** (**inv**entory **loc**ation) table is a list of the current locations of the library's inventory items (books). For any inventory item there shall be no more than one active (`is_current` == true) ***invloc*** record.

The record structure for ***invloc*** is:

```
move_id VARCHAR(10) NOT NULL
item_id CHAR(10) NOT NULL
id_from CHAR(10) NOT NULL
id_to CHAR(10) NOT NULL
trans_date DATE NOT NULL
due_date DATE
operator_id CHAR(10) NOT NULL
memo_id CHAR(10)
is_current CHAR(1) NOT NULL
PRIMARY KEY (move_id)
```

The purpose of the `operator_id` field is to record the `user_id` of the librarian or administrator logged in at the terminal where the transaction takes place. The `memo_id` field may hold a foreign key that identifies a memo. The intended purpose of the memo is to record notes of wear or damage noticed at check-in.

Only items in active records may be eligible for transactions (below). The fields `id_from` and `id_to` can be *user_id*s, *branch_id*s, or *hold_id*s according to the following rules:

from	**to**	**transaction classification**
branch	branch	OK (is 'inter-branch transfer') **NOTE*:*** When a book is added to inventory, its first ***invloc*** record will have the `branch_id` where the book enters the library system recorded as `id_from` and `id_to`.
branch	user	OK (is 'check-out')
user	branch	OK (is 'check-in')
user	user	*irrelevant*. This is not a transaction available at a library terminal, and so would not be recorded in the database. As far as the library system is concerned, a book turned in will be recorded in the database as checked in by the user who checked it out.
branch	hold	*place book on hold.* A hold request acts like a proxy checkout for the user who placed the hold request.
hold	branch	*release book from hold.* A book on hold shall be released from hold when the hold request expires, or when the user who placed the hold request comes to the branch where the book is held before the hold expires and asks a librarian to release that book.
hold	user	NEVER. A user picking up a book from hold gets it from the *branch*.
user	hold	NEVER
hold	hold	NEVER

invhist

The ***invhist*** (inventory history) table is the archive of all the *previous* locations of the library's inventory items. The record structure is identical to that of ***invloc***. Records are added to ***invhist*** by copying obsolete records (`is_current` == false) from ***invloc***. Obsolete records may be purged from ***invloc*** after verifying that they have been copied to ***invhist***.

onhold

The ***onhold*** table contains records of all unexpired hold requests. The record structure for the ***onhold*** table is:

```
hold_id CHAR(10) NOT NULL
catalog_id CHAR(10) NOT NULL
item_id CHAR(10)
user_id CHAR(10) NOT NULL
branch_id CHAR(10)
date_requested     DATE NOT NULL
expires_on DATE NOT NULL
is_current CHAR(1) NOT NULL
PRIMARY KEY (hold_id)
```

T. Fox, T. Nguyen, T. Pham

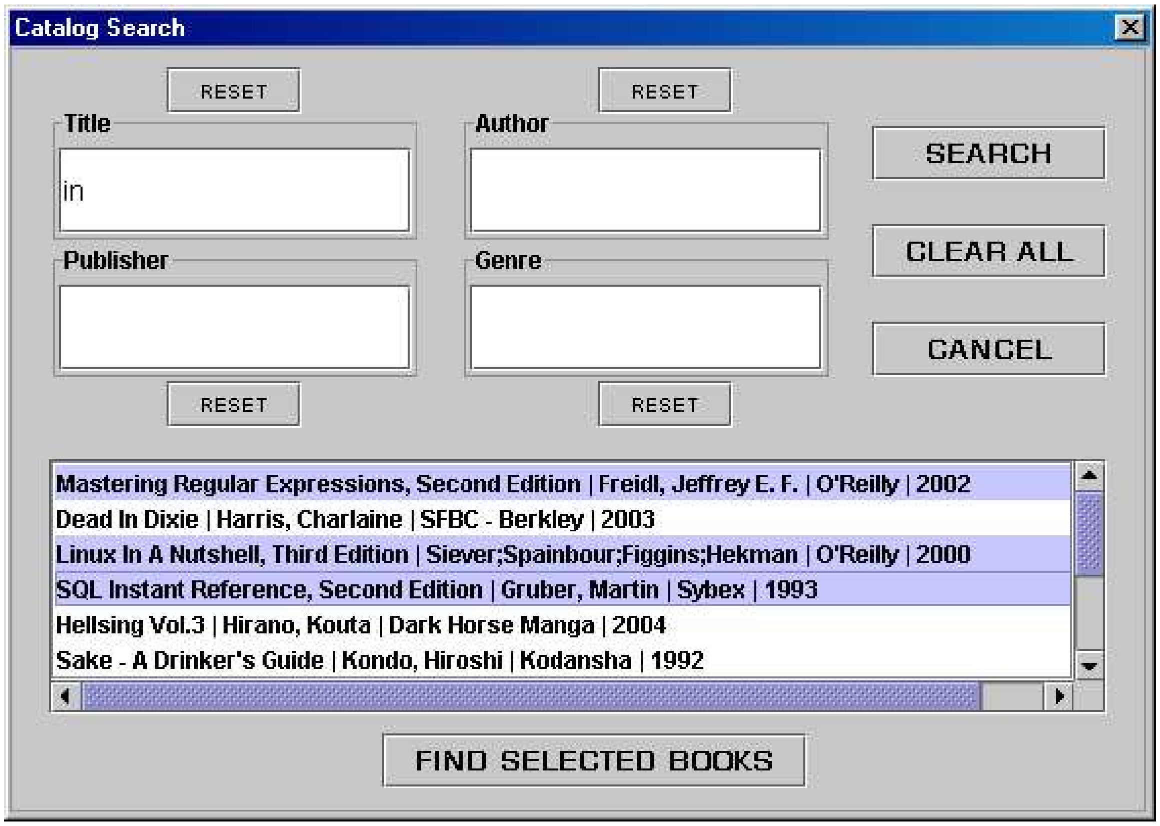
The GUI classes

These classes generate the screens that user sees, direct input from the terminal user to appropriate transaction objects, and display the transaction results.

welcome

catalog search

This dialog lets the user look for books based on Title, Author, Publisher, or Genre, or any combination of the four. See FindABook.java. This dialog uses two transaction support classes, CatalogSearch and InventoryStatus. When the user highlights items of interest and presses [FIND SELECTED BOOKS], the list will rewrite itself to show books that are available for check-out in the library system, and the library branches where they're shelved.



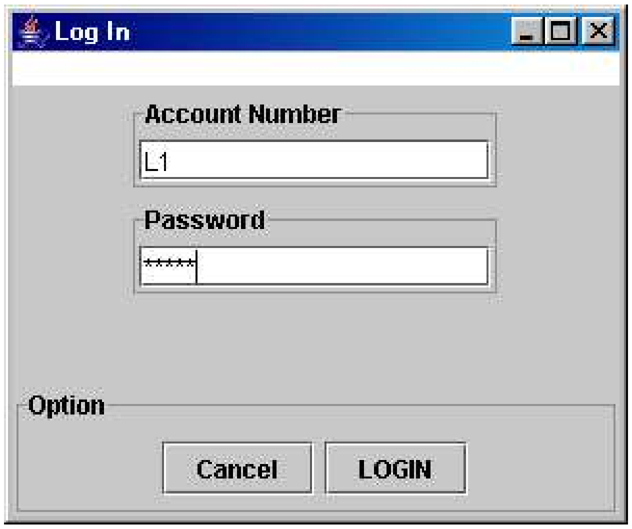
Title: in

Mastering Regular Expressions, Second Edition | Freidl, Jeffrey E. F. | O'Reilly | 2002
Dead In Dixie | Harris, Charlaine | SFBC - Berkley | 2003
Linux In A Nutshell, Third Edition | Siever;Spainbour;Figgins;Hekman | O'Reilly | 2000
SQL Instant Reference, Second Edition | Gruber, Martin | Sybex | 1993
Hellsing Vol.3 | Hirano, Kouta | Dark Horse Manga | 2004
Sake - A Drinker's Guide | Kondo, Hiroshi | Kodansha | 1992

T. Fox, T. Nguyen, T. Pham

user login

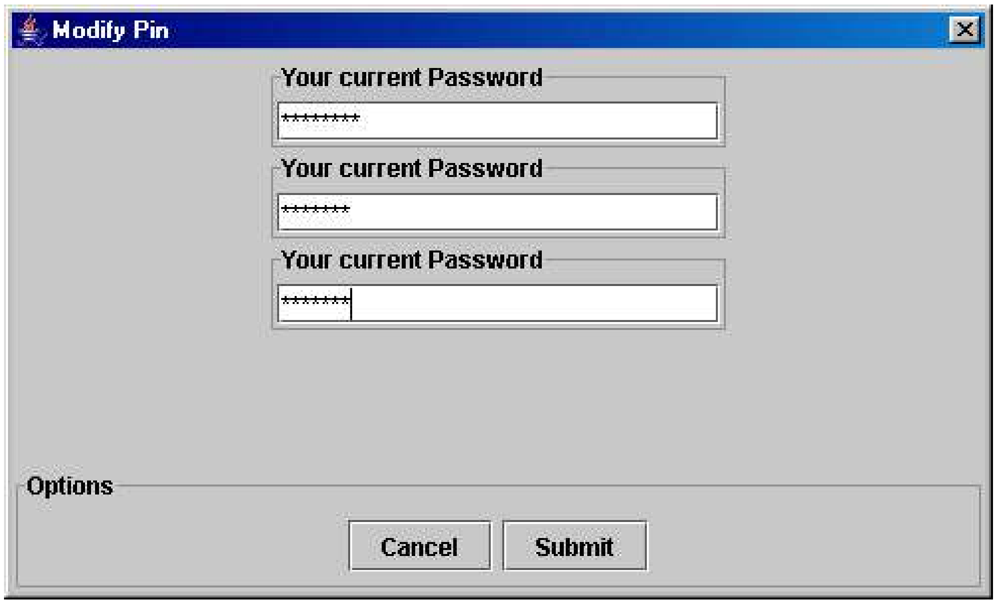
The user enters the Account Number (UserID), the password, then presses [LOGIN]. The UserID object will accept the abbreviated form (one letter and the significant digits) of Account Number.



Account Number: L1
Password: *****

patron services menu

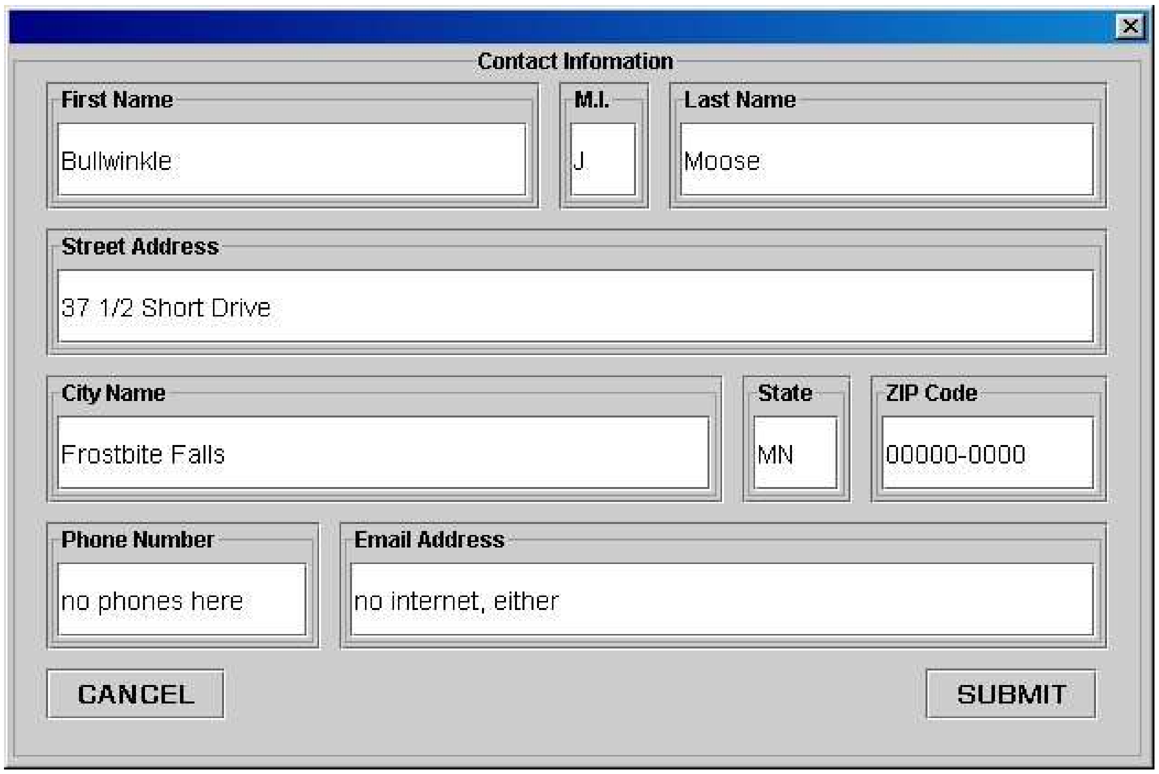
user: change password (PIN)



Your current Password: ********
Your current Password: *******
Your current Password: *******

user: view/update contact info

Depending on the parameters passed to it, this dialog can create, edit or show (as read-only) the data in a record from the **contactinfo** table. See ContactInfoEditor.java.



First Name	Bullwinkle
M.I.	J
Last Name	Moose
Street Address	37 1/2 Short Drive
City Name	Frostbite Falls
State	MN
ZIP Code	00000-0000
Phone Number	no phones here
Email Address	no internet, either

user: view checkouts/request renewal

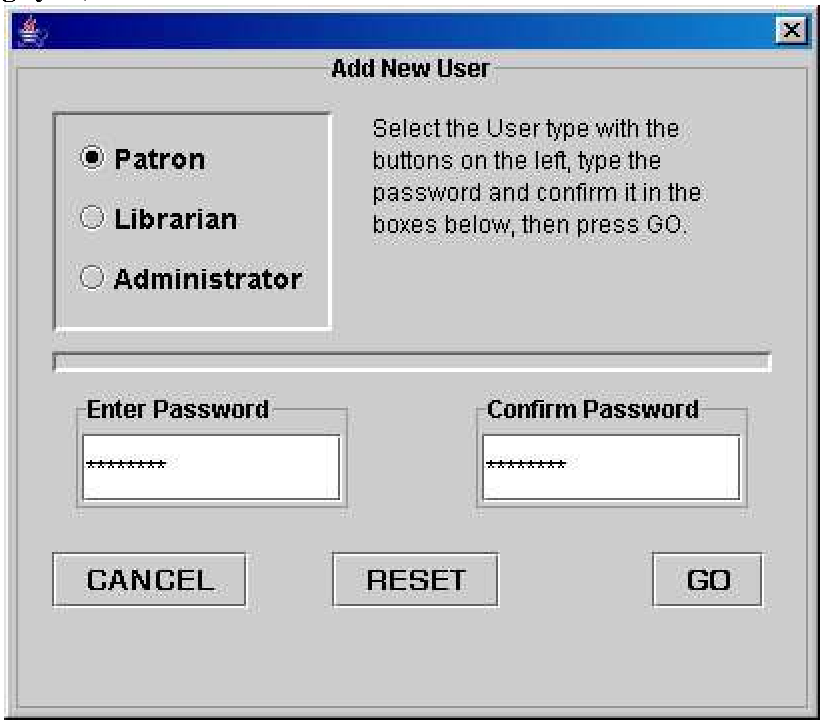
user: view fines assessed

user: request item hold

librarian services menu

librarian: patron account / contact info maintenance (add new / update / remove)

The dialog is in two parts. The first provides a way to select the kind of user account to be created, and two password fields for entering and confirming the user password.



Select the User type with the buttons on the left, type the password and confirm it in the boxes below, then press GO.

Enter Password: ********

Confirm Password: ********

The second part of user account maintenance, the dialog for creating / editing contact information, uses the same dialog as shown in ***user: view/update contact info***. A Patron can only invoke the dialog on his/her own information. A Librarian can invoke it on any Patron, and on the Librarian's own info.

librarian: check out

librarian: check in

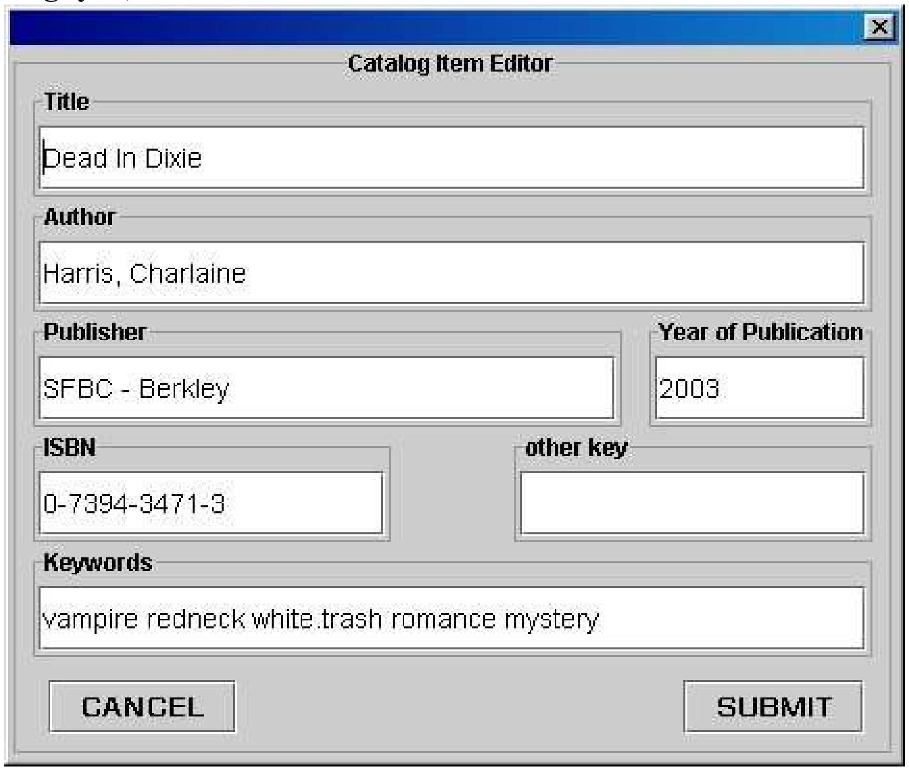
librarian: view / update / cancel hold requests

librarian: collect fees

librarian: catalog maintenance (add new / update)

Depending on the parameters passed to it, this dialog can create, edit or show (as read-only) the data in a record from the **catalog** table. The example below was invoked on an existing catalog item. See CatalogItemEditor.java.

T. Fox, T. Nguyen, T. Pham

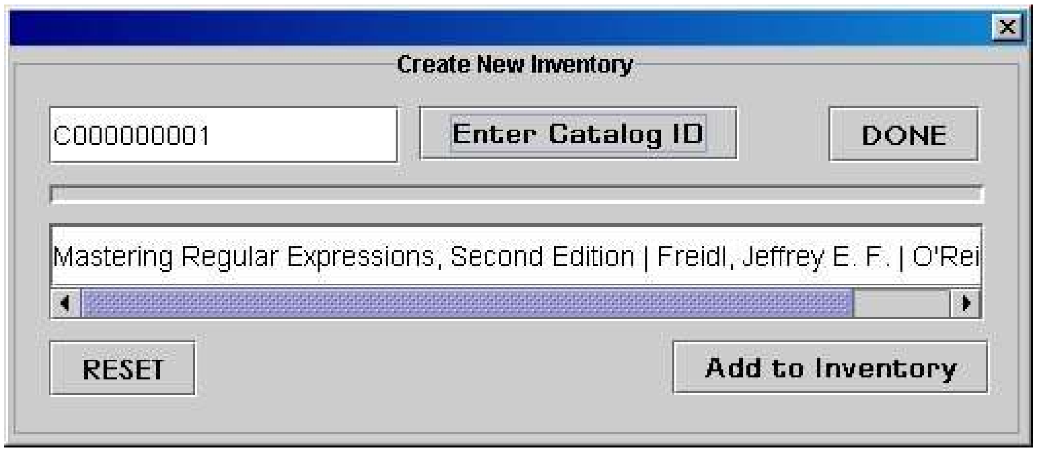


Title	Dead In Dixie
Author	Harris, Charlaine
Publisher	SFBC - Berkley
Year of Publication	2003
ISBN	0-7394-3471-3
other key	
Keywords	vampire redneck white.trash romance mystery

librarian: inventory maintenance (add new / update / remove)

This dialog lets a Librarian add a book to inventory. Usage: Enter a catalog ID (short form is OK) and press [Enter Catalog ID]. If that ID exists in the catalog, its description will be shown in the info window. If the description matches the book to be added, press [Add to Inventory], else press [RESET]. When all books have been added, press [DONE] to exit the dialog.

In the example below, the user entered “c1” and pressed [Enter Catalog ID]. The dialog found the record, rewrote its ID in expanded form, and displayed the catalog description. The information window is in a jScrollPane to allow long information strings to be displayed at a comfortably large font size.



C000000001

Mastering Regular Expressions, Second Edition | Freidl, Jeffrey E. F. | O'Rei

administrator services menu

administrator: librarian account maintenance (add / update / remove)

administrator: administrator account maintenance (add / update / remove)

administrator: branches maintenance (add / update / remove)

The transaction classes

If LM were to actually be deployed for a municipal library system, these transaction classes would be implemented as session EJBs. For our CIS-35B lab system, though, these are simply classes that exist in the data terminal application.

For a business version of LM, all transaction classes would include a method to log execution of a transaction to a journal file.

Create a new user, issue library card

To establish a new user account,

1. instantiate a User(userType) object, passing as userType 'A' for Administrator, 'L' for Librarian, or 'P' for Patron.
2. Use the setPassword() method to set the value of the User instance's password property This method shall not change the database.
3. Call the update() method to add the new User to the database. If update() succeeds, it shall set the values of the contactID and dateAdded properties.

Change user password

To access an existing user account,

instantiate a User(userID) object, passing as userID the user's account ID. The structure of a userID string shall be a single capital letter ['A', 'L','P'] followed by decimal digits. If that ID exists in the User database, the User instance's properties shall be set to the values fetched from the database. If such an ID is not found in the User database, the constructor shall fail.

Remove a user

Create a new branch and new empty contact info

Create a catalog item / Edit a catalog item

Does not exist as a distinct class; incorporated in the CatalogItemEditor dialog.

Library Manager <> Design and Execution Notes CIS-35B

T. Fox, T. Nguyen, T. Pham

Search the catalog

See CatalogSearch.java, InventoryStatus.java.

(Retire an inventory item – not for this version of LM)

Check out

Check in

Show checked out items for user

Show accrued fines for user

Place a hold request

A user places a hold request by `catalog_id`. The transaction that creates the onhold record fills in the `hold_id`, `user_id`, `catalog_id`, set `date_requested` to the current date, sets the `expiration_date` to current date + 30 days, and sets `is_current` to T (or '1' ?).

Scan to detect inactive records (invloc, onhold)

Daily, nominally at midnight, a process shall scan through the ***onhold*** table to find records where (`expires_on` < today). It shall set those records' `is_current` field to 'F' (or '0'?). When all records have been scanned, it shall copy the active records to a temp table, drop the old ***onhold*** table, and (re)create the ***onhold*** table from the temp table.

Similarly, the ***invloc*** table shall be scanned. The scan process shall ensure that ***invhist*** has a copy of every inactive record in ***invloc***. The process shall then copy the active records from ***invloc*** to a temp table, drop the old ***invloc*** table, and (re)create the ***invloc*** table from the temp table.

Scan to fulfill hold requests

This transaction shall take place periodically, on a schedule defined by library policy. While this process runs, the system may need to lock some database tables. This may disable some data terminal functionality while the scan process runs.

When scheduled, this process shall scan through the ***onhold*** table, sorting by `catalog_id` and `date_requested`. For each `catalog_id`, it shall search the ***invloc*** table for records that (1) are current, (2) have `in_to` == a `branch_id`, (3) `catalog_id` matches the ***onhold*** record. The process shall note the `item_ids` from the ***invloc*** records and use them to fill hold requests, oldest request first. For each `item_id` that satisfies a hold request, the process shall set the (previously empty) `item_id` of the ***onhold*** record, set the `expires_on` date to one week from today, and set the `branch_id` to the branch where the item was found. The process shall then set the existing ***invloc*** record of `item_id` to (`is_current` == false), and create a new ***invloc*** record to check out `item_id` from `branch_id` to `hold_id`. The process shall then send notification to the user who placed the request.

T. Fox, T. Nguyen, T. Pham

The data modeler classes

Each of these classes represents a record in a database table.

If LM were to actually be deployed for a municipal library system, these data modeler classes would be implemented as entity EJBs.

Because data modeler classes depend on connection to the database, these classes shall throw SQLException.

Because data modeler classes include methods that may fail when trying to match input args with field data, these classes shall throw IllegalArgumentException.

(But because all we're really concerned about is whether or not the constructors and class methods succeed, many of them just throw the generic Exception.)

Each data modeler class shall have

- a private property that shall indicate whether the record is being created for the first time, or is from an existing record in the table. Class methods may use this property to determine whether or not to modify the value of a field. (Example: Once the record has been written for the first time, it is never permissible to change the value of the record's primary key.)
- private properties for all the record's fields
- a constructor for provisionally creating a new record. This constructor shall set the value of the object's key field property, and of other properties which shall be passed to the constructor as parameters. The constructor shall not write to the table.
- a method to commit a record update that shall succeed only if data are defined for the record's NOT NULL fields.
- a constructor for retrieving a record that matches a primary key. This constructor shall fill the object's properties with the values read from the record whose primary key matches the constructor's input parameter.
- public getters for all an object's field properties. These getters shall not read the database. The only method that shall read the database is the constructor that finds a record for a matching primary key.
- public setters for all the field properties that are permissible to set. (The properties that may only be set when creating the record shall be passed as arguments to the constructor for new records.)

Library Manager <> Design and Execution Notes **CIS-35B**

T. Fox, T. Nguyen, T. Pham

User

This class models a record of the ***users*** table. See the Javadoc for *User*.

Branch

This class models a record of the ***branches*** table. See the Javadoc for *Branch*.

ContactInfo

This class models a record of the ***contactinfo*** table. See the Javadoc for *ContactInfo*.

CatalogItem

This class models a record of the ***catalog*** table. See the Javadoc for *CatalogItem*.

InventoryItem

This class models a record of the ***inventory*** table. See the Javadoc for *InventoryItem*.

MoveItem

This class models a record that is used in the ***invloc*** and ***invhist*** tables. See the Javadoc for *MoveItem*.

HoldItem

This class models a record of the ***onhold*** table. See the Javadoc for *HoldItem*.

data modeler support classes

To guarantee that the primary keys (xxxx_id fields) for database records will be unique, there shall be a database mechanism that holds the value of the last numeric primary key issued. Whenever a new record is created, the numeric part of the new record's primary key shall be generated by an atomic operation that increments the 'last_value' in the database. The incremented value shall be the numeric part of the new primary key.

AgentID

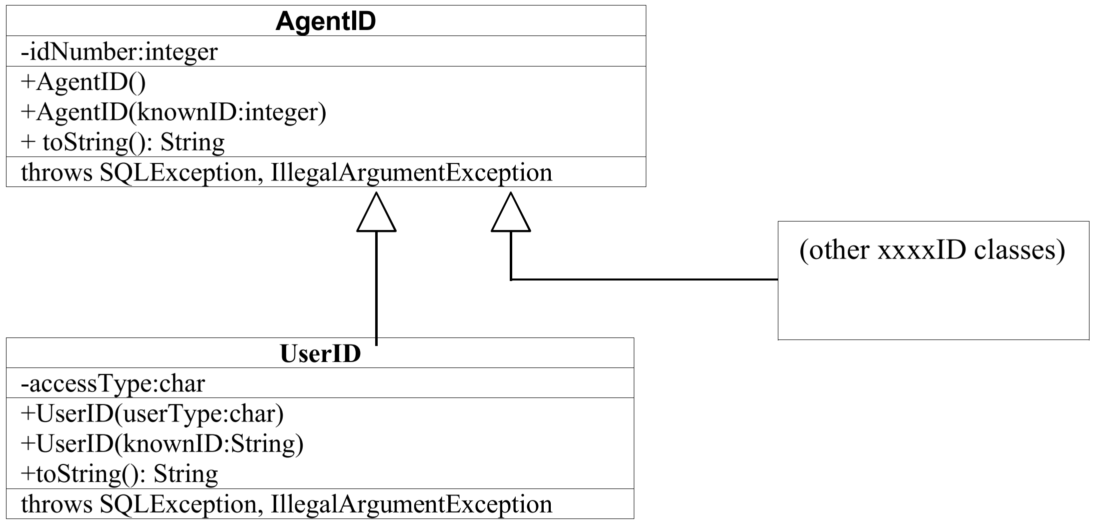
This class shall implement the atomic operation that increments 'last_value' and returns the unique numeric value for other primary key classes to use.

The AgentID() constructor instantiates an object with a unique idNumber.

The AgentID(knownID) constructor instantiates an object that has idNumber set to the value of knownID. The purpose of constructing an object this way is to support xxxxID subclasses when an xxxxID object is used to match an existing record's primary key.

The toString() method shall return a '0' padded (n?) character string of the value of idNumber

T. Fox, T. Nguyen, T. Pham



The description below for the subclass UserID is typical of all descendants of AgentID. See the javadocs for *AgentID* and its subclasses.

UserID extends AgentID

Calling the constructor UserID(userType:char)

- if UserType is not one of [‘A’,’L’,’P’], throws IllegalArgumentException. If good arg, saves it in accessType.
- calls super().This operation may throw SQLException.

Calling the constructor UserID(knownID:String)

- if the first character of string knownID is not one of [‘A’,’L’,’P’], throws IllegalArgumentException. If good arg, saves it in accessType.
- if the remaining characters in knownID do not parse as an integer (try … catch NumberFormatException), throws IllegalArgumentException. If success, calls super (knownID) with the good integer.
- sends query “SELECT * FROM users WHERE (user_id = “ + knownID + “)” . This operation may throw SQLException. If the result set contains anything other than one row, throw IllegalArgumentException. (If there are multiple rows, the database is corrupt. How to handle this?)

toString() shall return a string consisting of the accessType char and super.toString().

Library Manager <> Document History CIS-35B

T. Fox, T. Nguyen, T. Pham

Date	**Item**	**Action**
22 July 2004	Detailed class descriptions and UML were cluttering up the document. Some methods in the data modeler classes really belonged in the transaction classes.	Document name changed to LMDesign.doc. Old document saved as BaseClasses01.doc. Removed transaction detail from data modeler section to transaction section. Condensed the separate detailed data modeler descriptions into a generic description.
24 July 2004	Implementing AgentID and its subclasses made it necessary the all string `xxxx_id` fields be the same length, `CHAR(10)`. `catalog_id` had previously been defined as `CHAR (9)`.	Changed field definitions for `catalog_id` to `CHAR(10)`.
09 August 2004	Changed intention of document from Design plan (speculative) to Design / Implementation (actual).	Removed printouts of database table contents. Removed detail description of data modeler class functionality, cited javadocs for these classes. Added GUI screen shots and descriptions.
12 August 2004		Added screen shots from Tue's code.