NYU Used Books Marketplace
Design and Planning Document

March 11, 2005
Version 1.3

1 Document Revision History

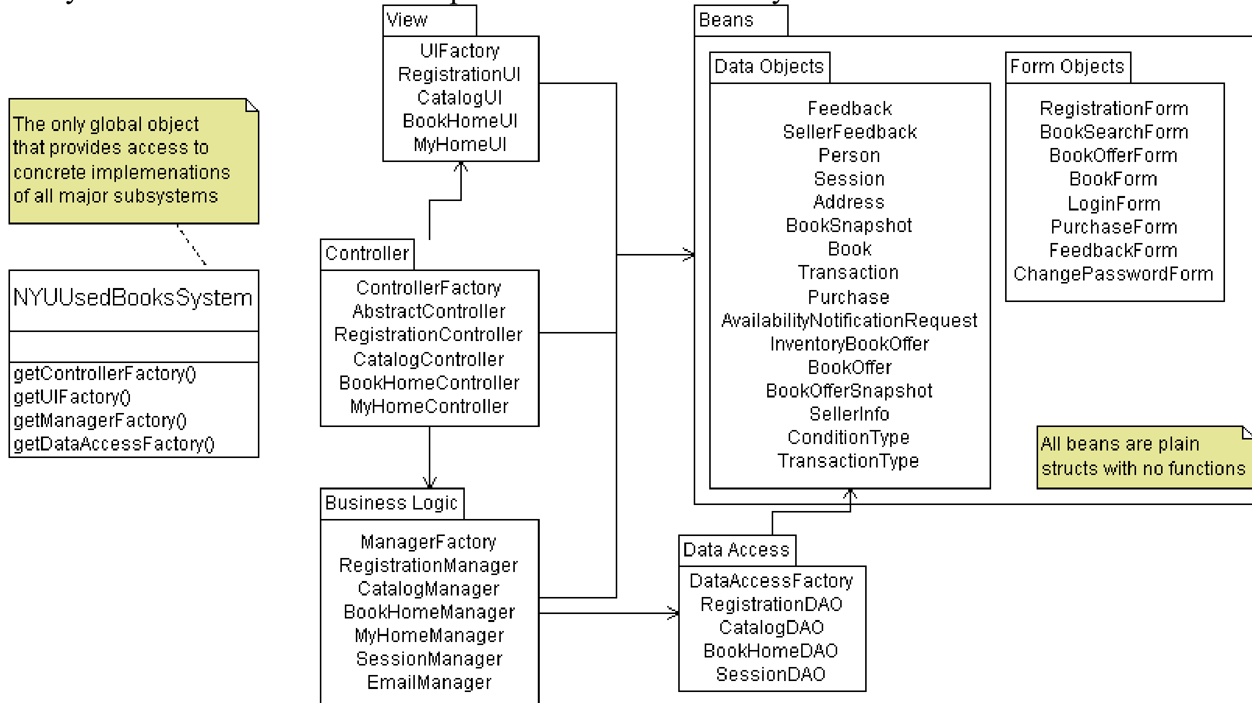
Rev. 1.0 March 9, 2005 – initial version
Rev. 1.1 March 9, 2005 – added more explanations for the diagrams
Rev. 1.2 March 9, 2005 – some typos corrected
Rev. 1.3 March 11, 2005 – fixed some of the diagrams

2 System Architecture

A major design goal of this system is to have loose coupling and separation of concerns among various components. This is achieved in part by using the Model-View-Controller pattern, where the presentation layer is decoupled from the business logic and data access layers.

2.1 *Subsystems*

The following diagram shows all major subsystems, where a package corresponds to a subsystem. The arrows indicate dependencies between subsystems.



2.1.1 NYUUsedBooksSystem Class

The NYUUsedBooksSystem class is meant to be the only global object in the whole system. It provides an interface to all major subsystems through the getter methods.

2.1.2 Controller Package

The job of the controller package is to coordinate the HTTP requests that the system receives from the browser. When the controller layer receives a request, it processes it by making calls to the business logic layer and then selecting an appropriate view in the presentation layer.

2.1.3 Business Logic Package

The business logic package contains all of the classes that are concerned with the various business rules. For example, the process of whether and how a new book should be added to the catalog is implemented somewhere in this package. Another example is the rule that only NYU students with valid @nyu.edu email addresses are allowed to register is also controlled here. In addition, the business logic layer is the only subsystem that depends on the Data Access subsystem directly.

2.1.4 Data Access Package

The only concern of the data access package is to access the database. It hides all the details about accessing the database. It provides a convenient interface for the business logic layer to access the database, but it does not implement any business rules. For example, if the business logic subsystem needs to retrieve an array of all books that a certain person is selling, it can simply make a call to the appropriate function in the data access subsystem passing it the id of the seller and get back an array of book offers. The data access subsystem will take care of running the appropriate select query and populating the array of objects with the data contained in the result set.

2.1.5 Beans Package

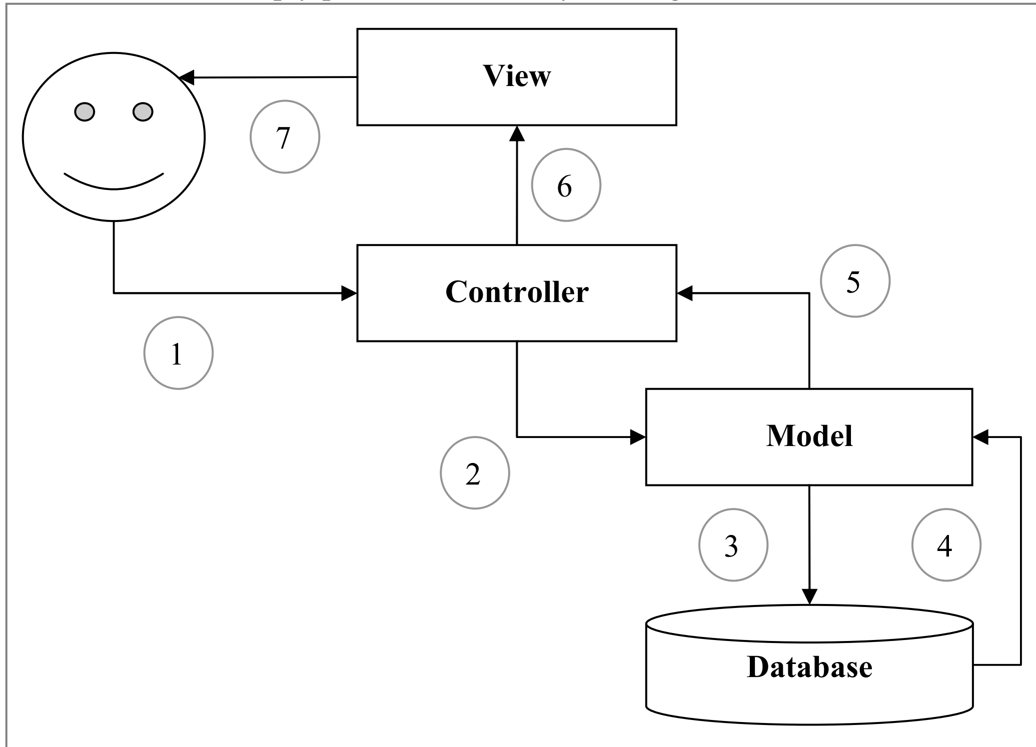
The beans package is simply a collection of data structures that get passed around between the various layers. None of the objects in this package have any methods - they have absolutely no behavior. The data structures are organized into two different sub-packages just for convenience. The data objects package contains all the objects that roughly correspond to tables in the database. However some of the objects are composed of several other "simple" objects. The form objects package is a collection of objects that are plain containers for form data of various forms that the users are asked to fill out.

2.2 Typical Request Cycle

Below you'll find the explanation and the diagram that represents a typical request cycle in the application.

1. The user clicks on a link or a button and an HTTP request is sent to the application.
2. The controller subsystem analyzes and parses the request. It then makes calls to the model (business logic layer) to complete the requested actions and/or retrieve information.
3. The business logic subsystem probably needs to access the database to store or retrieves the requested information. So, it makes a call to the data access subsystem.
4. The data access subsystem stores or retrieves the information in the database and returns it to the business logic subsystem.
5. The business logic subsystem might do something more with the data or perform other actions but then it finally returns the requested data to the controller.

6. Based on the response from the business logic layer, the controller picks a view and passes any needed information to it that it got from the business logic layer.
7. The view simply presents the data by sending back HTML to the browser.

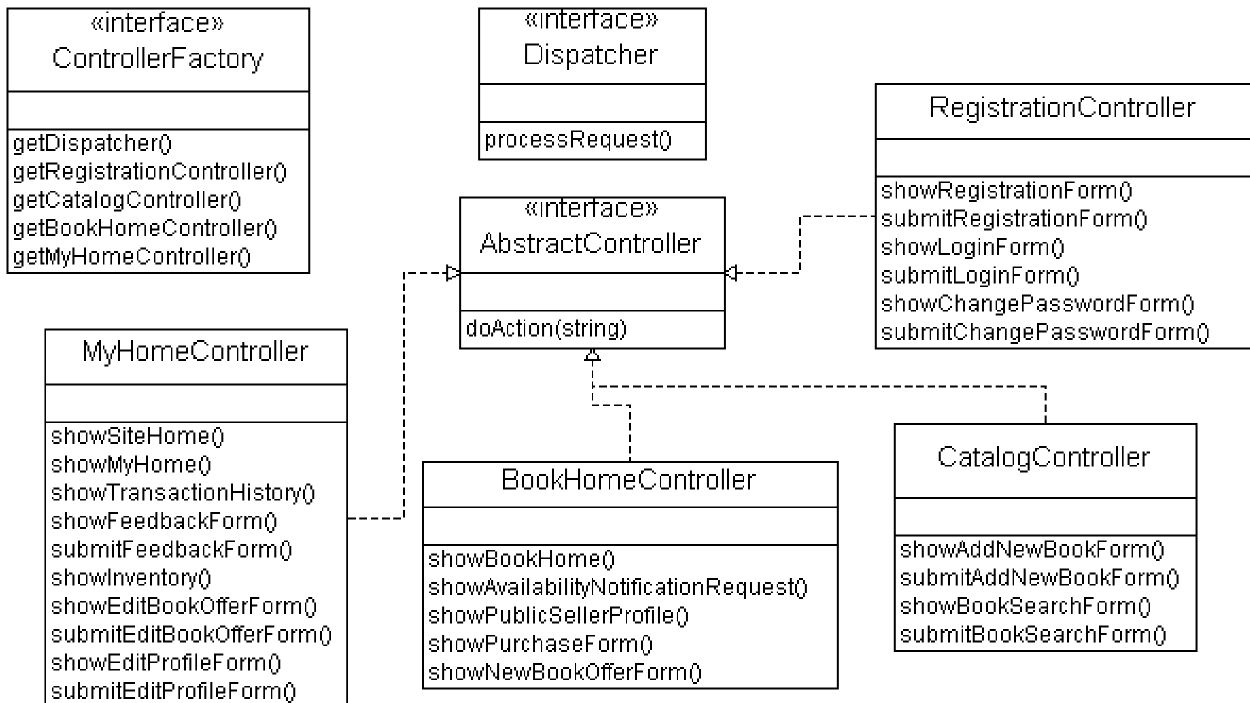


3 Class Design

Every major subsystem follows the Abstract Factory design pattern.

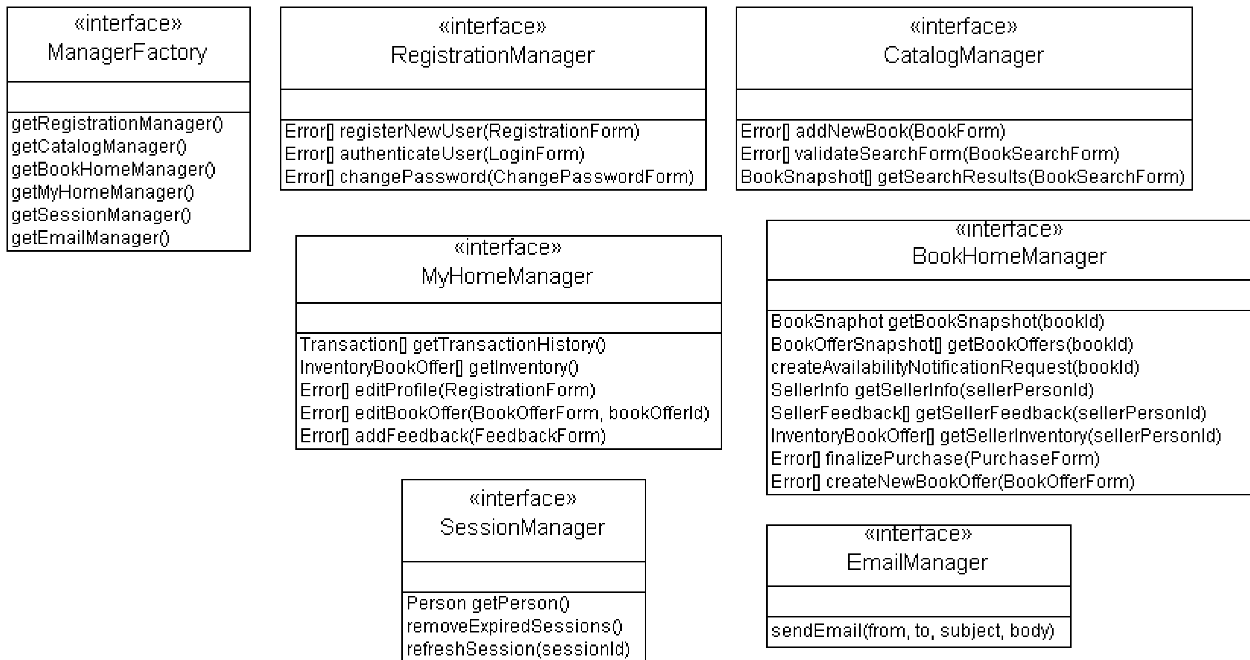
3.1 Controller

The ControllerFactory interface provides access to all concrete implementations of the AbstractController interface. Every request first comes to the dispatcher which determines which controller module to invoke. Then controller functionality is distributed among four different modules (controller classes), each responsible for a certain portion of the system.



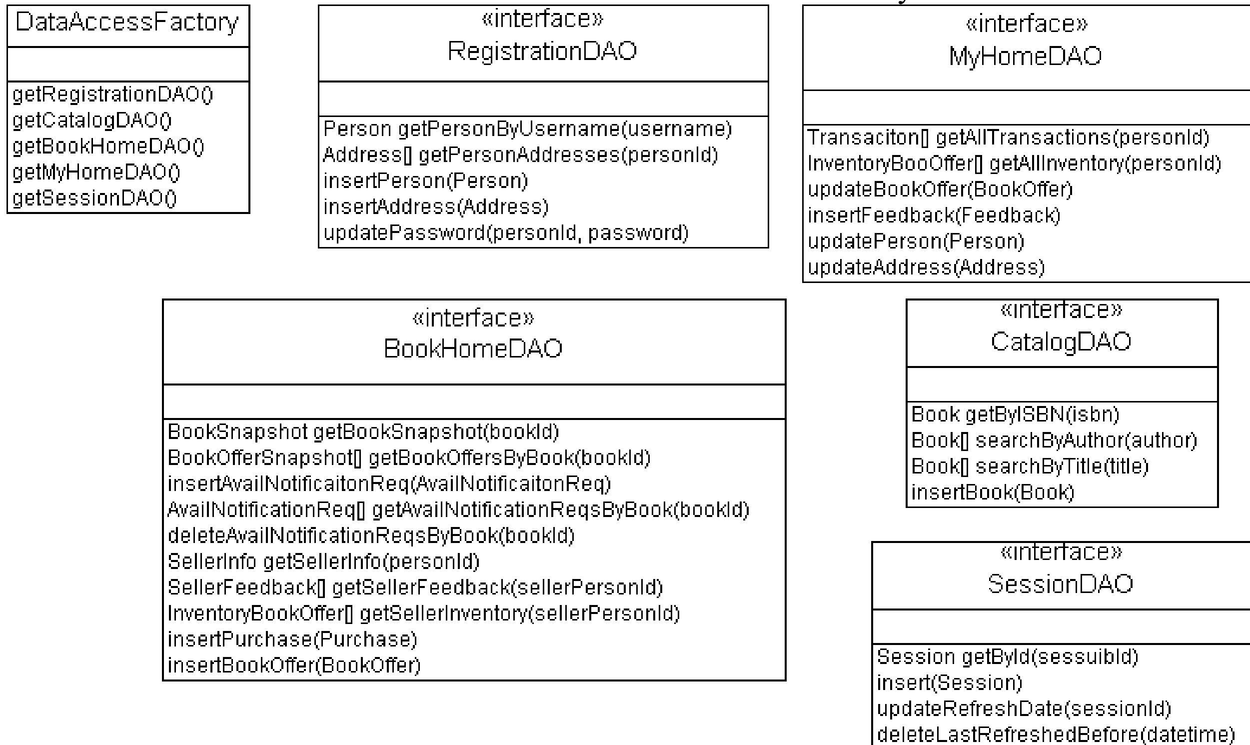
3.2 Business Logic Layer

Below is a diagram for a collection of interfaces for all major business logic components. The ManagerFactory interface is the abstract factory for concrete implementations of the components. Some methods return an array of Errors. Error is not a class but just an integer error code. Therefore, where it says Error[], it can be read as int[].



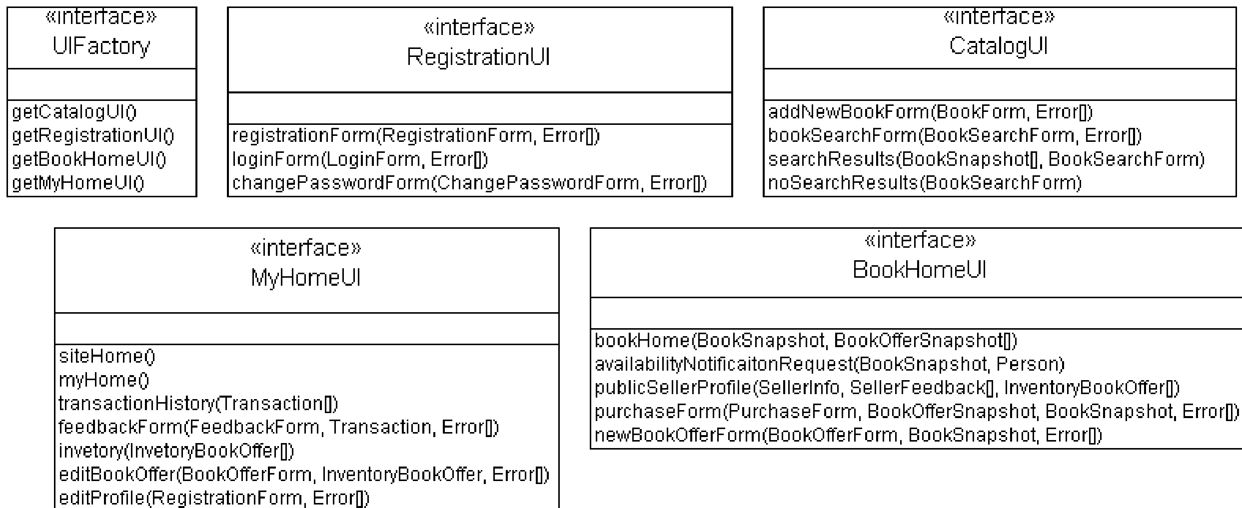
3.3 Data Access Layer

These are the interfaces to classes that will access the database directly.



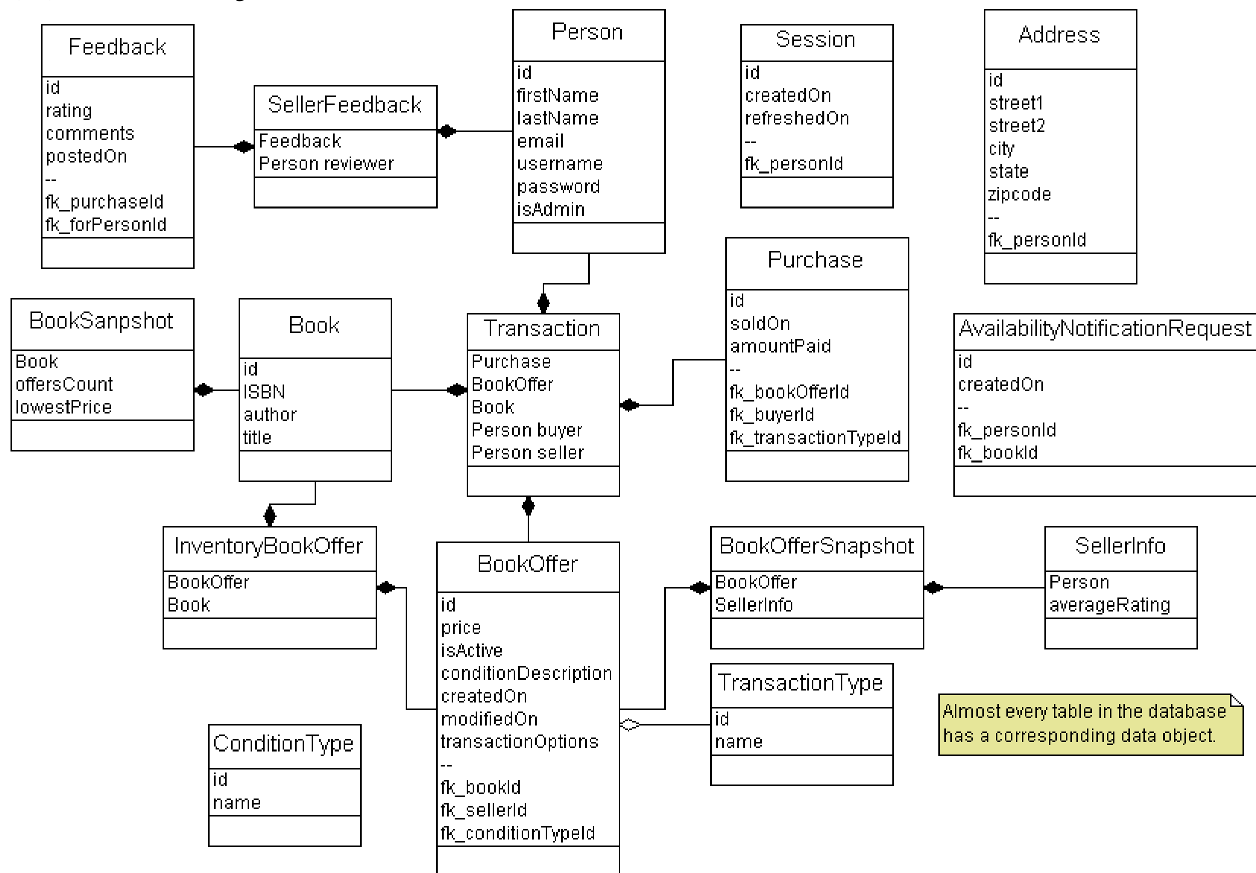
3.4 Presentation Layer

Below are the interfaces for classes that will be responsible for creating the dynamic html pages.



3.5 Beans

3.5.1 Data Objects



3.5.2 Form Objects

RegistrationForm
firstName lastName email username street1 street2 city zipcode

BookForm
ISBN author title

FeedbackForm
transactionId rating comments

LoginForm
username password

BookOfferForm
bookId isOnlineTransactionAvail isInPersonTransactionAvail price conditionTypeId conditionDescription

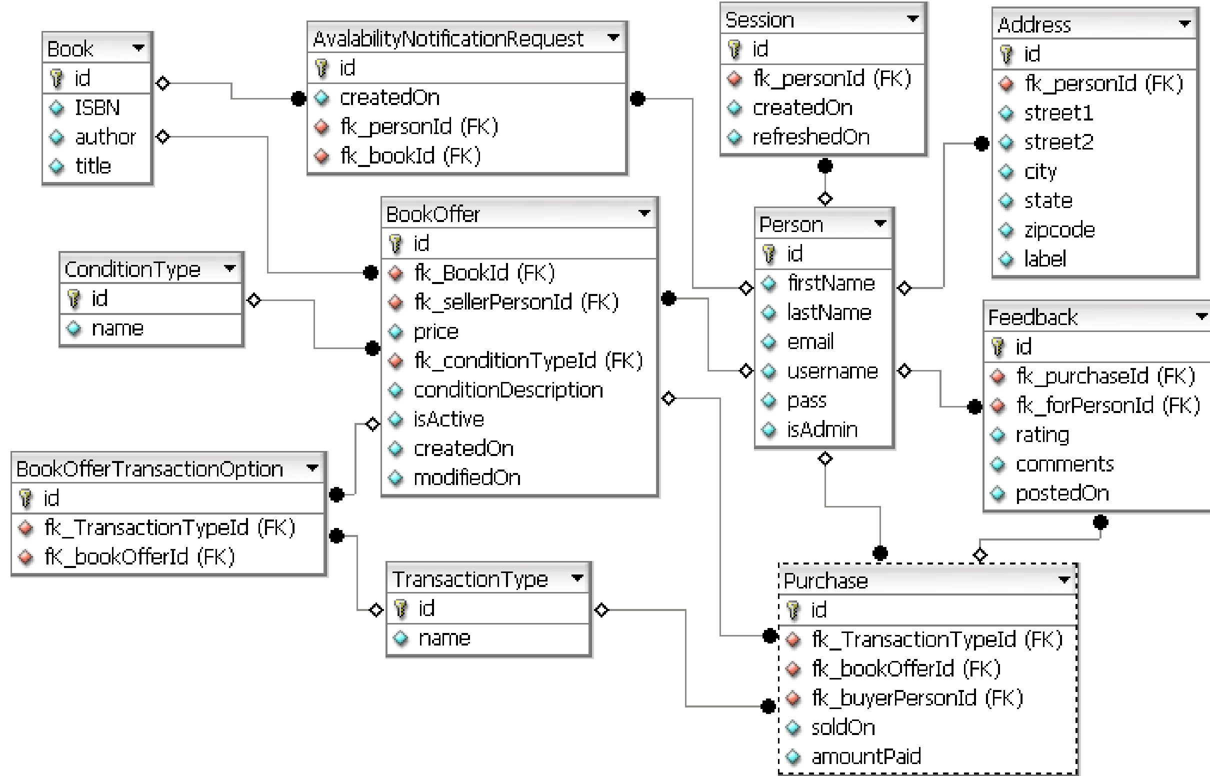
ChangePasswordForm
username passwordOld passwordNew

BookSearchForm
searchField searchString

PurchaseForm
bookOfferId selectedTransctionTypeId

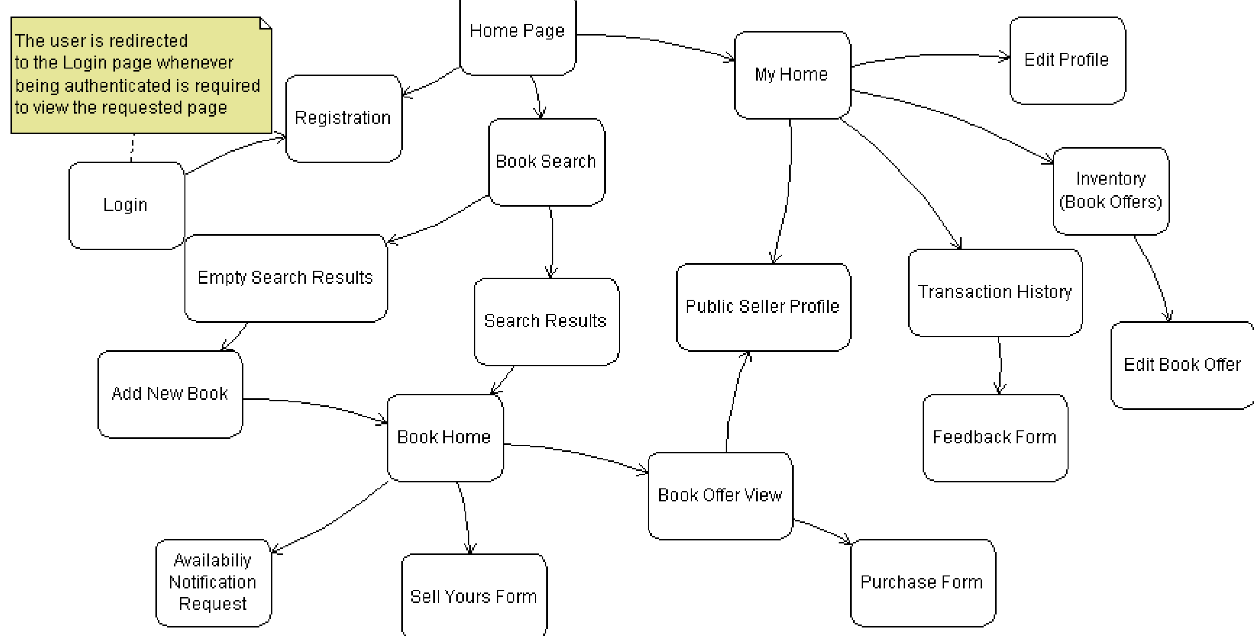
4 Detailed Design

4.1 Database Schema



4.2 Site Map

This is a map of possible transitions between different pages.



4.3 Miscellaneous Design Details

- Passwords will need to be stored in the database in encrypted form.
- NYU students will be validated based on their @nyu.edu email address to which the initial password will be sent as a protection. Thus, a malicious user won't be able to get into the system if he/she used a fake @nyu.edu email address.
- There will be only one central CGI script to which all HTTP requests will be sent. The action to be taken by the script will be specified by the action parameter. The format of the action parameter is <module.method>. For example, the URL to display the registration form might look something like this: /index.cgi?action=registration.showRegistrationForm. And the URL to submit the registration form might look something like this: /index.cgi?action=registration.submitRegistrationForm.

5 Testing plan

The system will be easy to test because the persistent state of the application across multiple requests is stored in the database, including any session information. Therefore, it will be easy to set up tests that will run on a replica of the database. No other state information needs to be initialized for testing.

5.1 Unit Testing

All major subsystems are very loosely coupled with interfaces into which alternative implementations or stubs can be easily plugged in for unit tests. Thus, most subsystems can be easily tested in isolation using unit tests.

5.2 Integration Testing

We plan to use the Top-Down Integration strategy, where in the beginning we will provide simple stub implementations for all major interfaces and get a basic system working right away and then incrementally add real functionality. For example, the presentation layer can initially contain "static" pages without any real data coming from the model.

5.3 System Testing

We plan to test the system as a whole manually by running use case acceptance tests. The customers that we have contacted will help us with the manual testing.

5.4 Regression Testing

We plan to automatically build the system every night and if the system compiled, unit tests that exhibit previously discovered bugs will be automatically run after the nightly compilation.

5.5 Load Testing

We plan to set up some kind of automated loading testing on the system because we have the requirement that the system must be able to operate with up to 50,000 users.

6 Plan

The system will start with an initial stub build and then will be continuously evolved into a full working system. We do not plan to have explicit system builds.
Each unit of time corresponds roughly to one full day of work.

Task	Time Estimate	Assigned To
Initial stub system with "empty" methods	3 units	Mike
RegistrationDAO and SessionDAO (database layer)	5 units	Casey
CatalogDAO (database layer)	5 units	Kashik
BookHomeDAO (database layer)	5 units	Danny
MyHomeDAO (database layer)	5 units	Mike
RegistrationManager (business logic layer)	2 units	Casey
CatalogManager (business logic layer)	2 units	Kaushik
BookHomeManager (business logic layer)	2 units	Danny
MyHomeManager (business logic layer)	2 units	Mike
SessionManger (business logic layer)	2 units	Casey
EmailManager (business logic layer)	2 units	Kaushik
RegistrationController	2 units	Danny
CatalogController	2 units	Mike
BookHomeController	2 units	Casey
MyHomeController	2 units	Kaushik
RegistrationUI (presentation layer)	2 units	Danny
CatalogUI (presentation layer)	2 units	Mike
BookHomeUI (presentation layer)	2 units	Casey
MyHomeUI (presentation layer)	2 units	Kashik

7 Checklist

The following checklist is provided to help you think about whether the document is complete and correct. These questions will also be used by the team which reviews your design. You may want to add your own addition questions to the list.

7.1 *Content*

- Are all the interfaces to the system specified in detail?
- Are acceptable solution alternatives and their trade-offs specified?
- Are you satisfied with all parts of the document?
- Do you believe all parts are possible to implement?
- Are all parts of the document in agreement with the product requirements?
- Is there a risk associated with the proposed design? Is it discussed in the document?
- Will the goal for each type of testing be met with the testing that is described?
- Are the testing activities scheduled at the appropriate times?
- Is each part of the document in agreement with all other parts?

7.2 Completeness

- Have alternative designs been considered and documented?
- Are the limitations of the specified implementation sufficiently documented?
- Are dependencies and assumptions thoroughly documented?
- Where information isn't available before review, are the areas of incompleteness specified?
- Are all the testing stages covered?
- Is the regression test strategy covered?
- Are there high-level validation tests necessary and covered (performance, security, etc.)?

7.3 Clarity

- Is the solution at a fairly consistent and appropriate level of detail?
- Is the solution clear enough to be turned over to an independent group for implementation and still be understood?
- Is the control flow and the data flow clear?
- Is each requirement (or feature list item) measurable (will it be possible for independent testing to determine whether each requirement has been satisfied)?
- Is the test strategy for each type of testing clearly described?
- Are the test report requirements clearly described?
- Are all items clear and not ambiguous? (Minor document readability issues should be handles off-line, not in the review.