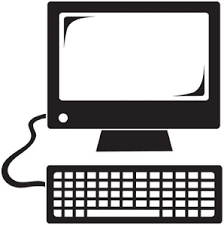
Library

Information

System



Phase II Requirements

Contents

[1. Architectural Design 3](#_Toc512195960)

[a. UML Deployment Diagram 3](#_Toc512195961)

[b. Deployment Diagram Description 4](#_Toc512195962)

[c. Allocation of Use Cases to Diagram 5](#_Toc512195963)

[2. Detailed Design 6](#_Toc512195964)

[a. UML Collaboration Diagrams with Descriptions 6](#_Toc512195965)

[1. Search 6](#_Toc512195966)

[2. Login 7](#_Toc512195967)

[3. Set Password 9](#_Toc512195968)

[4. Reset Password 11](#_Toc512195969)

[5. Request Personal Status Report 13](#_Toc512195970)

[6. Request Report 15](#_Toc512195971)

[7. Add Item 17](#_Toc512195972)

[8. Remove Item 19](#_Toc512195973)

[9. Register New User (Patron) 21](#_Toc512195974)

[10. Terminate Account 23](#_Toc512195975)

[11. Update Item Status 24](#_Toc512195976)

[12. Register New User (Librarian) 26](#_Toc512195977)

[13. Report Missing Item 27](#_Toc512195978)

[14. Place Hold on Account 30](#_Toc512195979)

[15. Suspend Account 32](#_Toc512195980)

[16. Patron reserves an item 34](#_Toc512195981)

[17. Patron check-out an item 35](#_Toc512195982)

[14. Design Class Diagram 37](#_Toc512195983)

[15. Class Design 38](#_Toc512195984)

[1. Login 38](#_Toc512195985)

[2. Report 40](#_Toc512195986)

[3. Account 41](#_Toc512195987)

[4. Item 42](#_Toc512195988)

[5. Notification 44](#_Toc512195989)

[3. Design Quality 45](#_Toc512195990)

# Architectural Design

## UML Deployment Diagram

## Deployment Diagram Description

There are two main architectural patterns that the Library Information Management System will utilize: Repository and Model-View-Controller (MVC). Because the application is a primarily transactional-based, with a large, common data store, a repository pattern is utilized to provide simple and standardized access to the database. An example of systems which make use of information from the database include the catalog management system, which indexes, queries, and modifies the collection of items that are available for checkout in the library. In addition to catalog items, the database will contain information regarding user accounts, with the repository pattern facilitating access to that data for the subsystems which depend on it.

Since our application is web-based, MVC pattern is well-suited for the purposes of our system. MVC is easy to adapt from the Boundary-Entity-Controller (BEC) pattern utilized in our Application Interaction Model. The Model classes are essentially augmentations of the concept of Entities. Models in MVC represent the data items as software objects and provide a limited set of core behaviors (no real business logic). These Models provide a way to map software to relational data which is stored in the database. Views are another component of MVC design adapted that are adapted from Boundary objects. Views make up the user interface, and they provide the means for the user to walk through use-cases and perform functions within the program. As actual software objects, MVC controllers can enclose a set of behaviors and use cases through simple transformation of BEC controllers. Our model shows a Client Tier which holds the Views that are visible in a clients web browser. The Web Tier hold the interfaces for the client. The Application Tier holds most of the information as it stores the business logic and the models that are used throughout. The Data Tier holds all of the data that is saved to the database by the controllers in the application tier.

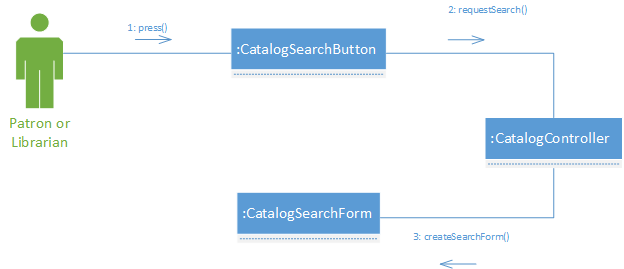
## Allocation of Use Cases to Diagram

In our MVC-based structure, all use cases will be initiated through the Views we have adapted from our boundary objects. The Login view subsystem within the Client device node will be used in the Login use case for both librarians and patrons and will be included as part of the Set Password and Reset Password use cases. Use cases relating to registering new accounts and Set/Reset password will also interact with the AccountManagement subsystem on the Client node. Other use cases that will use this View subsystem are the Terminate Account, Suspend Account, and Place Hold on Account use cases. Additionally, in requesting a Personal Status report, the user will interact with both the UserInfo and the Reporting subsystems on the Client device node. There are several use cases which are allocated to the Catalog View subsystem, and they are listed as follows: Add Item, Remove Item, Report Missing Item, Update Item Status, Reserve Item, Check-out Item and Search. Of these, Search will perhaps be the most commonly used Use Case within the entire system. Another integral subsystem is the Transaction subsystem, which is involved in the Update Item Status use case when a patron is checking out or returning an item.

# Detailed Design

## UML Collaboration Diagrams with Descriptions

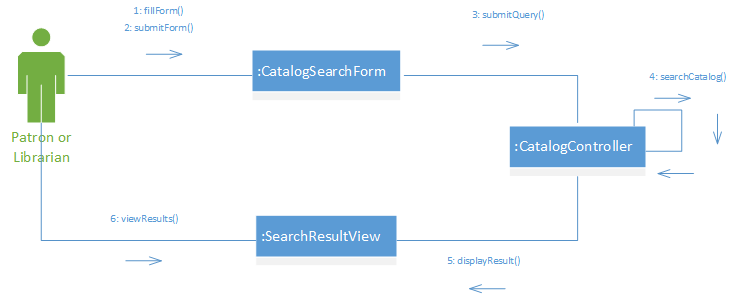
### Search

**1.1**

This graph utilizes the Controller pattern, implemented with the CatalogController. This class handles requests for action from the UI elements such as CatalogSearchButton. The catalog controller is responsible for being a sort of coordinator not only for the Search use case, but a larger set of use cases having to do with the catalog subsystem. Another pattern displayed is the Creator pattern, exemplified again by the CatalogController which acts as a factory for creating the view element CatalogSearchForm.

**1.2**

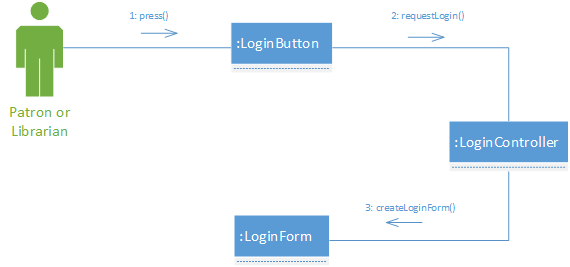
In this graph, the first GRASP pattern utilized is the Controller pattern. Like the previous graph, CatalogController coordinates system operations involved in this use case and is responsible for bridging the UI to the data. In the way the controller handles requests for a search query, it is also an Information Expert that knows what needs to be done to perform the operation and where it can find the data that the use case is requesting. Finally the CatalogController-SearchResultView relationship is an example of the Creator pattern, in which CatalogController is responsible for creating all instances of a SearchResultView to display the desired information to the user.



### Login

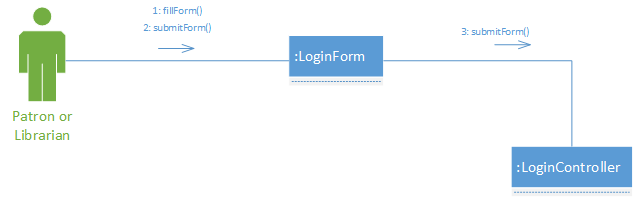
**2.1**

The first pattern exemplified in this collaboration graph is the Controller pattern. The LoginController handles requests for action given to it by UI elements, specifically LoginButton. LoginController holds a similar controller role across the Login subsystem and for login-related use cases. Because of the way the LoginController handles the creation of the LoginForm UI element, then we can apply the Creator pattern in this diagram as well.



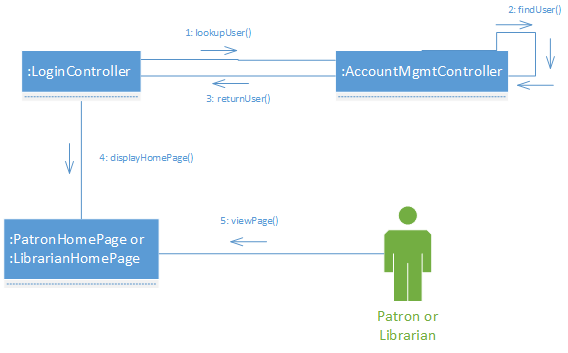
**2.2**

This collaboration graph simply shows the submission of the LoginForm by the user its presentation. Due to the simple nature of the graph, the only pattern used in this example is the Controller pattern. The Controller pattern is used for a similar reason as the previous diagram. The user requests an action (login) via the UI element which is the LoginForm, and it is sent to the LoginController for further action.



**2.3**

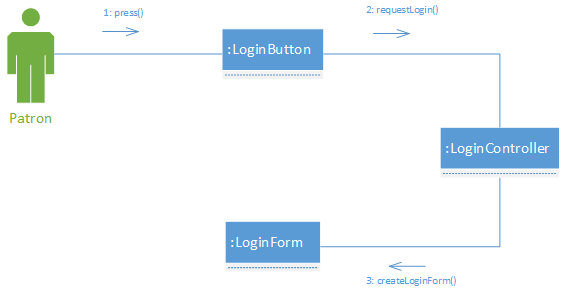
The Information Expert pattern is displayed by the AccountMgmtController, which is deferred to by the LoginController for the purposes of finding information on the user who is logging in. High Cohesion is shown by the separation of LoginController and AccountMgmtController for the handling of different kinds of operations. LoginController acts as a Creator for the HomePage, and once a user is authenticated, creates either the PatronHomePage or the LibrarianHomePage depending on who is initiating the use case.



### Set Password

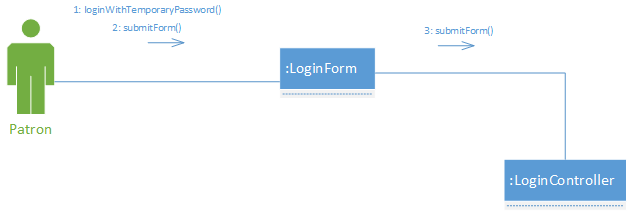
**3.1**

This use case includes elements from the standard Login use case. In the first collaboration diagram, the Controller pattern is shown by the relationship between LoginButton, LoginController, and LoginForm. The LoginController acts as the controller which coordinates actions based on user intent from the UI Elements. LoginController additionally acts as a Creator which is responsible for generating instances of the LoginForm.



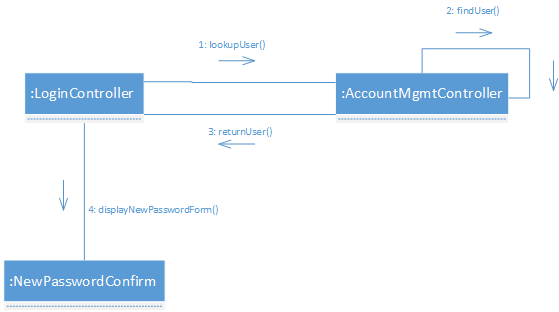
**3.2**

In this collaboration graph, the Controller GRASP pattern is exemplified by the LoginController class. After a user fills in the LoginForm presented to them in the previous diagram, the intent of the user to login to the system is passed from the UI element to the controller to continue on with the actions of the use case.



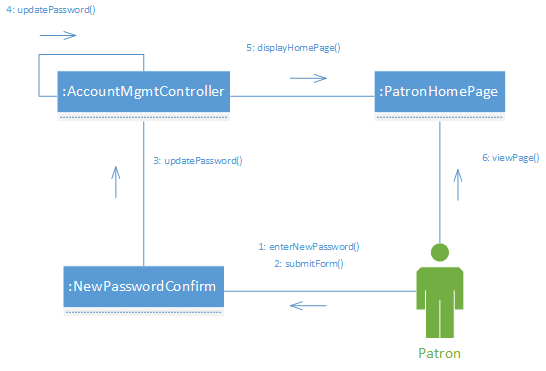
**3.3**

The principles of High Cohesion and Information Expert are shown in the collaboration diagram by the LoginController and the AccountMgmtController. Concerns associated with authentication and user management are separated into different classes, promoting High Cohesion. By this pattern, LoginController defers to AccountMgmtController which is an Information Expert regarding user and account information. LoginController additionally implements the Creator pattern because of the way that it instantiates NewPasswordConfirm UI objects.



**3.4**

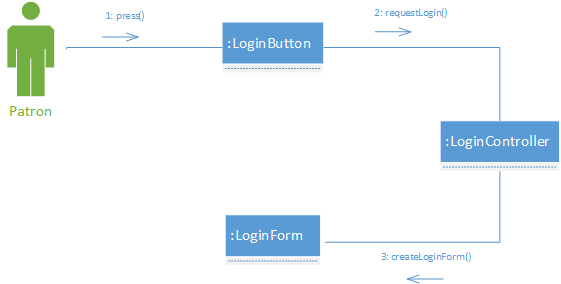
There are several GRASP guidelines utilized in this collaboration graph, the first of which is the Controller pattern. AccountMgmtController takes on this role for this segment of the use case which has to deal with actually setting the user’s password. When the Patron submits the NewPasswordConfirm, the responsibility is passed on to the controller to perform the business logic. After the user’s password is updated, the AccountMgmtController takes on the role by conveying the results back to the user by displaying the PatronHomePage. In this respect, AccountMgmtController also acts as a Creator which instantiates the PatronHomePage.



### Reset Password

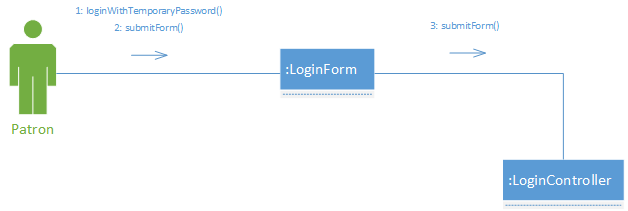
**4.1**

This collaboration graph exemplifies the Controller pattern with the LoginButton-LoginController and the LoginForm-LoginController relationships. LoginController handles the actions in this use case which are triggered by user input and then carrying out business logic within itself.



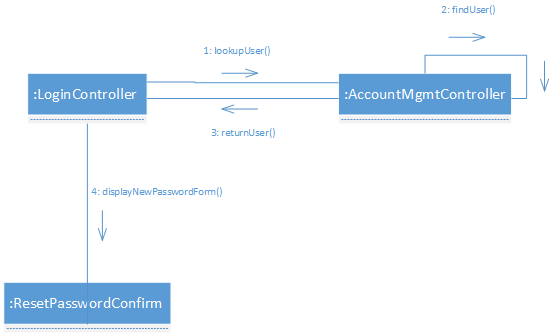
**4.2**

Similar to the previous graph, LoginController passes on user intent to the LoginSubsystem to perform actions on its end. This is an example of the Controller GRASP pattern in the same way the previous collaboration graph for this use case did.



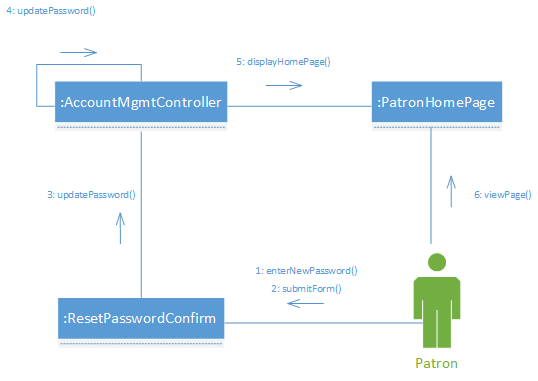
**4.3**

By the separation of concerns exhibited by LoginController and AccountMgmtController, this diagram implements the High Cohesion pattern. Authentication and user management actions are segregated into the two classes, promoting the GRASP guideline. Hand-in-hand with this principle is the Information Expert pattern, which is shown by the way AccountMgmtController has operations regarding account information deferred to it by LoginController, which is strictly for authentication based on the existing credentials. LoginController also acts in a way consistent with the Creator pattern because of the way it is responsible for the creation of ResetPasswordConfirm UI objects.



**4.4**

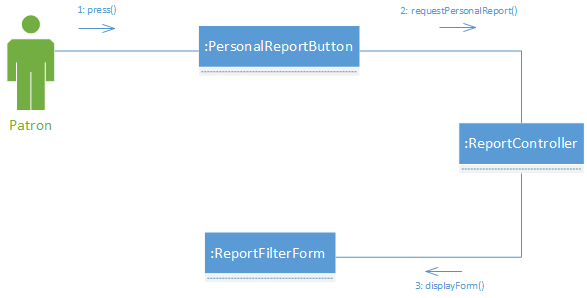
In this graph, the Controller pattern is shown by the AccountMgmtController. This controller takes on the role of handling Patron intent upon the submission of the ResetPasswordConfirm form and enacting the actual changes to the user’s account. Following this operation, the AccountMgmtController takes the user back to his home page, which is another example of the orchestration typical of the Controller pattern. Again, in this diagram, the Creator pattern is shown, this time by the AccountMgmtController. This class is responsible for the instantiation of the PatronHomePage, exemplifying the pattern.



### Request Personal Status Report

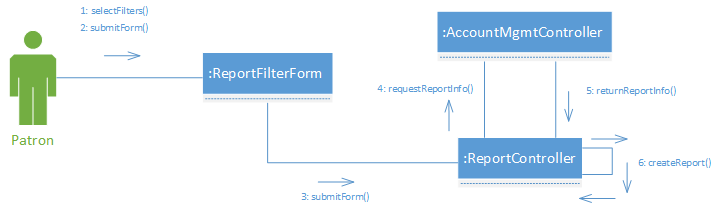
**5.1**

In this collaboration graph, the Controller GRASP pattern is implemented by the ReportController as shown by its interactions with PersonalReportButton and ReportFilterForm. When the Patron interacts with the PersonalReportButton, the ReportController fulfills the user’s intent by presenting to the Patron a ReportFilterForm. ReportController additionally acts as a Creator which is responsible for instantiating the ReportFilterForm.



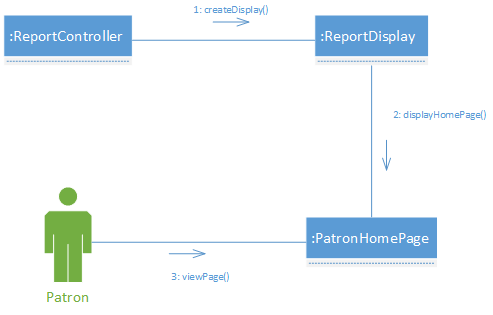
**5.2**

ReportController again fulfills the Controller GRASP guideline by encapsulating the business logic for the use case and taking action based on user interaction with the UI elements. After the Patron submits the ReportFilterForm, the ReportController takes the next step in the use case and defers to the AccountMgmtController, which acts as an Information Expert. This controller is knowledgeable of the information requested for the report and passes it back to the ReportController. This is also an implementation of the High Cohesion principle, because of the way each controller has a defined scope of operations regarding this use case.



**5.3**

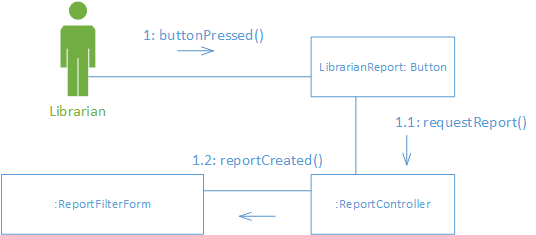
This collaboration graph models the Indirection pattern by supplying ReportDisplay as a mediator between ReportController and PatronHomePage. The data passed on from ReportController is carried on to a UI element on the PatronHomePage by the ReportDisplayObject.



### Request Report

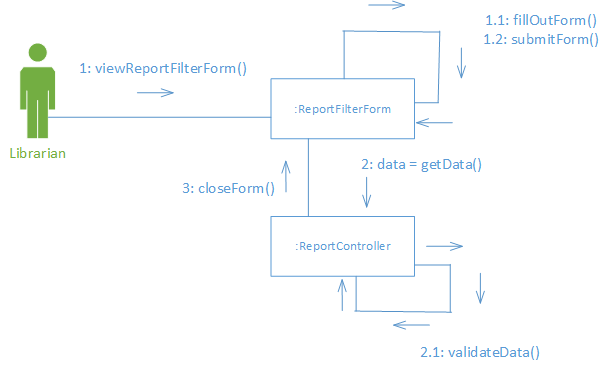
**6.1**

In this collaboration graph, the Controller GRASP pattern is implemented by the ReportController as shown by its interactions with LibrarianReportButton and ReportFilterForm. When the Librarian interacts with the LibrarianReportButton, the ReportController fulfills the user’s intent by presenting to the Librarian a ReportFilterForm. ReportController additionally acts as a Creator which is responsible for instantiating the ReportFilterForm.

****

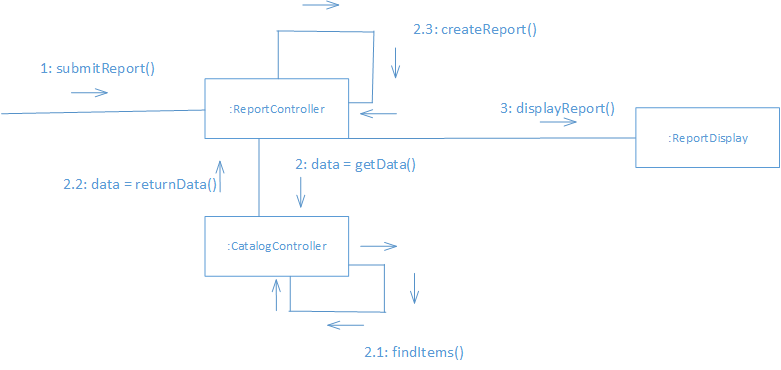
**6.2**

ReportController again fulfills the Controller GRASP guideline by encapsulating the business logic for the use case and taking action based on user interaction with the UI elements. This is also an implementation of the High Cohesion principle, because of the way each controller has a defined scope of operations regarding this use case.

****

**6.3**

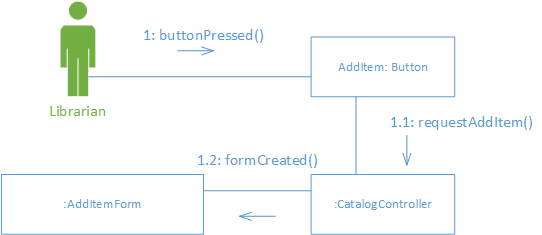
This collaboration graph models the Indirection pattern by supplying ReportDisplay as a mediator between ReportController and LibrarianHomePage. The data passed on from ReportController is carried on to a UI element on the LibrarianHomePage by the ReportDisplayObject. After the Librarian submits the ReportFilterForm, the ReportController takes the next step in the use case and defers to theCatalogController, which acts as an Information Expert. This controller is knowledgeable of the information requested for the report and passes it back to the ReportController.



### Add Item

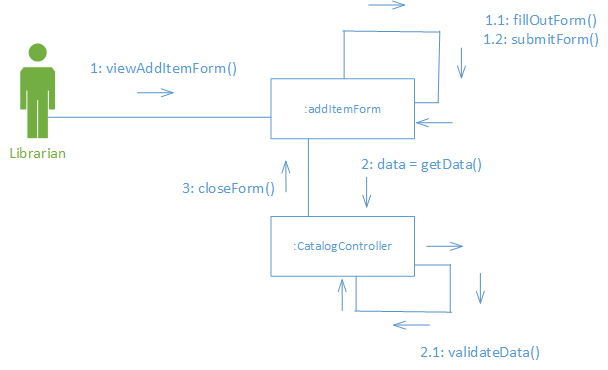
**7.1**

In this collaboration graph, the Controller GRASP pattern is implemented by the CatalogController as shown by its interactions with AddItemButton and AddItemForm. When the Librarian interacts with the AddItemButton, the CatalogController fulfills the user’s intent by presenting to the Librarian a AddItemForm. CatalogController additionally acts as a Creator which is responsible for instantiating the AddItemForm.

****

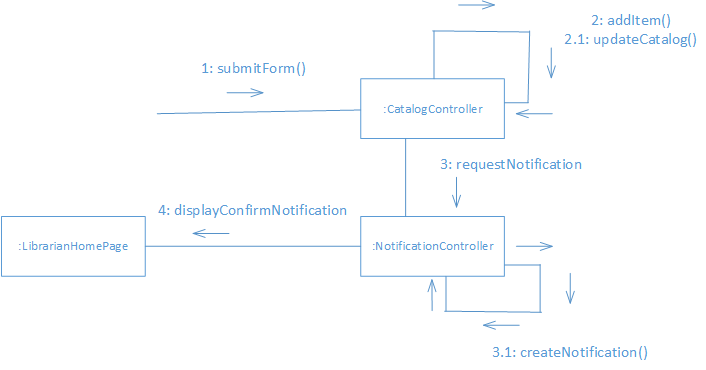
**7.2**

CatalogController again fulfills the Controller GRASP guideline by encapsulating the business logic for the use case and taking action based on user interaction with the UI elements. This is also an implementation of the High Cohesion principle, because of the way each controller has a defined scope of operations regarding this use case.

****

**7.3**

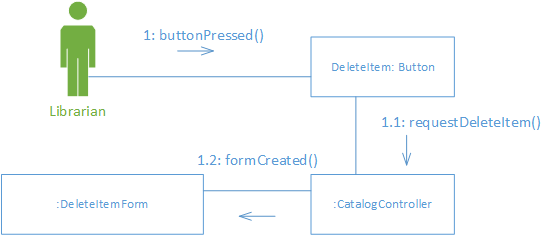
The data passed from CatalogController is carried on to a UI element on the LibrarianHomePage by a notification. After the Librarian submits the AddItemForm, the CatalogController takes the next step in the use case and adds the item to the catalog. This controller is knowledgeable of the information requested to be added. After an item is added then a notification is requested from the notification controller and sent to the librarian home page.

****

### Remove Item

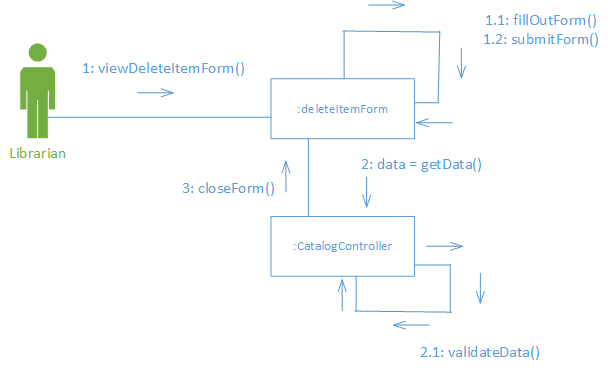
8.1

In this collaboration graph, the Controller GRASP pattern is implemented by the CatalogController as shown by its interactions with DeleteItemButton and DeleteItemForm. When the Librarian interacts with the DeleteItemButton, the CatalogController fulfills the user’s intent by presenting to the Librarian a DeleteItemForm. CatalogController additionally acts as a Creator which is responsible for instantiating the DeleteItemForm.



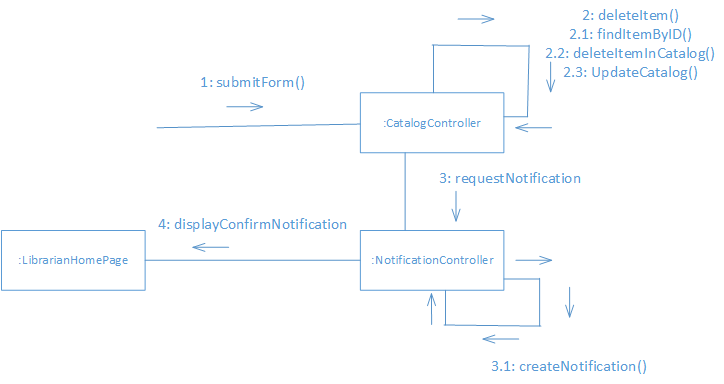
8.2

CatalogController again fulfills the Controller GRASP guideline by encapsulating the business logic for the use case and taking action based on user interaction with the UI elements. This is also an implementation of the High Cohesion principle, because of the way each controller has a defined scope of operations regarding this use case.



8.3

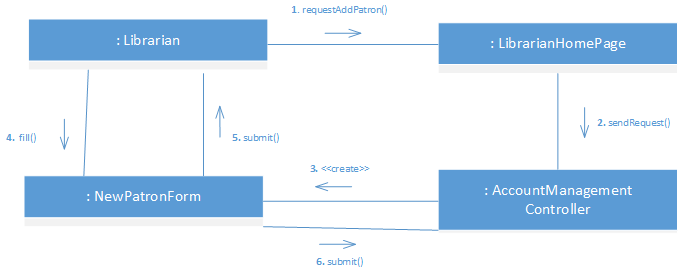
The data passed from CatalogController is carried on to a UI element on the LibrarianHomePage by a notification. After the Librarian submits the DeleteItemForm, the CatalogController takes the next step in the use case and deletes the item in the catalog. This controller is knowledgeable of the information requested to be deleted. After an item is deleted then a notification is requested from the notification controller and sent to the librarian home page.



### Register New User (Patron)

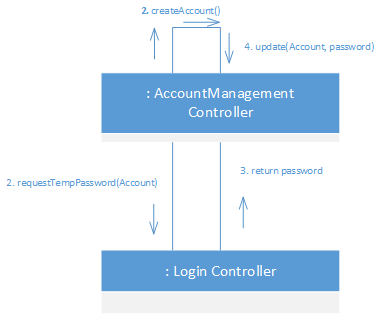
**9.1**

In this collaboration graph, the Controller GRASP pattern is implemented by the AccountManagementController as shown by its interaction with LibrarianHomePage and NewPatronForm. When the Librarian interacts with the LibrarianHomePage, the AccountManagementController fulfills the user’s intent by presenting to the Librarian a NewPatronForm. AccountManagementController also acts as a creator because it instantiates NewPatronForm.



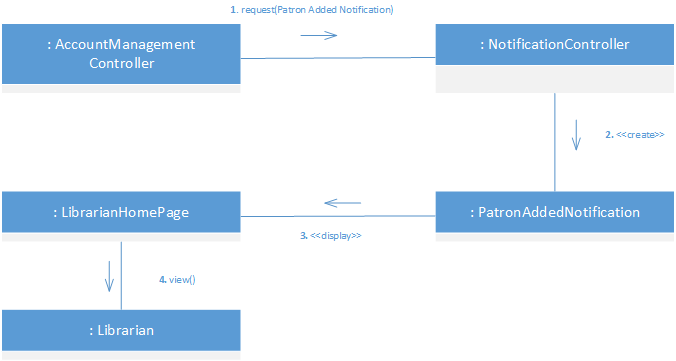
**9.2**

AccountManagementController again fulfills the Controller GRASP guideline. Once a NewPatronForm has been submitted (shown in **9.1**), AccountManagementController handles the actual creation of the Patron’s account using the submitted form, communicating with the LoginController to associate a temporary password with the Patron’s account, and finally interacts with the User Database to add the new account to the database.



**9.3**

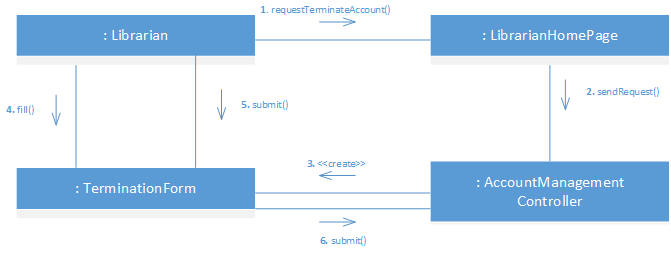
NotificationController fulfills the Controller and Creator GRASP guidelines. Once a new Patron account has been created (shown in **9.2**), it receives a request from the AccountManagementController to create a new Notification and handles this request by creating the PatronAddedNotification which then displays itself on the LibrarianHomePage.



### Terminate Account

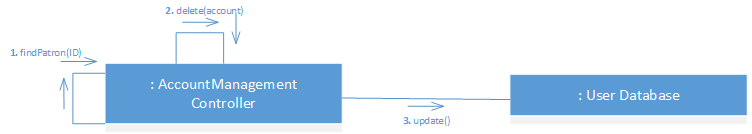
**10.1**

In this collaboration graph, the Controller GRASP pattern is implemented by the AccountManagementController as shown by its interaction with LibrarianHomePage and TerminationForm. When the Librarian interacts with the LibrarianHomePage, the AccountManagementController fulfills the Librarian’s request by presenting to them the TerminationForm. AccountManagementController additionally acts as a Creator since it is responsible for instantiating the TerminationForm.



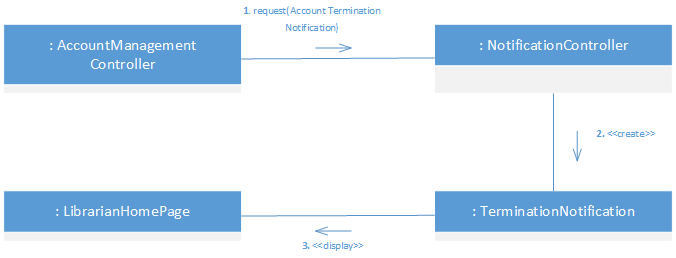
**10.2**

AccountManagementController again fulfills the Controller GRASP guideline. Once a TerminationForm has been submitted (show in **10.1**), AccountManagementController handles deletion of the account based on the info in the TerminationForm and updates the User Database with the deletion.



**10.3**

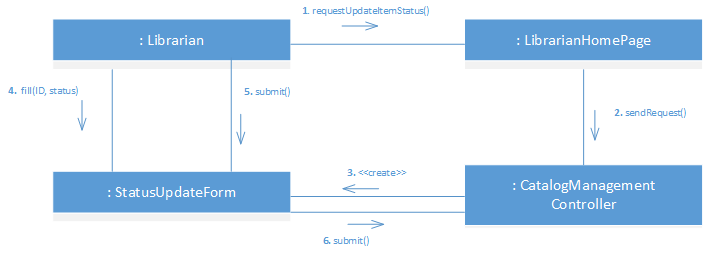
NotificationController fulfills the Controller and Creator GRASP guidelines. Once an account has been terminated (shown in **10.2**), NotificationController receives a request from the AccountManagementController to create a new Notification and handles this request by creating the TerminationNotification which then displays itself on the LibrarianHomePage.



### Update Item Status

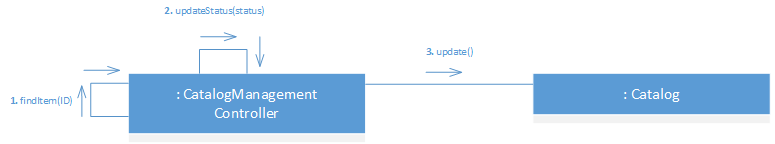
**11.1**

In this collaboration graph, the Controller GRASP pattern is implemented by the CatalogController as shown by its interaction with LibrarianHomePage and StatusUpdateForm. When the Librarian interacts with the LibrarianHomePage, the CatalogController fulfills the Librarian’s request by presenting to them the StatusUpdateForm. AccountManagementController additionally acts as a Creator since it is responsible for instantiating the StatusUpdateForm.



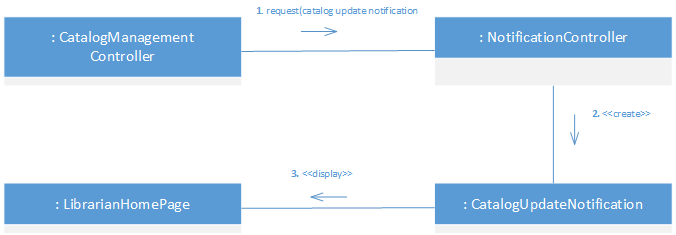
**11.2**

CatalogController again fulfills the Controller GRASP guideline. Once a StatusUpdateForm has been submitted (show in **11.1**), CatalogController handles updating an item in the Catalog based on the info in the StatusUpdateForm.



**11.3**

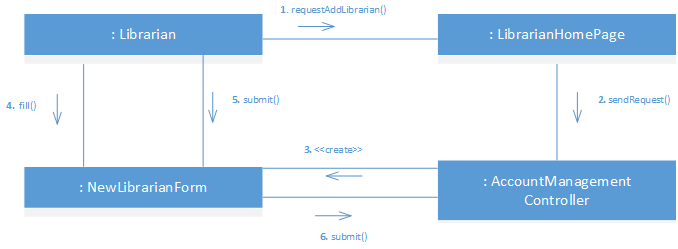
NotificationController fulfills the Controller and Creator GRASP guidelines. Once an item’s status has been updated (shown in **11.2**), NotificationController receives a request from the CatalogController to create a new Notification and handles this request by creating the CatalogUpdateNotification which then displays itself on the LibrarianHomePage.



### Register New User (Librarian)

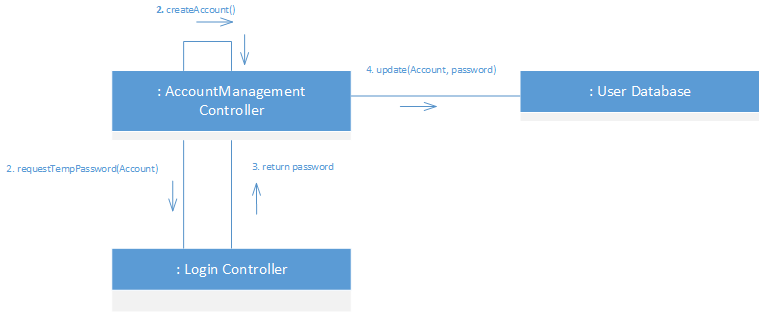
**12.1**

In this collaboration graph, the Controller GRASP pattern is implemented by the AccountManagementController as shown by its interaction with LibrarianHomePage and NewLibrarianForm. When the Librarian interacts with the LibrarianHomePage, the AccountManagementController fulfills the user’s intent by presenting to the Librarian a NewLibrarianForm. AccountManagementController also acts as a creator because it instantiates NewLibrarianForm.



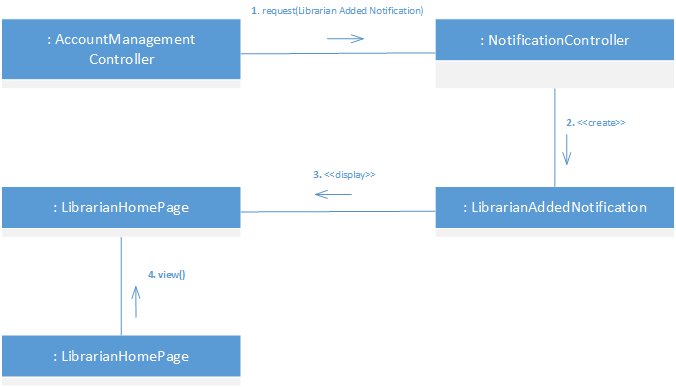
**12.2**

AccountManagementController again fulfills the Controller GRASP guideline. Once a NewLibrarianForm has been submitted (shown in **12.1**), AccountManagementController handles the actual creation of the Librarian’s account using the submitted form, communicating with the LoginController to associate a temporary password with the Librarian’s account, and finally interacts with the User Database to add the new account to the database.



**12.3**

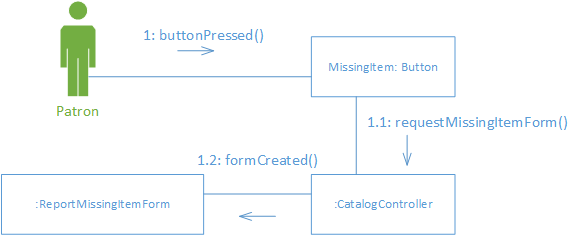
NotificationController fulfills the Controller and Creator GRASP guidelines. Once a new Librarian account has been created (shown in **9.2**), it receives a request from the AccountManagementController to create a new Notification and handles this request by creating the LibrarianAddedNotification which then displays itself on the LibrarianHomePage.



### Report Missing Item

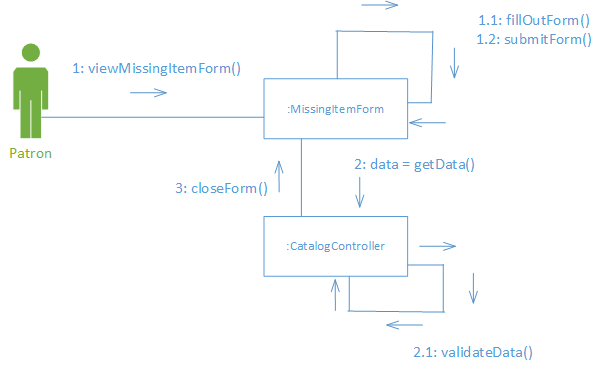
**13.1**

This collaboration graph exemplifies the Controller pattern with the MissingItemButton-CatalogController and the ReportMissingItemForm-CatalogController relationships. CatalogController handles the actions in this use case which are triggered by user input and then carrying out business logic within itself.

****

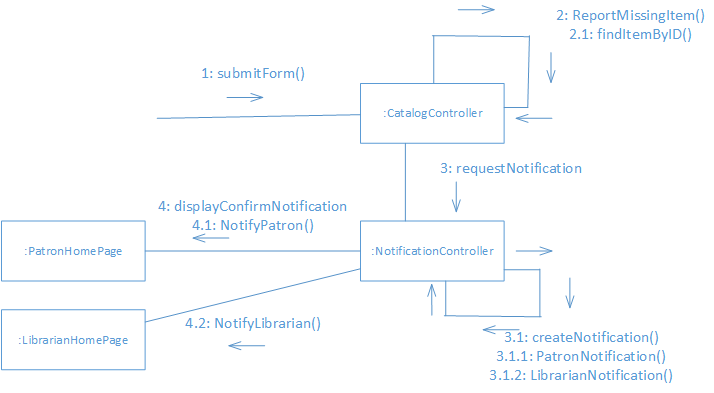
**13.2**

CatalogController again fulfills the Controller GRASP guideline. Once a MissingItemForm has been submitted, CatalogController verifies the information about an item in the Catalog based on the info in the MissingItemForm.

****

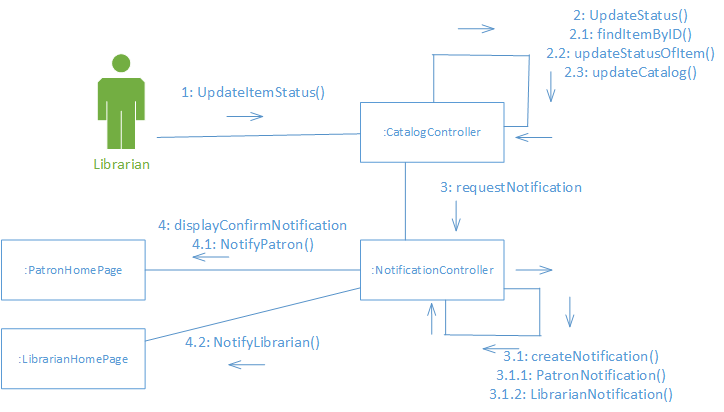
**13.3**

CatalogController again fulfills the Controller GRASP guideline. Once a MissingItemForm has been submitted, CatalogController handles updating an item in the Catalog based on the info in the MissingItemForm.

****

**13.4**

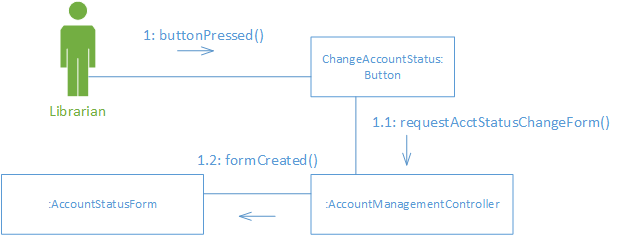
CatalogController again fulfills the Controller GRASP guideline. Once a form has been submitted then the CatalogController will find the item and update the item. Then the catalogController will request a notification from the NotificationController and send it to the LibrarianHomePage and PatronHomePage.



### Place Hold on Account

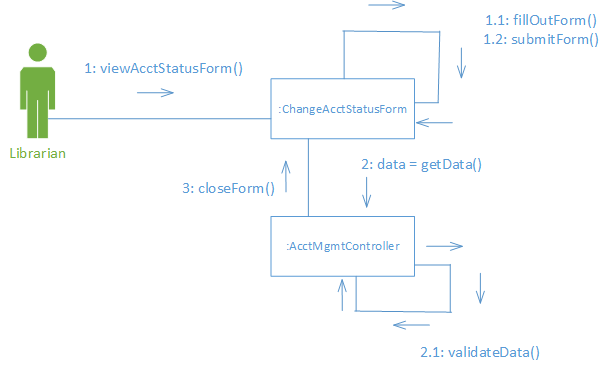
**14.1**

This collaboration graph exemplifies the Controller pattern with the ChangeAccountStatusButton-AccountManagementController and the AccountStatusForm- AccountManagementController relationships. AccountManagementController handles the actions in this use case which are triggered by user input and then carrying out business logic within itself.

****

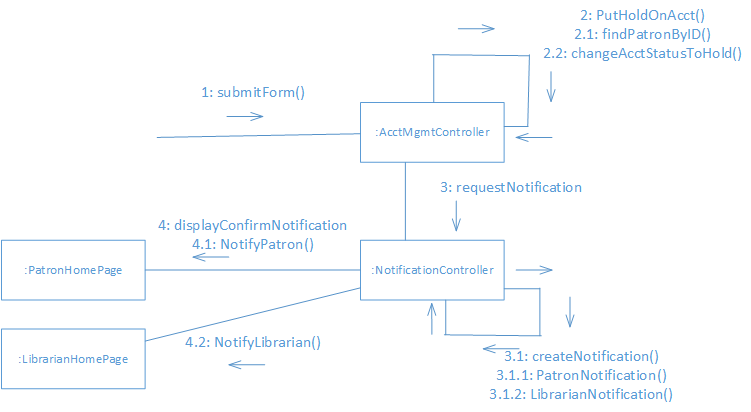
**14.2**

AccountManagementController again fulfills the Controller GRASP guideline. Once a ChangeAccountStatusForm has been submitted, AccountManagementController verifies the information about about an account based on the info in the ChangeAccountStatusForm.

****

**14.3**

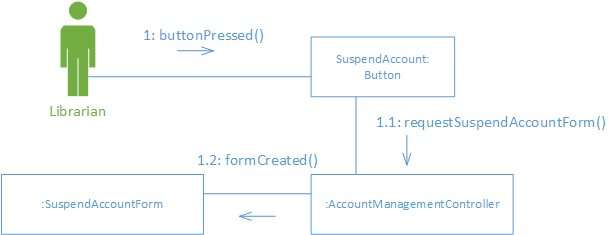
AccountManagementController again fulfills the Controller GRASP guideline. Once a ChangeAccountStatusForm has been submitted, AccountManagementController handles updating an account based on the info in the ChangeAccountStatusForm. The AccountManagementController will then send a request to the NotificationController which will send a notification to the PatronHomePage and the LibrarianHomePage.

****

### Suspend Account

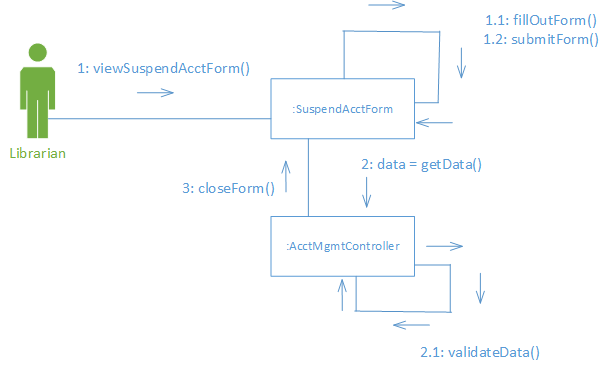
**15.1**

This collaboration graph exemplifies the Controller pattern with the SuspendAccountButton-AccountManagementController and the SuspendAccountForm- AccountManagementController relationships. AccountManagementController handles the actions in this use case which are triggered by user input and then carrying out business logic within itself.

****

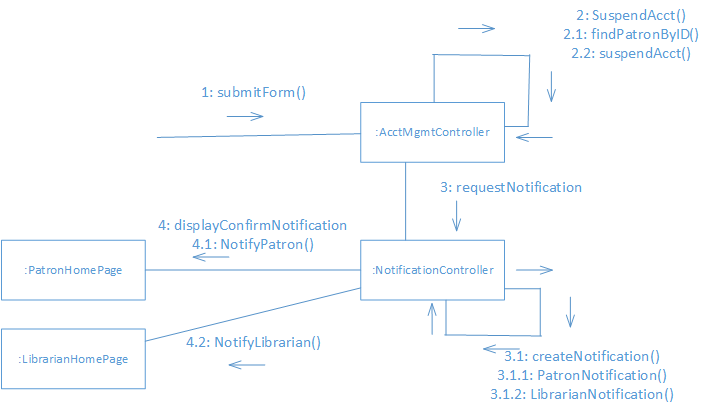
**15.2**

AccountManagementController again fulfills the Controller GRASP guideline. Once a SuspendAccountForm has been submitted, AccountManagementController verifies the information about about an account based on the info in the SuspendAccountForm.

****

**15.3**

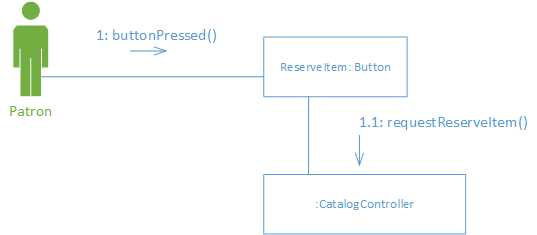
AccountManagementController again fulfills the Controller GRASP guideline. Once a SuspendAccountForm has been submitted, AccountManagementController handles updating an account based on the info in the SuspendAccountForm.

****

### Patron reserves an item

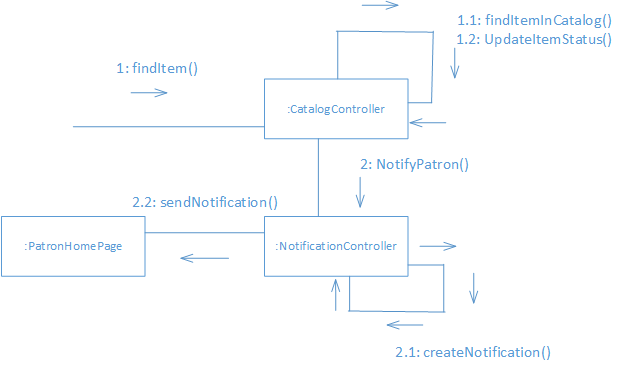
**16.1**

This collaboration graph exemplifies the Controller pattern with the ReserveItemButton-CatalogController relationship. CatalogController handles the actions in this use case which are triggered by user input and then carrying out business logic within itself.

****

**16.2**

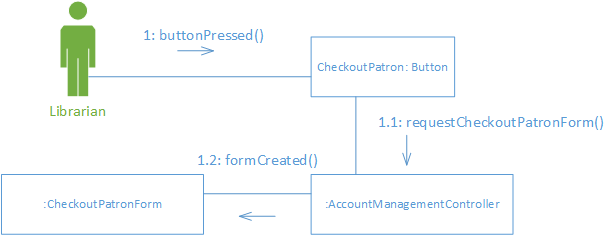
CatalogController again fulfills the Controller GRASP guideline. Once an item is found and the reserveItemButton has been pushed then the item to be reserved is found by the CatalogController and then the CatalogController tells the NotificationController to alert the patron.

****

### Patron check-out an item

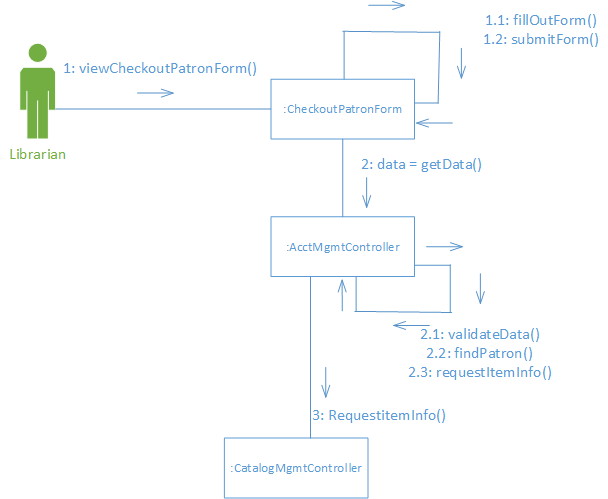
**17.1**

This collaboration graph exemplifies the Controller pattern with the CheckoutPatronButton-AccountManagementController and the CheckoutPatronForm- AccountManagementController relationships. AccountManagementController handles the actions in this use case which are triggered by user input and then carrying out business logic within itself.

****

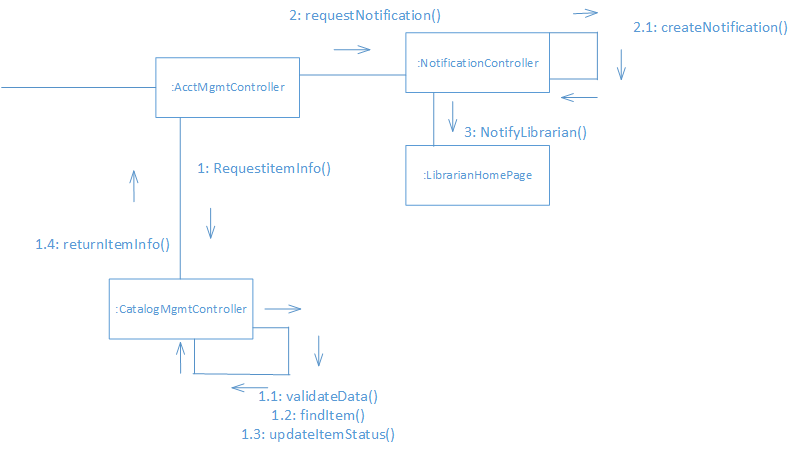
**17.2**

AccountManagementController again fulfills the Controller GRASP guideline. Once a CheckoutPatronForm has been submitted, AccountManagementController verifies the information about an account based on the info in the CheckoutPatronForm. Then the AccountManagementController will request information from the CatalogController on the item being checked out.

****

**17.3**

CatalogController again fulfills the Controller GRASP guideline. Once a CheckoutPatronForm has been submitted, CatalogController handles updating an account based on the info in the CheckoutPatronForm and attaches the item to the patron.



## 14. Design Class Diagram

This design class diagram uses a couple different design patterns. One design pattern used is the Adapter structural design pattern. This is shown when looking at the Account class. The account class can be adapted to fit a patron or a librarian entity object. Another structural design pattern used is the bridge pattern. This is shown with the CatalogManagementController because that one class will then call different boundary objects to show different windows depending on which action the user is trying to accomplish.

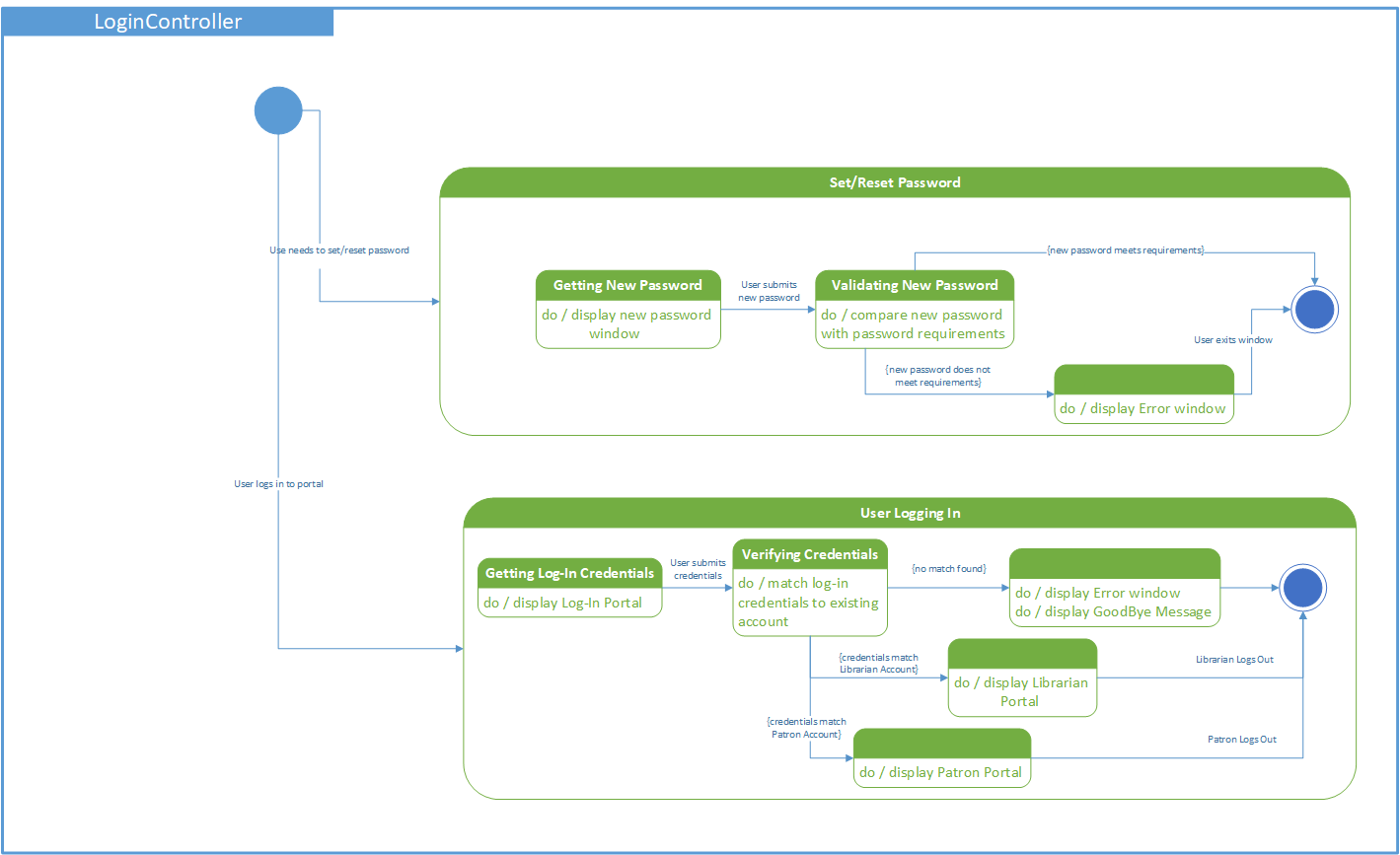
## 15. Class Design

### 1. Login

Context: Login

Invariants: self.patron.account.hasAccount(true) || self.librarian.account.hasAccount(true)

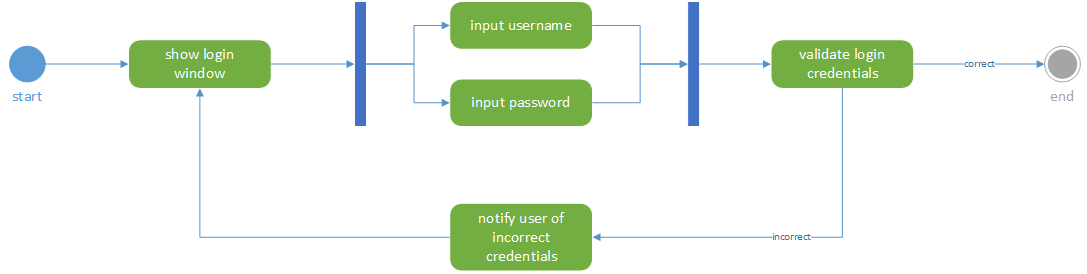
UML State Chart:



1.1 Login

Pre-condition: self.Account.hasUsername(true) && self.Account.hasPassword(true) && self.Account.isLoggedIn(false)

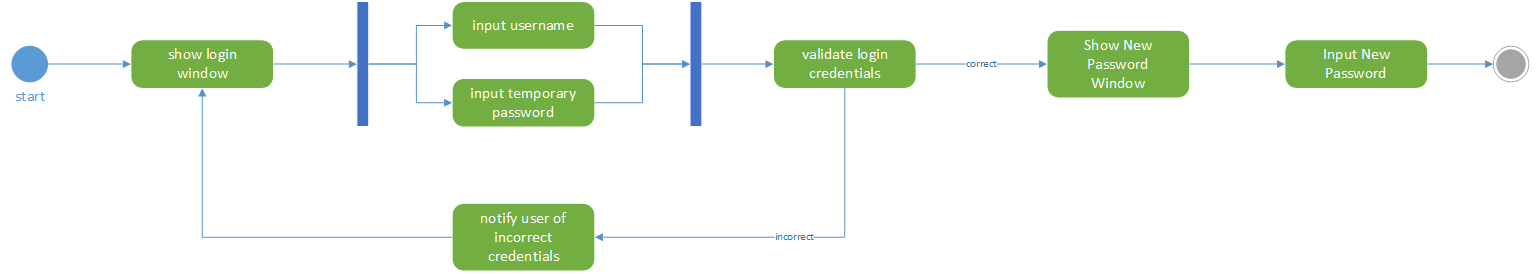
Post-condition: self.Account.isLoggedIn(true)

Activity Diagram:

1.2 setPassword

Pre-condition: self. Account.hasUsername(true) && self.Account.Password(temporaryPassword)

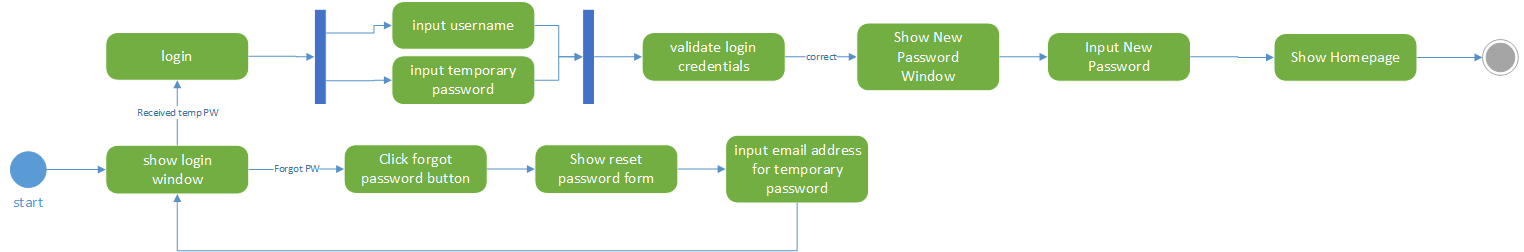
Post-condition: self.Account.Password(newPassword)

Activity Diagram:

1.3 resetPassword

Pre-condition: self.Account.hasUsername(true) && self.Account.Password().ForgotPassword()

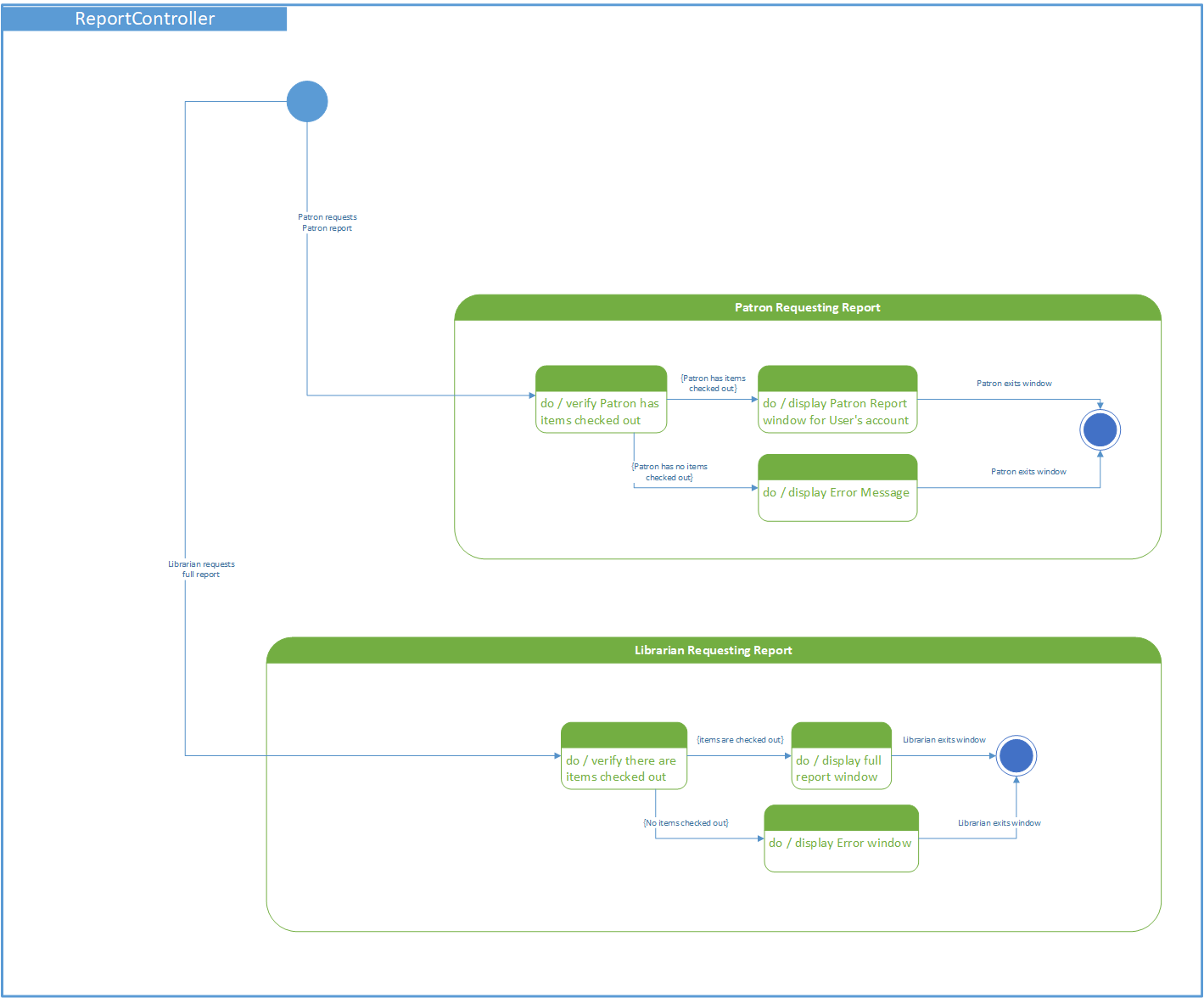
Post-condition: self.Account.Password(newPassword)

Activity Diagram:

### 2. Report

Invariants: Account.isLoggedIn(true)

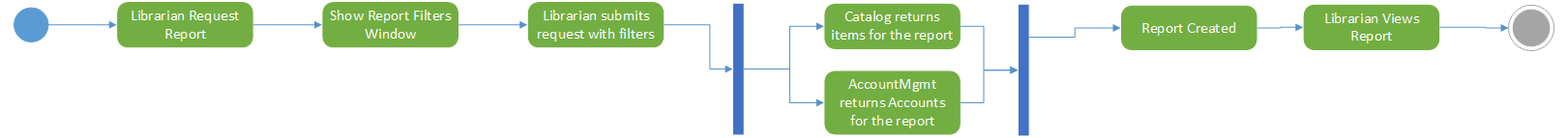
UML State Chart:



2.1 createLibrarianReport

Pre-condition: Catalog.ItemCount() >= 0

Post-condition: Report.Item.ItemCount() <= Catalog.Item.ItemCount()

Activity Diagram:

2.2 createPatronReport

Pre-condition: Account.Patron.Item.ItemCount() >= 0

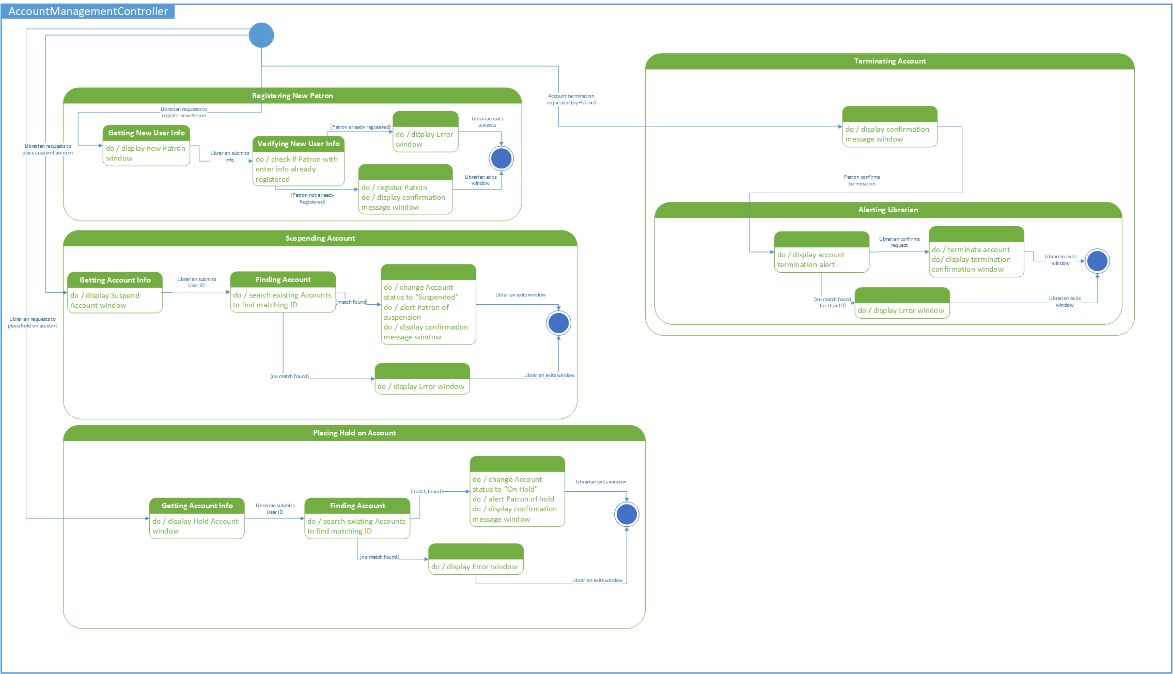
Post-condition: Report.Item.ItemCount() <= Account.Patron.Item.ItemCount()

Activity Diagram:

### 3. Account

Invariants: Librarian.Account.IsLoggedIn(true)

UML State Chart:



3.1 Add Account

Pre-condition: Patron.Account.AccountID == 0

Post Condition: Patron.Account.AccountID == accountID

Activity Diagram:

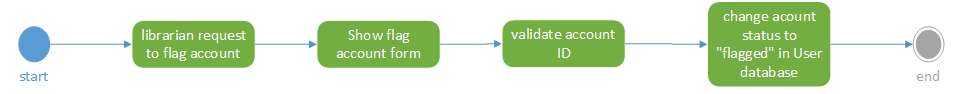


3.2 Flag Account

Pre-condition: Patron.Account.findAccountByID(accountID)

Post Condition: Patron.Account.accountStatus(flagged)

Activity Diagram:

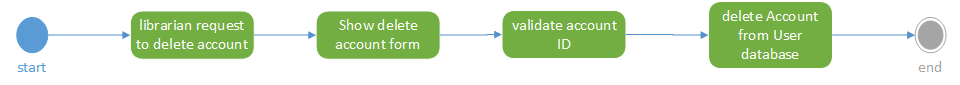


3.3 Remove Account

Pre-condition: Patron.Account.findAccountByID(accountID)

Post Condition: Patron.Account.findAccountByID(accountID) == Notification.createNotification(“Account Not Found”)

Activity Diagram:

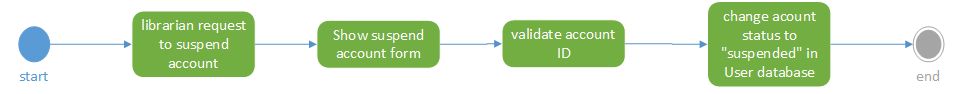


3.4. Suspend Account

Pre-condition: Patron.Account.findAccountByID(accountID)

Post Condition: Patron.Account.accountStatus(suspended)

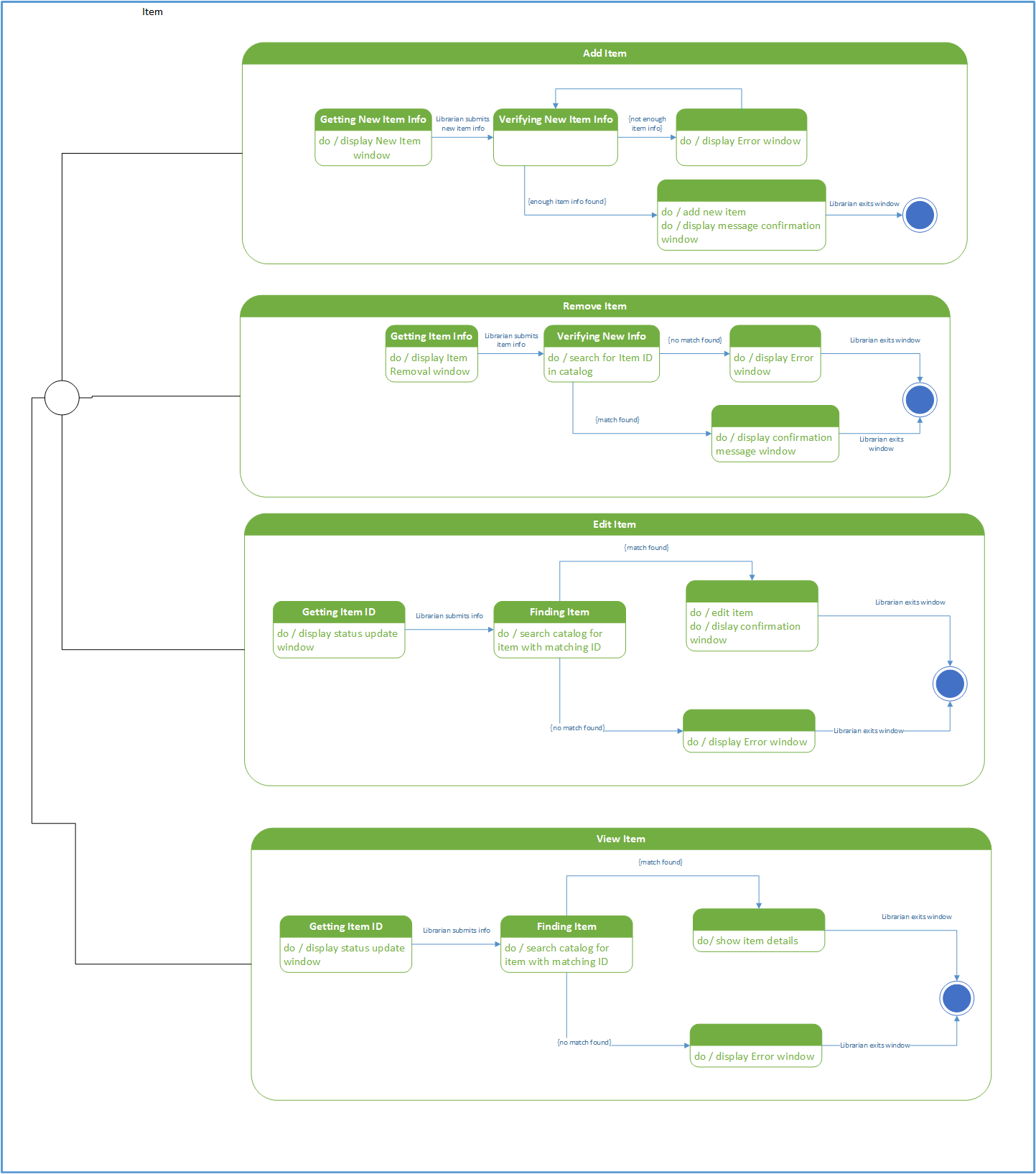
Activity Diagram:



### 4. Item

Invariants: Catalog.Item.ItemCount >= 0

UML State Chart:



4.1 Add Item

Pre-condition: Librarian.Account.LoggedIn(true)

Post Condition: Catalog.Item.ItemCount() > @pre Catalog.Item.ItemCount()

Activity Diagram:



4.2 Delete Item

Pre-condition: Librarian.Account.LoggedIn(true) && Catalog.Item.findItemByID(itemID)

Post Condition: Catalog.Item.ItemCount() < @pre Catalog.Item.ItemCount()

Activity Diagram:

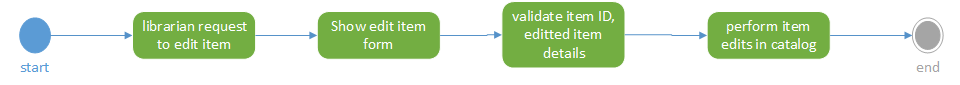


4.3 Edit Item

Pre-condition: Librarian.Account.LoggedIn(true) && Catalog.Item.findItemByID(itemID)

Post Condition: Catalog.Item.ItemUpdated(true)

Activity Diagram:



4.4 View Item Details

Pre-condition: Catalog.Item.findItemByID(itemID)

Post Condition: Catalog.Item.viewItemDetails(itemID)

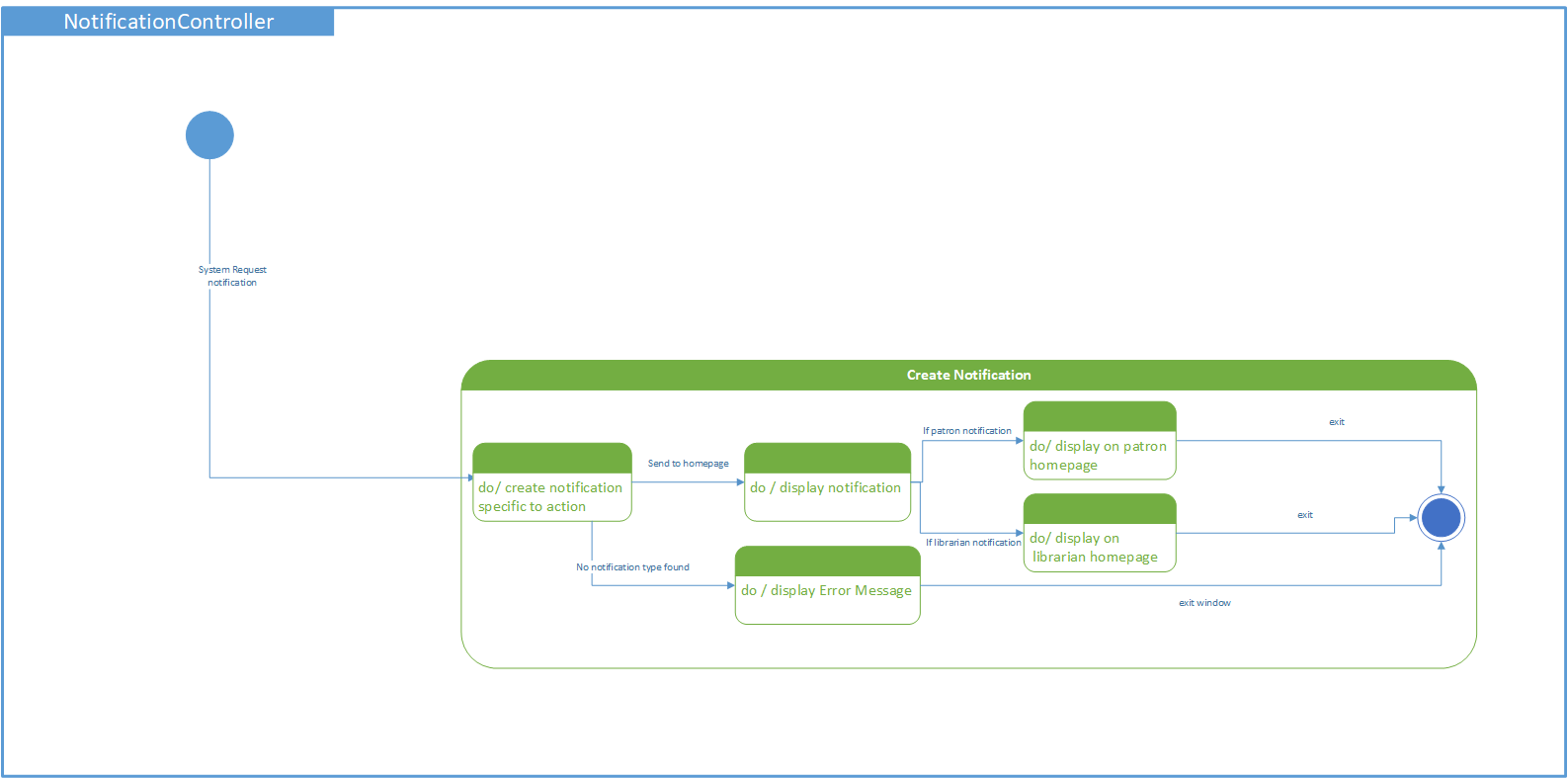
Activity Diagram:



### 5. Notification

Invariants: Notification.getNotificationRequests() >= 1 (There has to be one or more requests for a notification in order for one to be created)

UML State Chart:

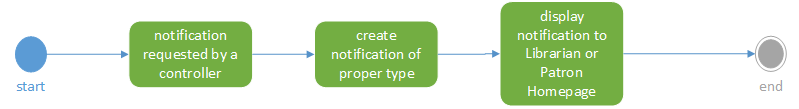


5.1 Create Notification

Pre-condition: CatalogManagementController.requestNotification() == 1 || AccountManagementController.requestNotification() == 1 || LoginController.requestNotification() == 1

Post Condition: CatalogManagementController.notificationSent(true) || AccountManagementController.notificationSent(true) || LoginController.notificationSent(true)

Activity Diagram:



## Design Quality

The quality, completeness, and consistency of the detailed design model for this Library Information System project can be evaluated by looking at the systems reliability, efficiency, maintainability, and usability. The key components of a quality system are measured by the system’s simplicity and modularity. After completing the architectural design of the Library Information System, it is easy to see how all of the pieces of the system fit together and what the key roles of each component in the system play. The UML deployment diagram is broken down into the main subsystems and their components. The MVC approach, Model-View-Controller, is used throughout all of the diagrams to make components easily identifiable. The client tier of the deployment diagram is the home of the web browser which the user will be using and also all of the views that the user may interact with. The Web tier is the home of the interface for the patron and for the librarian. The Application tier holds the controllers, models, and data access. This tier is the backbone of the entire application. The data tier holds all of the data such as all of the items in the library and all of the user accounts.

The design process became more specific after the deployment diagram. The system has 17 concrete use cases that are explained in great detail such as pre/post conditions, descriptions, summaries, actors, and a step by step process through the use cases. In order to show these use cases at a deeper level the detailed system sequence diagrams were taken to another level of detail and collaboration diagrams where created. For each use case there are three or more collaboration diagrams. The diagrams show a step by step process in which a use case takes place. Each collaboration diagram has a detailed paragraph stating all of the information that is being shown in the use case along with the GRASP patterns that are used. From the Detailed System Sequence Diagrams and the UML Collaboration Diagrams a Design Class Diagram began to take shape. This class diagram shows the entity, boundary, and controller objects throughout the Library Information System. The Design Class Diagram also shows multiple design patterns that were used and gives an explanation of how the design patterns fit into the Design Class Diagram. The diagram clearly shows the multiplicity of the objects as well as their relationships between the classes. There are a few classes that are dependent on other classes as well, which is shown.

From the Design Class Diagram five classes were selected that show state dependent behavior. The classes that were chose are the Login class, Account Class, Report Class, Notification class, and the Item class. These classes were chosen for not only their state dependent behavior, but because they are very complex classes that interact with multiple other classes. To show even more detail to each of the classes, the pre/post conditions for each class method are given along with the invariant for each class. The pre/post conditions and invariants are expressed in OCL statements. For each of the classes a UML State Chart Diagram was designed that shows how the methods of the class flow. The last component expressed for the individual classes is an a UML activity diagram for each of the class methods. These diagrams help to specify the control flow in terms of UML.

The detailed design model shows completeness because of the fact that you can see how the entire system will operate. You can see what views the patron of the library will interact with and which views the librarian will interact with. You can see the data models that will be passed back and forth from the controllers to the views and from the views to the controllers. You can see the controllers and which methods they will implement in order to allow a user to complete the requirements of the system. Not only are these things shown in one model for viewing, but they are shown in multiple models and are explained in each model to a different level of detail. The detailed design model shows consistency because a specific method or action can be traced from the bigger less detailed models all the way to the more detailed class specific models. An example of this traceability is shown with login. Login is explained into great detail in the class design, the same class can be found on the Design Class Diagrams, Use Cases, System Sequence Diagram, Detailed System Sequence Diagrams, and the Collaboration Diagrams. This traceability helps to make the diagrams readable and less confusing to programmers who need to program the system and a business professional who may be looking to get the system implemented.

The quality of the detailed design model is demonstrated by multiple factors. The design models show an average of four methods per class which are mostly CRUD methods. CRUD methods are methods for creating, reading, updating, and deleting an object. The class diagram shows a few classes with inheritance. The depth of these classes is minimal which in return shows a relationship of high cohesion and low coupling, a software design goal. High cohesion and low coupling is also shown because objects in the Design Class diagram do not have a large number of dependencies. The classes are centered around doing one job and doing it well. Not only is high cohesion and low coupling a goal when designing software, but it will make the software produced after the design is done easier to maintain. The design models are finally of quality because they show simplicity. Complex methods are difficult to maintain, make it harder to understand the application, make the application less reliable, and also are harder to test. The methods that are being implemented have one or two parameters, which make them easy to maintain.

Overall the design quality of the detailed design model is good. This is proved by all of the factors mentioned in the above paragraphs. There, of course, is always room for improvement. Other software engineers may see things a little differently, therefore, implement them differently. The diagrams seem to be consistent and complete with the guidelines that were discussed in class.