**SE 452 Group 3 Analysis Document**

November 7, 2012

**Table of Contents**

[Overview 3](#_Toc340071209)

[Requirements 3](#_Toc340071210)

[Design 4](#_Toc340071211)

[Persistence Layer (Table Layout): 4](#_Toc340071212)

[Application Design 5](#_Toc340071213)

[Discussion of lessons learned 7](#_Toc340071214)

[Appendix 7](#_Toc340071215)

[Decisions log 7](#_Toc340071216)

[Work break down 8](#_Toc340071217)

# Overview

For the problem domain, we have chosen to implement a banking application that will allow users

to create, access, and manipulate various types of accounts. As illustrated by the design model, the primary users of our application will be account holders represented as AccountHolder objects. AccountHolder objects contain information related to the customer. Account holders can be either Primary or Dependent customers, both of which can access their assigned account. Additionally, account holders can open multiple accounts.

The core of our application is represented by both the Account holders, and Accounts. Account objects are affected by the operations performed by transaction entities (Credits and Debits). The AccountHolder generates the transaction entities that affect Account objects through interaction with the application. Account holders can also manage their various accounts through interaction with the application.

As illustrated in the design model, persistence for our application revolves around the logging of various domain classes into the database upon creation.

We utilized the Spring MVC framework to provide separation between the model, view, and controller aspects of our application. We chose Spring MVC for many of its features, including dependency injection, which made testing controller logic easier. Binding directly to domain objects, utilizing interfaces, and integrating with JSP made Spring the favorable choice for our development.

We also utilized Hibernate to implement our persistence layer. We chose this solution for various reasons, but mostly because of the flexibility and extensibility offered by Hibernate. Being able to store and retrieve our domain objects straight to the database without having to store and retrieve the individual attributes definitely made development much more straight-forward. Additionally, our database schema changed a time or two during development, and utilizing Hibernate allowed for easy modification by simply requiring changes in our Hibernate configuration file, and the various mapping files.

# Requirements

The following use case descriptions describe the various interactions the User will have with the application.

Use Case Descriptions:

Add Account Holder – this is essentially user creation. In order to interact with the application the user must register as an account holder. This interaction involves entering personal information needed by the application to register a user as an account holder.

Create a Login – the user is prompted to enter in their choice of login name and password. This information is used to authenticate a user and allow them access to the various parts of the application.

User Login – the user is prompted for their login name and password. The system will authenticate this information. If the information presented by the user cannot be authenticated, an error message indicating an invalid login name or password was presented.

Add Account – the user is prompted to create an account. The user has a choice to open a Savings, Checking, or Loan account. Currently, the user has the option to select an opening balance and an initial interest rate to open the account with.

View Account – the user is presented with all the accounts and account information for the accounts they have opened.

Close Account – the user has the option to close any existing account associated to them. The user is prompted to select an account to close. Once the user makes a selection, that account is “closed” and removed from the accounts they have opened.

Link Other Bank Account – the user has the option to link an account from an outside banking institution to their user account in the system. The link is created utilizing the bank routing number and account number for the account from the outside banking institution.

Edit Personal Information – the user is prompted to edit their personal information currently stored in the system.

View Debit History – the user has the option to view the debit transaction history for their accounts. The user is presented with a table detailing debit transactions for their account.

View Credit History – the user has the option to view the credit transaction history for their accounts. The user is presented with a table detailing credit transaction for their account.

Add Debit – for testing and administration purposes, debits can be added to accounts to update account balances.

Add Credit – for testing and administration purposes, credits can be added to accounts to update account balances.

\*\*\*See use case model for visual documentation of use case scenarios

The application is designed to allow an administrator to create an account for a banking customer. It allows them to generate debits and credits on their accounts. In addition it allows a user to login with a username and password and view their account information. It also allows the user to edit their personal information and to upload bank account information from other financial institutions.

# Design

## Persistence Layer (Table Layout):

Account\_Holder table – stores instances of the AccountHolder class. Holds information pertinent to customer records

Account table – stores instances of the Account class. Stores information pertinent to bank account records

Holder\_To\_Account table – used to link AccountHolder objects to Account objects using a user id and account id foreign key

Debit table – used to store Debit transaction objects. Stores information pertinent to debit transactions related to an account

Credit table – used to store Credit transaction objects. Stores information pertinent to credit transactions related to an account

Account\_Holder\_Login table – used to store login information for account holders. Information in this account is used to authenticate users with user accounts in the system.

Other\_Bank\_Account table – used to store information pertinent to linking bank accounts from other banking institutions

Check\_Debit table – stores information that details check debit transactions associated with an account. Check\_Debit transactions are linked to Debit objects using a debit id foreign key

Bill\_Pay\_Debit table – stores information that details bill pay debit transactions associated with an account. Bill\_Pay\_Debit transactions are linked to Debit objects using a debit id foreign key

Debit\_Card\_Transactions table – stores information that details debit card transactions associated with an account. Debit\_Card\_Transactions are linked to Debit objects using a debit id foreign key

\*\*\*See Database Schema for detailed visual documentation of persistence table layout

## Application Design

Package Structure:

Our application source code is separated into distinct packages that represent class responsibility in system execution. Our package structure is as follows:

Form package – packages the domain classes used in the application

DAO package – packages the data access object interfaces and implementation classes used in conjunction with service classes to persist data to the database

Service package – packages the service interfaces and implementation classes that data access objects use to persist data to the database

Controller package – packages the controller classes used to implement business logic, and system navigation for the application

Domain Classes (Model):

AccountHolder – the users of the application. AccountHolder instances are essentially user instances that have various interactions with the application.

Account – Account instances represent the various accounts that account holders can create/open and manage. Accounts are linked to the account holders that create them.

AccountHolderToAccount – instances of this class are used as “linkers” that link account holders to the accounts that they open.

OtherBankAccount – instances of this class represent accounts that originate from outside banking institutions and are linked into the application

Debit – instances of this class represent debit transactions that debit the balance of an associated account. Debit instances are associated with an account via account id.

Credit – instances of this class represent credit transactions that credit the balance of an associated account.

Credit instances are associated with an account via account id.

Data Access Classes (Model):

AccountHolderDAO – interface that defines methods used to persist AccountHolder instances to the database

AccountDAO – interface that defines methods used to persist Account instances to the database

CreditDAO – interface that defines methods used to persist Credit instances to the database

DebitDAO – interface that defines methods used to persist Debit instances to the database

OtherBankAccountDAO – interface that defines methods used to persist OtherBankAccount instances to the database

AccountHolderService – interface used by AccountHolderDAO to persist data to the database

AccountService – interface used by AccountDAO to persist data to the database

CreditService – interface used by CreditDAO to persist data to the database

DebitService – interface used by DebitDAO to persist data to the database

Controller Classes (Controller):

AccountHolderController – class is used to implement business logic and system navigation associated with account holder specific system interaction

AccountController – class is used to implement business logic and system navigation associated with account specific system interaction

CreditController - class is used to implement business logic and system navigation associated with credit specific system interaction

DebitController - class is used to implement business logic and system navigation associated with debit specific system interaction

OtherBankAccountController - class is used to implement business logic and system navigation associated with other bank account specific system interaction

LoginController – class is used to implement login and authentication functionality

UserHomeController – class is used to implement user home page navigation and functionality

JSPs (View):

accountCreateAdd – page used to create/open an account

accountDelete – page used to delete/close an account

showAccountList – page used to present account information to the user

addAccountHolder – page used to create a “user account”

addOtherBankAccount – page used to link a banking account from another banking institution

createLoginCredentials – page used to create login name and password

editPersonalInformation – page used to edit user/account holder information

addDebit – (administrative page) used to add debits to an account held by an account holder

getDebitHistory – page used to present debit transaction history to the user

addCredit – (administrative page) used to add credits to an account held by an account holder

getCreditHistory – page used to present credit transaction history to the user

login – page used to authenticate login credentials for a user attempting to access his/her user home page

userHome – page represents the home page for a user after said user has logged in to the system. User home essentially represents the gateway into the application

## 

# Discussion of lessons learned

This project has proven to be a little challenging for various reasons. But it has also proven to be a solid learning experience. Learning a new framework (Spring MVC) and implementing it into the application offered its own set of challenges. Configuration of the components and environment necessary to “wire” and “glue” everything together proved to be the most difficult task of implementing Spring into development. The benefits of object relational mapping (ORM) solutions were also made much clearer through the development of this project. Without the use of such technology, it is evident that development would have been more difficult and much less straightforward. Additionally, the challenges of online group collaboration provided a good learning experience outside of writing code. Group organization, and communication were pivotal aspects necessary to complete this project on schedule, and we believe we have succeeded in that regard.

Zachary Wilson:

The biggest lessons learned for me is that is painful to pick up a new framework and implement it cleanly. There were many times that I rewrote the same code to make it conform to accepted standards for that framework. There are still many things that I would like to redesign to allow for concurrent development to be cleaner. For example the bean definition file should be split up for each java object instead of being lumped together into one xml file. Also I realized too late that I could make the compiled code/resources point to particular directories in the production environment and this caused the creation of two identical bean definition files to be created for test and for production. I will admit though, that once I started to grasp the Spring concepts, I was surprised how easy it was to develop things, especially on the front end. I could develop the jsp pages without knowing a single thing about java and I could do it quickly.

# Appendix

## Decisions log

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Date | Problem | What was decided | Alternatives considered | Rationale |
| 9/19/12 | To decide on development environment. | **IDE:**  Eclipse  **Core Framework:**  Java 7  **Web framework:**  Spring  **Database:**  HSQL  **Code sharing:**  Google drive | **Database:**  MySQL  **Code sharing:**  Apache Subversion  Dropbox  CVS (Concurrent Versions System)  Google Drive  **Web framework:**  Spring  Apache Struts  Eclipse RAP  GRAILS  JBOSS SEAM  Tapestry | GRAILS would require additional time to learn the technology. The group was comfortable with the Java platform. |
| 9/26/12 | To decide on persistence framework. | **Database Frameworks:** Hibernate  JDBC | **Database Frameworks:**  RimuDB  JPA  Apache Cayenne  Ibatis | The group wanted to use Hibernate for learning purposes. |
| 10/8/12 | Application architecture created | Use Spring MVC/Hibernate architecture |  | This architecture leveraged Spring and Hibernate frameworks. |
|  |  |  |  |  |

## Work break down

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Application Area | Problem | What was decided | Alternatives considered | Rationale |
| Buildmaster | To merge code from all group members. | Buildmaster:  Zach | Buildmaster:  Steve | Zach seems to have the most Java experience, so putting him in this role would allow for additional code reviewing by a more experienced team member. |
| Developer | To develop a banking application. | Developer:  Zach  Corey  Steve  Walter | No other alternatives considered. | All group members should contribute to building the application. |
| Project Coordinator | To coordinate documentation, submissions, logistics, and meetings. | Project Coordinator:  Steve | Project Coordinator:  Zach | Steve is the in-class member of the group who is presenting the final product to the class. He should be involved in all aspects of the project so as to convey the information to the class during the presentation. |
| Database Design | To develop a database schema and associated documentation. | Zach |  | Zach was the most well-versed in database design. |
| Other Design Documents | To provide supporting documentation explaining our application design. | Corey |  | Corey wanted to take on the design documents. |