Shark Byte Financial Suite

Analysis and Design Phase

COMP 3700 - Yilmaz

Jeremy Roberts

Will Hendrix

Lawrence Smith

Chase Townson

Table of Contents

PHASE I - ANALYSIS

Domain Analysis

Concept Statement

Shark Byte is a collection of financial tools that enables users to take control of their financial status by offering an easy-to-read interface that anyone can understand. Users will be able to quickly see exactly where their money is being spent and how much is being saved. By providing users with simple tools, we hope to encourage those who are daunted by trying to calculate their budgets and investments to keep track of their spending. With an easy to use yet powerful toolset, users will be able to keep up with their finances as take control of their spending, saving, and investing habits.

On first startup, the user is asked to make a user account. A user account is made of a username and a password. After one user exists, additional user accounts can be made from the login menu. These users cannot see each other's information. These accounts let a user log into a personal finance application. Once one user account exists, the application always starts with a login screen. Each user account will keep track of the username, password, transactions, budgets, goals, investments, bank accounts, and transaction labels. A database will store each user account and all of its information.

The app has five tabs for transactions, budget, bank accounts, investments, and the home screen. Each tab provides an interface to interact with each topic along with charts, graphs, and readouts of all relevant information. The home screen tab the app displays the user's most current budget graph, investment value graph, and a summary of the most important information from each section. The home screen also provides a link to an account management page, where the user can update their password or delete their account.

The transaction tab leads the user to a page with a list of all transactions in chronological order. These transactions include income and money spent. The top of the transactions list has an input box. The input will ask for a transaction date, amount, bank account, and label. The app has some generic labels, and inside the label dropdown, the user can add new labels or remove one from the list. The added transaction then applies itself to the bank account and is saved to the repository where other parts of the application can see it. In addition to being able to create transactions manually, the user will be able to set up recurring transactions. These transactions have an associated interval describing the rate at which the transactions apply, and the recurring transaction has either a set number of repetitions or is perpetual. The list of transactions will show every transaction in chronological order, and will load a small portion of the list initially. As the user scrolls down the list, it will load more transactions, until the end of the list is reached. Further, the user can filter the list by date, label, amount, and bank account.

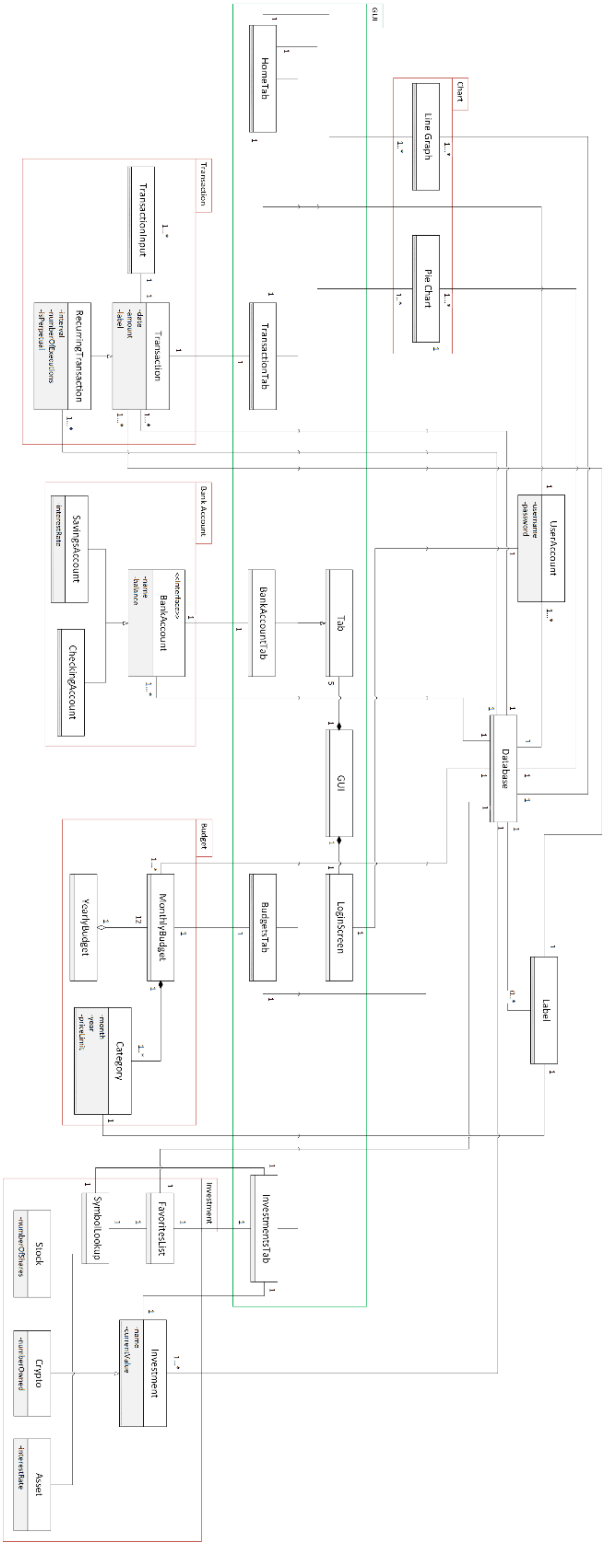
The primary feature of Shark Byte is the budgeting tool it provides. The budgeting tools will be consolidated into the "Budgeting" tab. These budgeting features are primarily concerned with transactions related to expenditures made by the user as well as transactions that increase the user's income. Budgets will be created on a monthly basis, and these monthly budgets can be aggregated into a yearly budget, showing the monthly budgets over a given year. Users of the budgeting tools will be interested in setting, maintaining, and reaching their spending goals. In order to make use of the budgeting tools, the user will need to create budget categories and

set price limits to these different budget categories, which will represent the maximum amount the user is willing to spend on goods and services belonging to that category. These budget categories represent different types of expenditures the user may makes such as utilities, food, entertainment, etc. A category's name must exist within the list of transaction labels. Spending transactions belonging to a given budget category will be totaled, and using these totals and the price limits associated with each budget category, Shark Byte will display two pie charts to the user; the first contains the percentage of the user's actual spending on different categories while the second contains the percentage of the user's planned spending on different categories. The tab will also show comparisons in numbers so users can see in detail how their spending stacks up against their budget. From each category in the list, the user will be able to see a list of transactions with that label that happened in that month. When the user can see exactly which transactions pertain to a budget category, they will be able to instantly identify how their habits affect their budget.

The Bank Accounts Tab will allow users to view and manage their bank accounts. Bank accounts can be either checking or savings accounts. Bank accounts all have a name and balance, and savings accounts have an interest rate. Shark Byte will include three default savings accounts: General, Retirement, and Emergency. The user will be able to add, remove and rename bank accounts. The Bank Accounts tab will include line graphs for seeing changes in value over time. Also, each bank account will have an associated list of transactions for the user to quickly see how money moves in and out of each bank account.

Shark Byte also includes a section dedicated to tracking investments. Investment options include stocks and cryptocurrencies. In addition, users will be able to enter custom assets with optional interest rates for keeping track of valuable things like homes or art. The investments tab will feature a summary on its main screen, complete with charts for quick readability. Further, detailed readouts will show changes in value over time for each asset. It will also feature a "favorites list" to track both owned and unowned stocks and cryptocurrencies. A search function will also be available to allow users to quickly look up stocks and cryptocurrencies. Stocks and cryptocurrencies are distinguished by their symbol, and each of them has a price. The application will allow users to keep track of how much of each they own. Prices in the application will be real-time, provided by an external API. Charts will be available to show changes in investments over hours, days, weeks, months, years, and all time. The investments portion is designed to empower both short-term and long-term investors.

Conceptual Domain Model



Domain Class Model OCL Constraints

```
Context UserAccount
        inv: self.username.size() < 10
```

```
Context UserAccount
        inv: self.username -> forall (<x,y> | x != y)
```

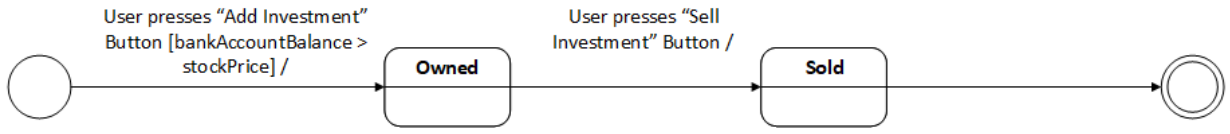
```
Context SavingsAccount
        inv: self.interestRate >= 0
```

```
Context Transaction
        Inv: self.amount != 0
```

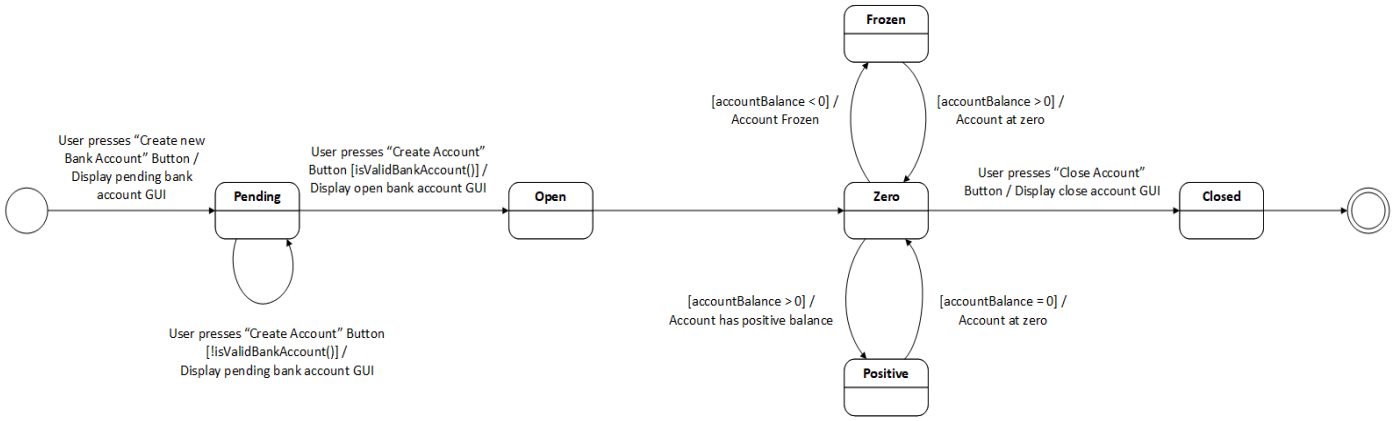
```
Context Stock
        inv: self.numberOfShares > 0
```

Domain State Model

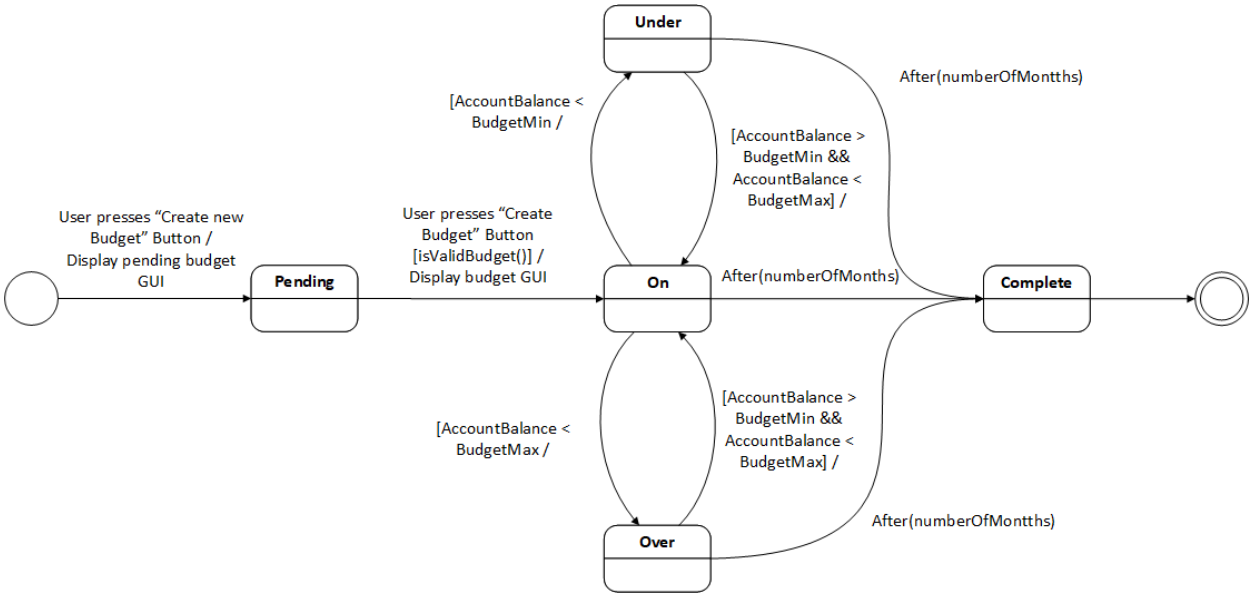
Investment



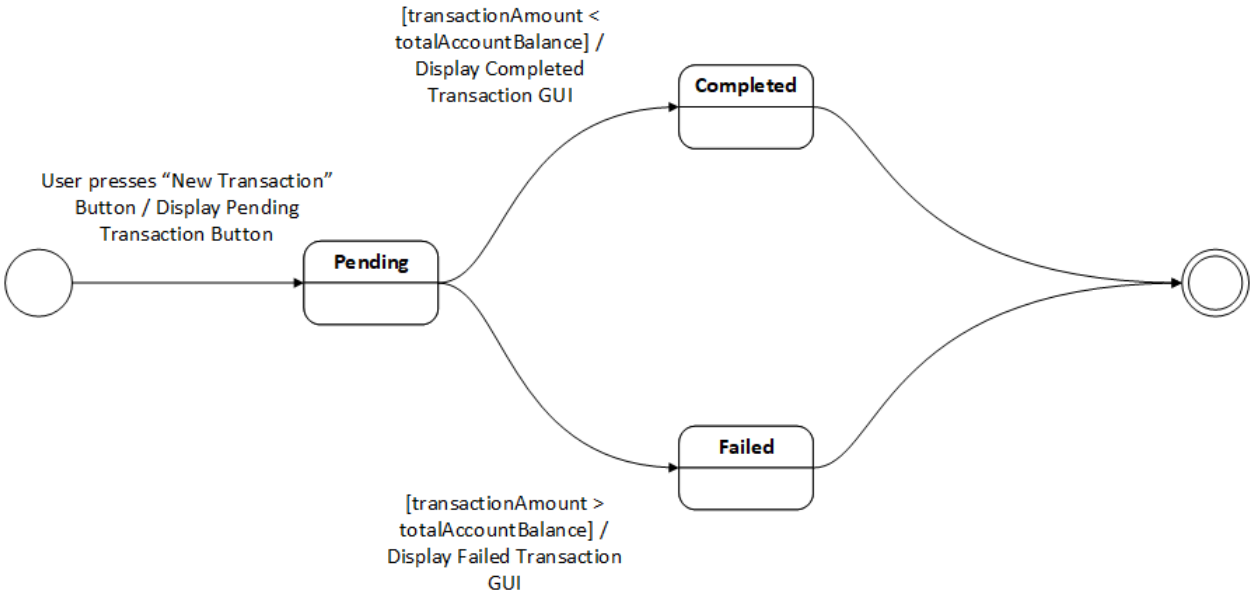
Bank Account



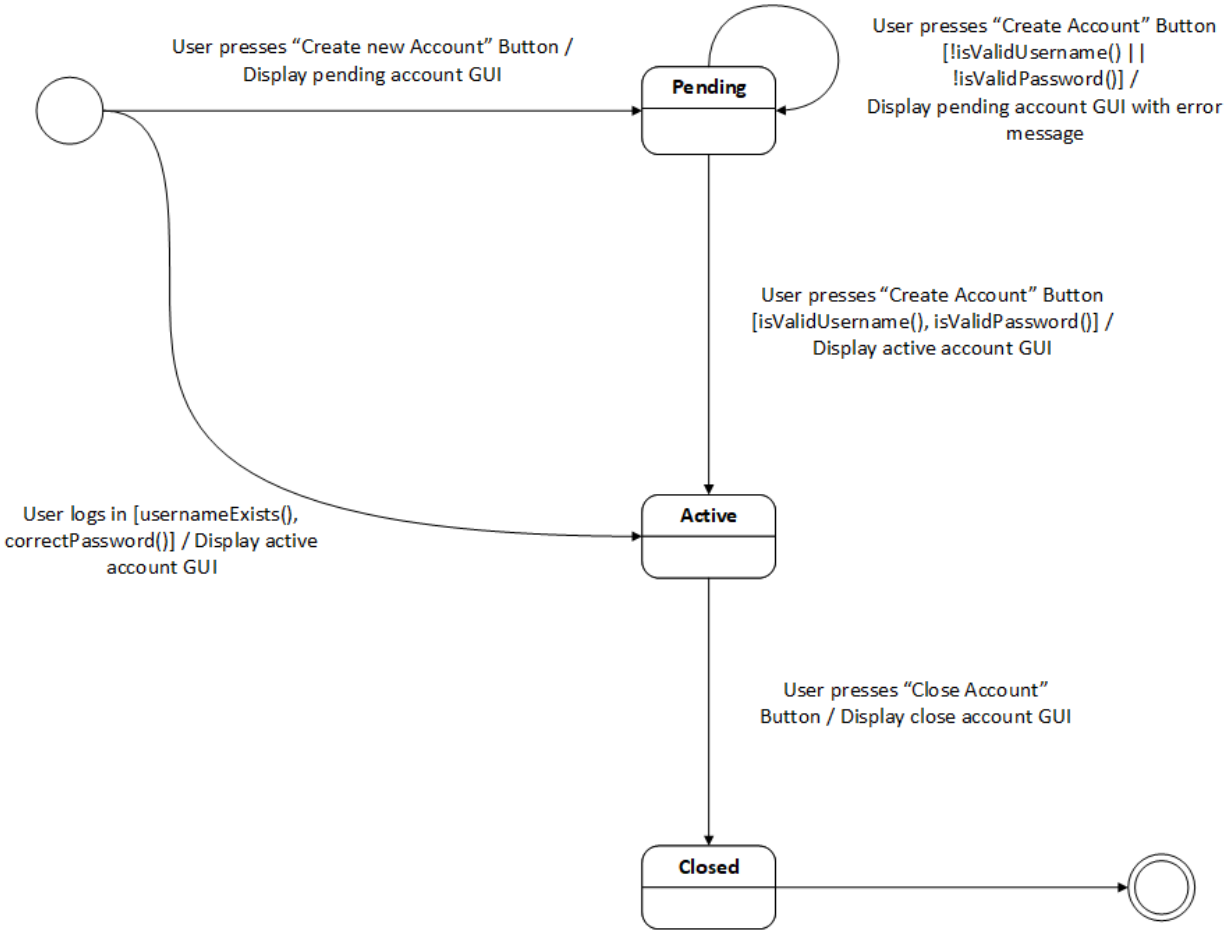
Budget



Transaction

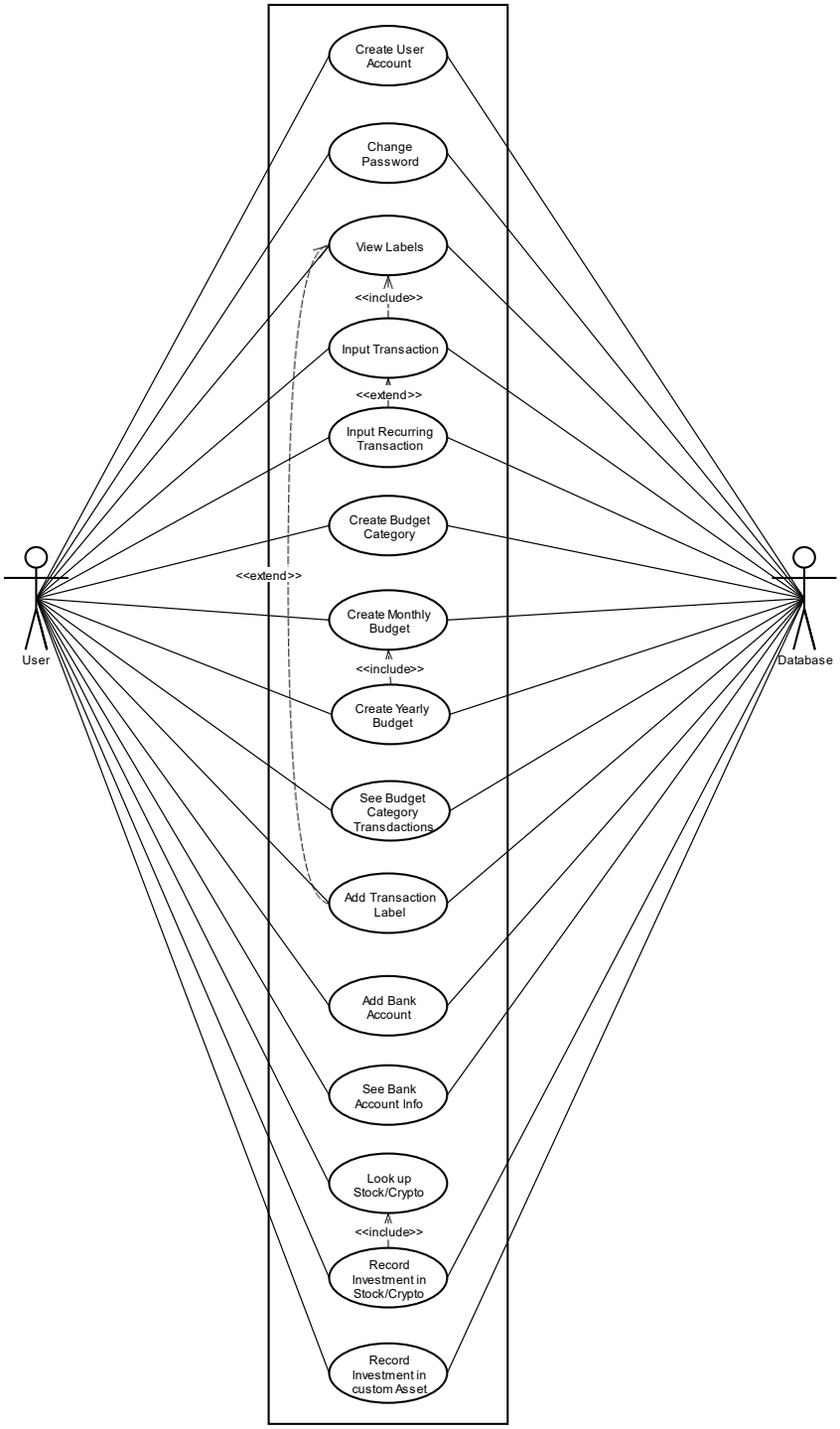


User Account



Application Analysis

Application Interaction Model



Essential Use Cases, Scenarios, HLSSD:

Use Case: Create User Account
Participants: User, Database
Precondition: User is on the login screen
Flow of Events:

Actor Intentions	System Responsibility
1) User selects create account 3) User enters information	2) System prompts user to enter username and password. 4) System creates a user account in the database.

Alternative Flow of Events:
Step 4: User entered invalid information, and the system returns to step 2.

Postcondition: The account has been created and the user is redirected to the login page.

Scenarios:

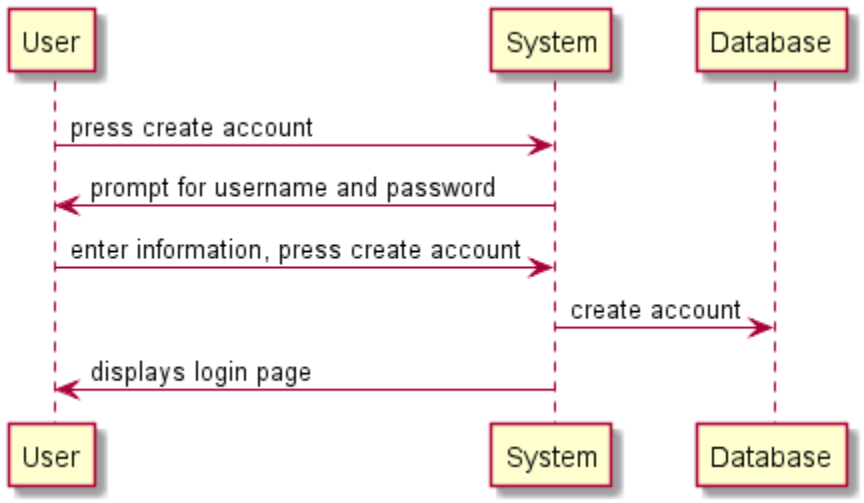
User selects "create account" from the login page.

System displays a prompt for a username and password.

User enters a username and password and clicks "create accounts".

System enters the information into a database entry.

System directs user to the login page.



User selects "create account" from the login page.

System displays a prompt for a username and password.

User enters a username and an invalid password and clicks "create accounts".

System sees that the password doesn't meet the requirements.

System displays "invalid password"

System displays a prompt for a username and password.

User enters a username and password and clicks "create accounts".

System enters the information into a database entry.

System directs user to the login page.

Use Case: Change Password
Participants: User, Database
Precondition: User is on the home screen
Flow of Events:

Actor Intentions	System Responsibility
1) User selects manage account. 3) User selects change password. 5) User enters information.	2) System prompts user with account management screen. 4) System prompts user to enter old password and new password. 6) System updates the database with new password.

Alternative Flow of Events:
Step 6: User incorrectly enters old password. System returns to step 4.

Postcondition: The new password is reflected in the database. User is returned to home screen.

User selects "manage account".

System displays the account management screen.

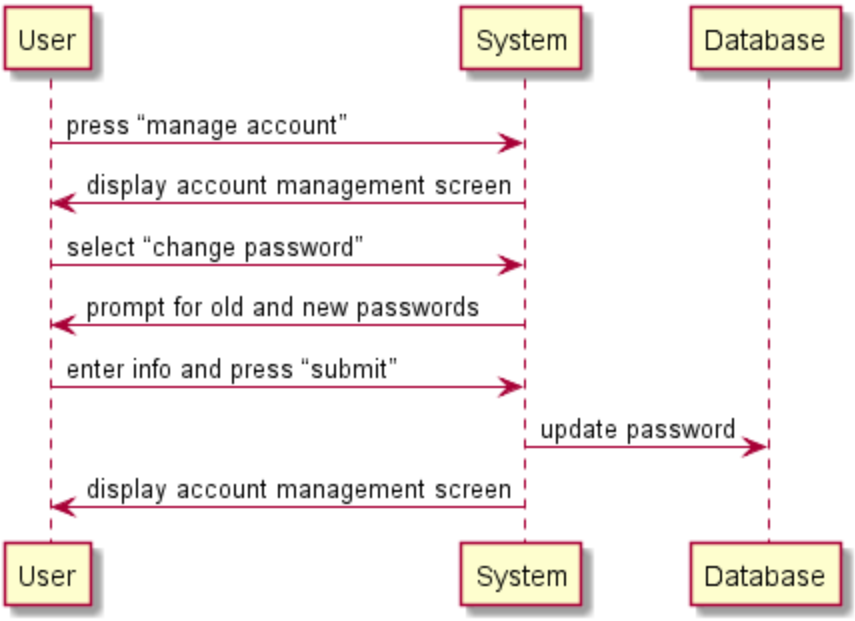
User selects "change password".

System displays a prompt for the old password and a new password.

User enters the information and presses "submit".

System updates the account's entry in the database.

System displays the account management screen.



User selects "manage account".

System displays the account management screen.

User selects "change password".

System displays a prompt for the old password and a new password.

User enters the information and presses "submit".

System sees that the password doesn't meet the requirements.

System displays "invalid password"

System displays a prompt for the old password and a new password.

System updates the account's entry in the database.

System displays the account management screen.

Use Case: View Labels
Participants: User, Database
Precondition: User is in the Transaction Tab
Flow of Events:

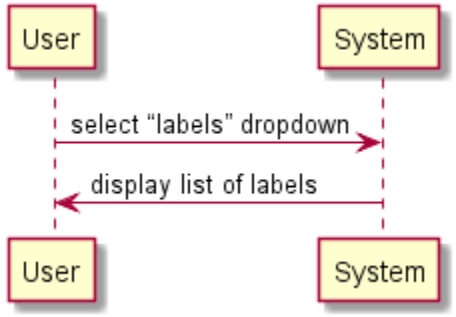
Actor Intentions	System Responsibility
1) User selects the label dropdown. 3) Database returns list of labels	2) System queries database for transaction labels. 4) System shows list of labels.

Alternative Flow of Events:
Step 3: Database finds no labels. System shows an empty list with a prompt to enter a new label.

Postcondition: The list of labels is visible.
User selects the "labels" dropdown.

System queries database for transaction labels.

System displays a list of the labels.



User selects the "labels" dropdown.

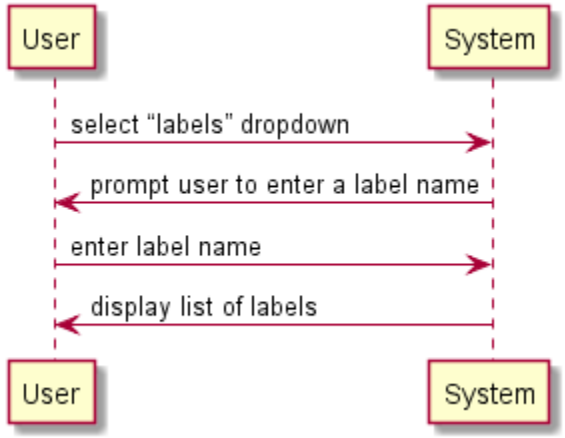
System queries database for transaction labels.

System sees an empty list, and prompts the user to enter a new label.

User enters a label name.

System adds the label to the database.

System displays the list of labels.



Use Case: Input Transaction
Participants: User, Database
Precondition: User is in the Transaction Tab
Flow of Events:

Actor Intentions	System Responsibility
1) User enters transaction information in the new transaction row.	3) System displays labels.

2) User selects the labels dropdown. 4) User selects a label. 5) User submits transaction.	6) System adds the transaction to database and displays the updated list.

Alternative Flow of Events:
Step 2: User incorrectly input information. System alerts user.

Postcondition: The transaction has been added to the database and is shown at the correct location in the transaction list.

User enters transaction information in the new transaction row.

User selects the "labels" dropdown.

System displays a list of the labels.

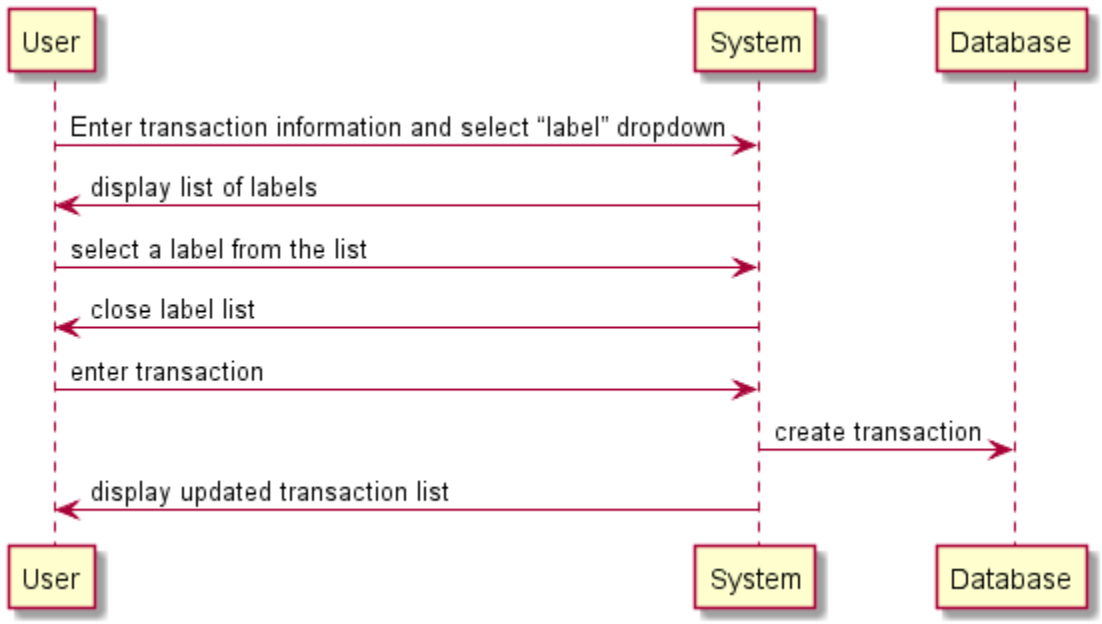
User selects a label from the list.

System closes the label list.

User submits the transaction by pressing enter.

System adds the transaction to the database.

System displays updated transaction list.



User enters transaction information in the new transaction row.

User selects the "labels" dropdown.

System displays a list of the labels.

User selects a label from the list.

System closes the label list.

User submits the transaction by pressing enter.

System identifies an invalid transaction amount.

System highlights the incorrect entry.

User updates the transaction amount.

User submits the transaction by pressing enter.

System adds the transaction to the database.

System displays the transaction in the correct spot in the transaction list.

Use Case: Input Recurring Transaction
Participants: User, Database
Precondition: User is in the Transactions Tab
Flow of Events:

Actor Intentions	System Responsibility
1) User enters transaction in the new transaction row, marking that it's a recurring transaction. 3) User enters the information.	2) System prompts user to choose if the transaction is perpetual or if it has a certain number of instances and the interval of the transaction. 4) System creates the transaction and adds it to the database.

Alternative Flow of Events:
Step 3: User chooses an invalid interval. System returns to step 2.

Postcondition: The transaction is added to the database and the user sees it at the appropriate location in the transaction list.

User enters transaction information in the new transaction row.

User selects "recurring transaction".

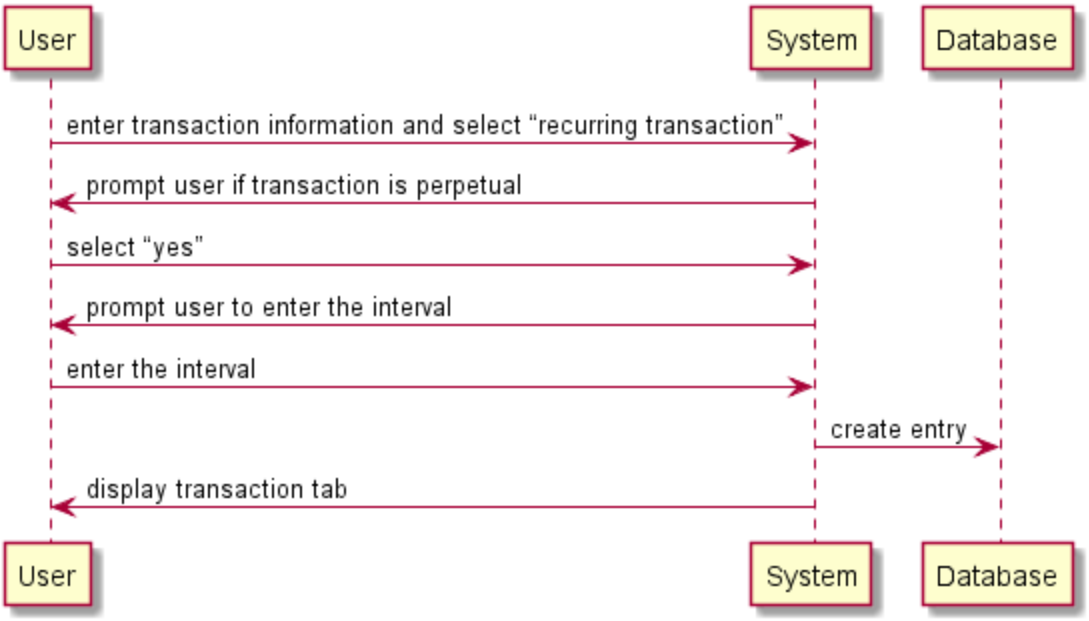
System prompts the user to choose if the transaction is perpetual.

User selects "yes".

System prompts the user to enter the interval.

User enters the interval and presses enter.

System creates the transaction and adds it to the database.



User enters transaction information in the new transaction row.

User selects "recurring transaction".

System prompts the user to choose if the transaction or if it is perpetual.

User selects "no".

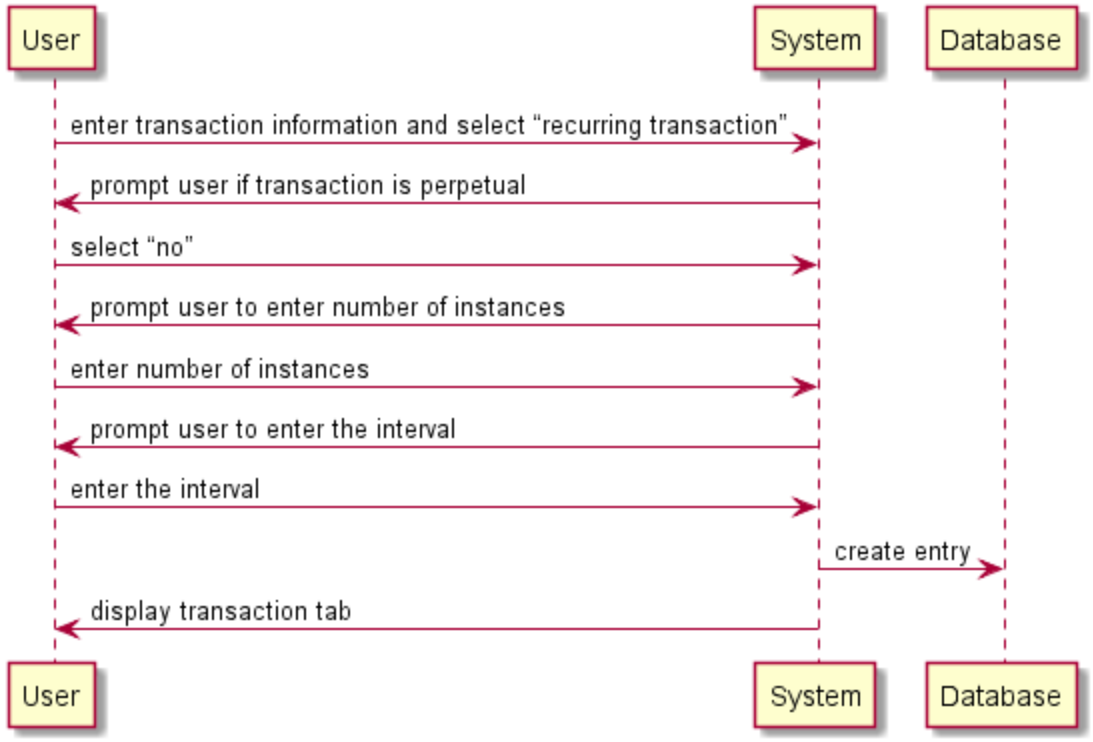
System prompts the user to enter the number of instances.

User enters the number of instances.

System prompts the user to enter the interval.

User enters the interval and presses enter.

System creates the transaction and adds it to the database.



User enters transaction information in the new transaction row.

User selects "recurring transaction".

System prompts the user to choose if the transaction or if it is perpetual.

User selects "no".

System prompts the user to enter the number of instances.

User enters the number of instances.

System prompts the user to enter the interval.

User enters the interval and presses enter.

System sees the interval is invalid.

System displays "invalid interval"

System prompts the user to enter the interval.

User enters the interval and presses enter.

System creates the transaction and adds it to the database.

Use Case: Create Budget Category
Participants: User, Database

Precondition: User is in the Budget Tab, and a monthly budget to add the category to has been selected.
Flow of Events:

Actor Intentions	System Responsibility
1) User requests to add budget category to a monthly budget. 3) Database provides available labels to system. 5) User selects a label. 7) User sets the price limit and name.	2) System queries database for available labels. 4) System presents available labels to user. 6) System prompts user to set the price limit and name. 8) System stores budget category in the database.

Alternative Flow of Events:
Step 3: No labels are available in the database. Use case will be ended prematurely after the system tells the user to create a label in the Transactions Tab.

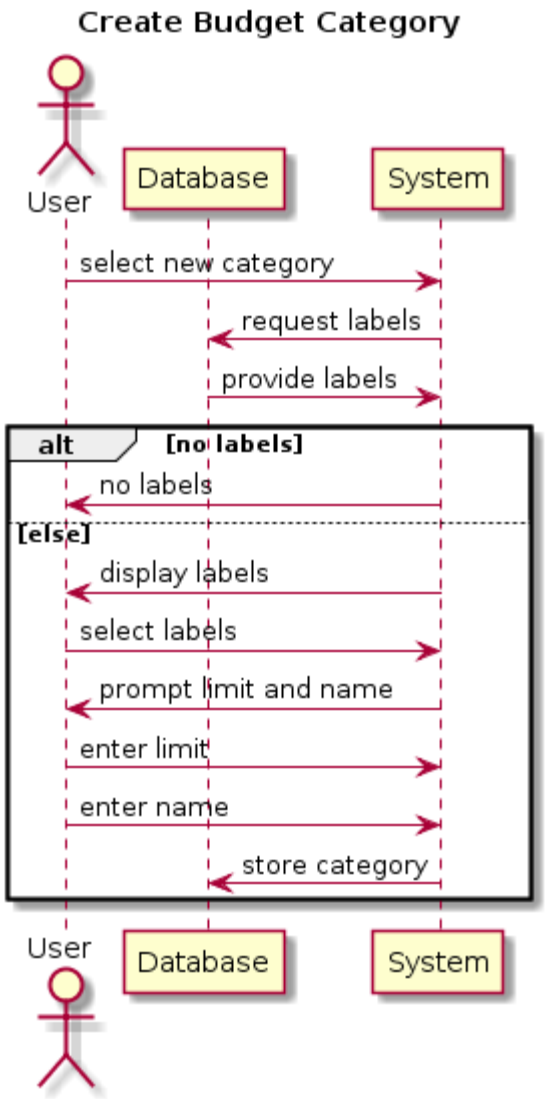
Postcondition: The budget category has been created and saved in the database.

Scenarios:

1) The user requests to add a budget category to a specific monthly budget.
   The system queries the database for the list of available labels.
   The database provides the available labels to the system.
   The system displays the available labels to the user
   The user selects a label from the list.
   The system prompts the user to set the price limit and name.
   The user enters a price.
   The user enters a name.
   The system creates the budget category and stores it in the database.

2) The user requests to add a budget category to a specific monthly budget.
   The system queries the database for the list of available labels.
   The database provides an empty list of labels.
   The system notifies the user that no labels are available and one needs to be created.
   The system ends the use case prematurely.

High Level System Sequence Diagram:


Create Budget Category
User
Database
System
select new category
request labels
provide labels
alt
[no labels]
no labels
[else]
display labels
select labels
prompt limit and name
enter limit
enter name
store category
User
Database
System

Use Case: Create Monthly Budget
Participants: User, Database
Precondition: User is in the Budget Tab
Flow of Events:

Actor Intentions	System Responsibility
1) User requests to create a monthly budget. 3) User provides the month and year.	2) System prompts user to enter the month and year of the budget. 4) System creates the monthly budget and stores it in the database.

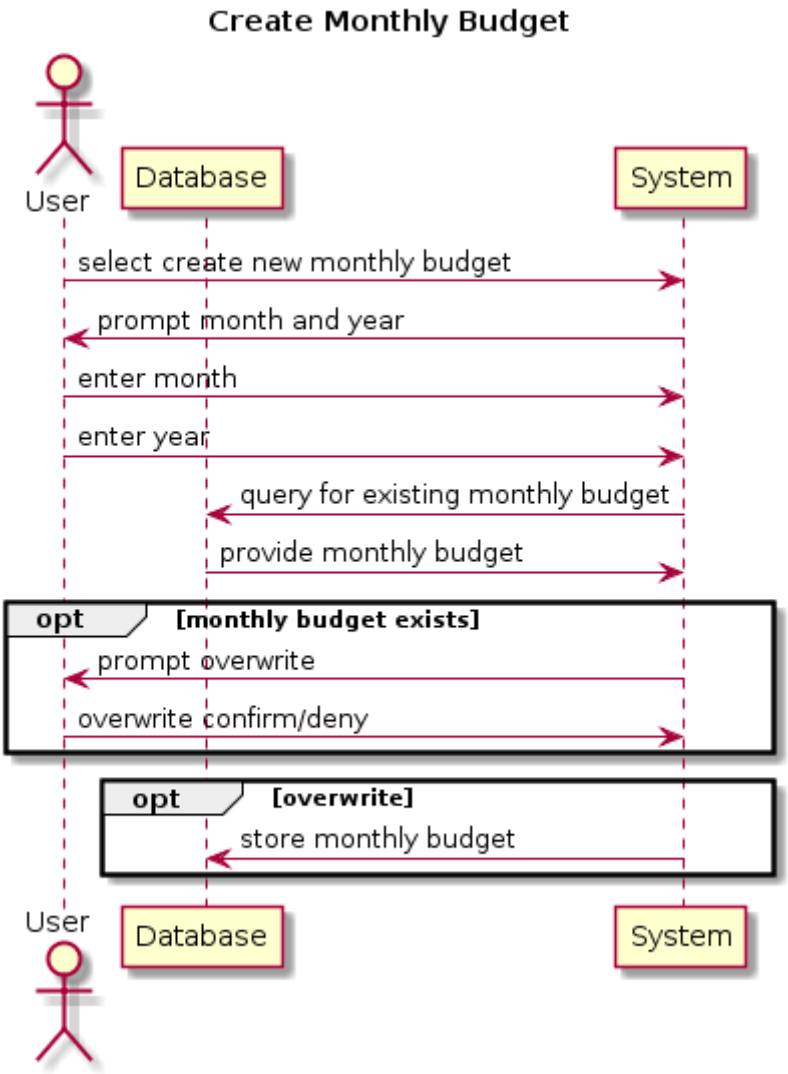
Alternative Flow of Events:
Step 4: A budget already exists in the database that is associated with that month and year. The user is asked if they wish to override the old budget. If so, the old budget in the database is overwritten. Otherwise, the use case ends.

Postcondition: The monthly budget has been created and stored in the database.

Scenarios:

1) The user requests to create a monthly budget.
   The system prompts the user to enter the month and year for the budget.
   The user enters the year.
   The user enters the month.
   The system creates the monthly budget and stores it in the database
2) The user requests to create a monthly budget.
   The system prompts the user to enter the month and year for the budget.
   The user enters the year.
   The user enters the month.
   The system recognizes that the given monthly budget already exists in the database.
   The system asks the user if they want to overwrite the old monthly budget.
   The user requests to overwrite the old budget.
   The system deletes the old monthly budget from the database.
   The system creates and stores the new monthly budget in the database.
3) The user requests to create a monthly budget.
   The system prompts the user to enter the month and year for the budget.
   The user enters the year.
   The user enters the month.
   The system recognizes that the given monthly budget already exists in the database.
   The system asks the user if they want to overwrite the old monthly budget.
   The user requests to keep the old budget.
   The system ends the use case.

High Level System Sequence Diagram:



Use Case: Create Yearly Budget
Participants: User, Database
Precondition: User is in the Budget Tab
Flow of Events:

Actor Intentions	System Responsibility
1) User requests to create a yearly budget. 3) User enters the year. 5) Database provides monthly budgets associated with that year.	2) System prompts user to enter the year. 4) System queries the database for monthly budgets associated with that year. 6) System populates the yearly budget with the monthly budgets from the database (monthly budgets are created for any missing months). 7) System stores yearly budget in the database.

Alternative Flow of Events:
Step 4: A yearly budget already exists in the database that is associated with that year. The user is asked if they wish to override the old budget. If so, the old budget in the database is overwritten. Otherwise, the use case ends.

Postcondition: The yearly budget has been created and stored in the database.

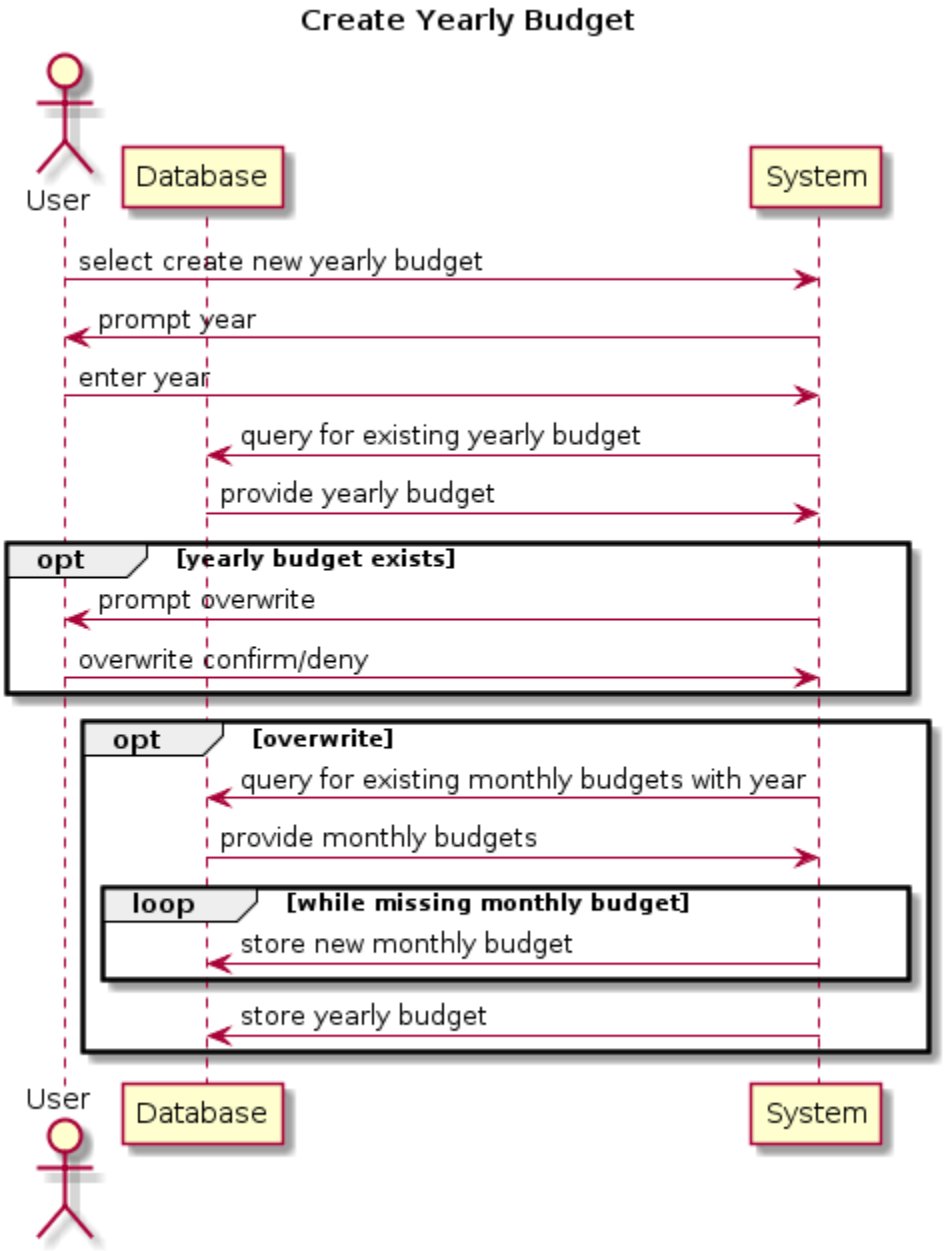
Scenarios:

1) The user requests to create a yearly budget.
   The system prompts the user to enter the year.
   The user enters the year.
   The system queries the database for monthly budgets occurring during that year.
   The database provides monthly budgets associated with that year.
   The system creates a yearly budget containing the monthly budgets from the database.
   The system fills the missing months in the yearly budget with placeholder monthly budgets.
   The system stores yearly budgets in the database.

2) The user requests to create a yearly budget.
   The system prompts the user to enter the year.
   The user enters the year.
   The system recognizes that the yearly budget with that year exists in the database.
   The system asks the user if they want to overwrite the old yearly budget.
   The user requests to overwrite the old yearly budget.
   The system queries the database for monthly budgets occurring during that year.
   The database provides monthly budgets associated with that year.
   The system creates a yearly budget containing the monthly budgets from the database.
   The system fills the missing months in the yearly budget with placeholder monthly budgets.
   The system stores yearly budgets in the database.

3) The user requests to create a yearly budget.
   The system prompts the user to enter the year.
   The user enters the year.
   The system recognizes that the yearly budget with that year exists in the database.
   The system asks the user if they want to overwrite the old yearly budget.
   The user requests to keep the old yearly budget.
   The system ends the use case prematurely.

High Level System Sequence Diagram:



Use Case: See Budget Category Transactions
Participants: User, Database
Precondition: User is in the Budget Tab, User has selected a budget category
Flow of Events:

Actor Intentions	System Responsibility
1) User requests to view transactions belonging to a budget category. 3) Database provides the transactions to the system.	2) System queries database for transactions sharing the same label, month, and year as the given budget category. 4) System provides list of transactions to the user.

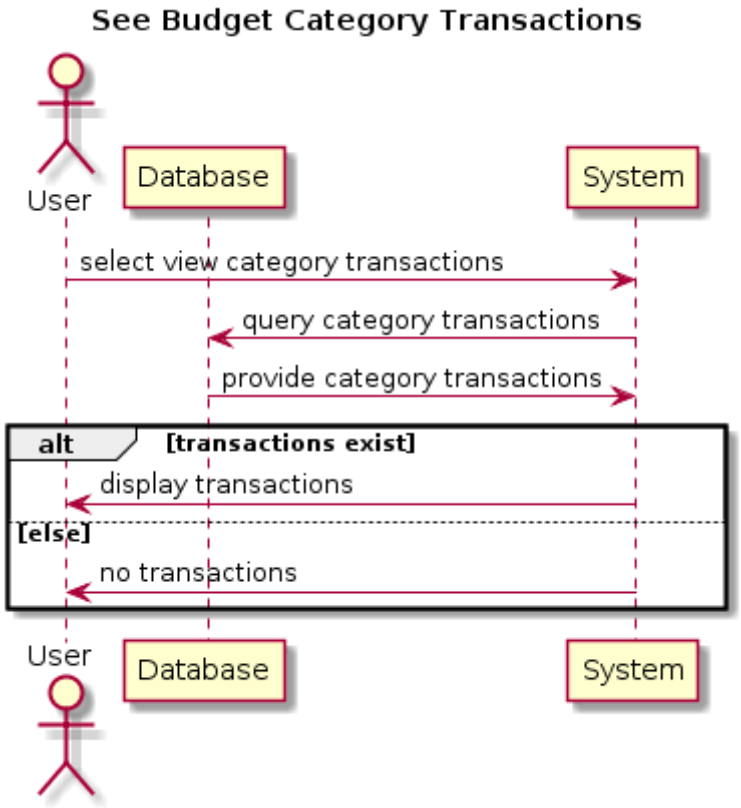
Alternative Flow of Events:
Step 2: No transactions with the given constraints exist in the database. The user will be notified that no transactions meet their constraints, then the use case will end.

Postcondition: The user has been shown the transactions belonging to the budget category they selected.

Scenarios:

1) The user requests to view transactions belonging to a selected budget category.
   The system queries the database for transactions sharing the same label, month, and year as the given budget category.
   The database provides the transactions to the system.
   The system provides the list of transactions to the user.
2) The user requests to view transactions belonging to a selected budget category.
   The system queries the database for transactions sharing the same label, month, and year as the given budget category.
   The database notifies the system that no transactions in the database meet the constraints.
   The system notifies the user that no transactions currently belong to the budget category.
   The system ends the use case prematurely.

High Level System Sequence Diagram:



Use Case: Add Transaction Label
Participants: User, Database
Precondition: User is in the Transactions Tab
Flow of Events:

Actor Intentions	System Responsibility
1) User requests to create a transaction label. 3) User enters the label name.	2) System prompts user to enter the label name. 4) System creates a label and stores the label in the database.

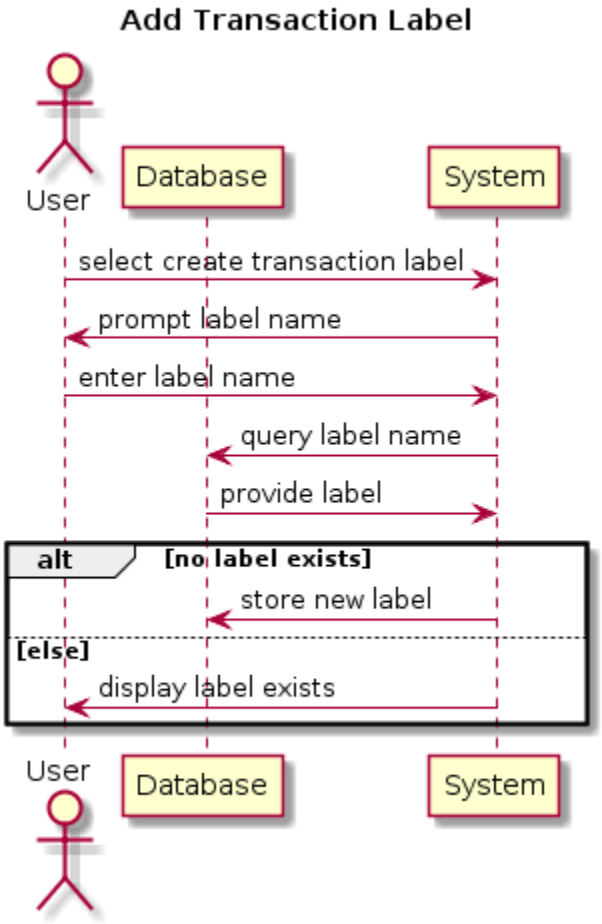
Alternative Flow of Events:
Step 4: A label sharing that name has already been stored in the database. The user will be notified, and the use case will end prematurely.

Postcondition: The user has created a label that has been stored in the database.

Scenarios:

1) The user requests to create a transaction label.
   The system prompts the user to enter a label name.
   The user enters a label name.
   The system creates a label and stores the label in the database.
2) The user requests to create a transaction label.
   The system prompts the user to enter a label name.
   The user enters a label name.
   The system creates a label.
   The system attempts to add the label to the database.
   The database notifies the system that that label already exists.
   The system notifies the user that the entered label already exists.
   The system ends the use case prematurely.

High Level System Sequence Diagram:



Use Case: Add Bank Account

Participants: User and Database Manager

Precondition: User is in the Bank Account Tab

Actor Intentions	**System Responsibility**
1. User selects Add a Bank Account 3. User selects Account Type 4. User inputs amount for Bank Account 5. User inputs Interest Rate and submits the form	2. Displays a form with a Bank Account type selection, input amount box, and input interest rate box 6. Creates appropriate bank account and sends the bank account to the database

Alternative Course:

Step 4: User inputs an invalid amount and the System asks for a correct one.

Step 5: User inputs an invalid interest rate and the System asks for a correct one.

Post condition: The new Bank Account is displayed in the Bank Account Tab

Scenarios:

1. User selects add a Bank Account

   System displays a Bank Account Form

   User selects an Account Type from a list

   User inputs the amount

   User inputs the interest rate

   User submits the form

   System takes the form and makes a bank account

   System sends the bank account and its info to the database

2. User selects add a Bank Account

   System displays a Bank Account Form

   User selects an Account Type from a list

   User inputs an invalid amount

   User inputs the interest rate

   User submits the form

   System asks the User to input a valid amount

   User selects an Account Type from a list

   User inputs the valid amount

   User inputs the interest rate

   User submits the form

   System takes the form and makes a bank account

   System sends the bank account and its info to the database

3. User selects add a Bank Account

   System displays a Bank Account Form

   User selects an Account Type from a list

   User inputs an amount

   User inputs an invalid interest rate

   User submits the form

   System asks the User to input a valid interest rate

   User selects an Account Type from a list

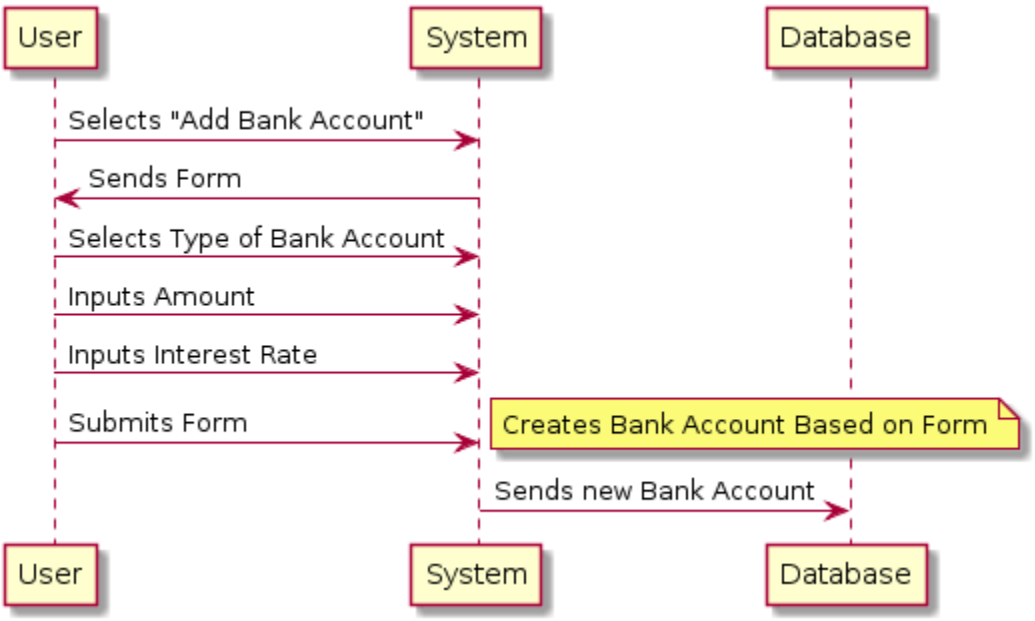
   User inputs the amount

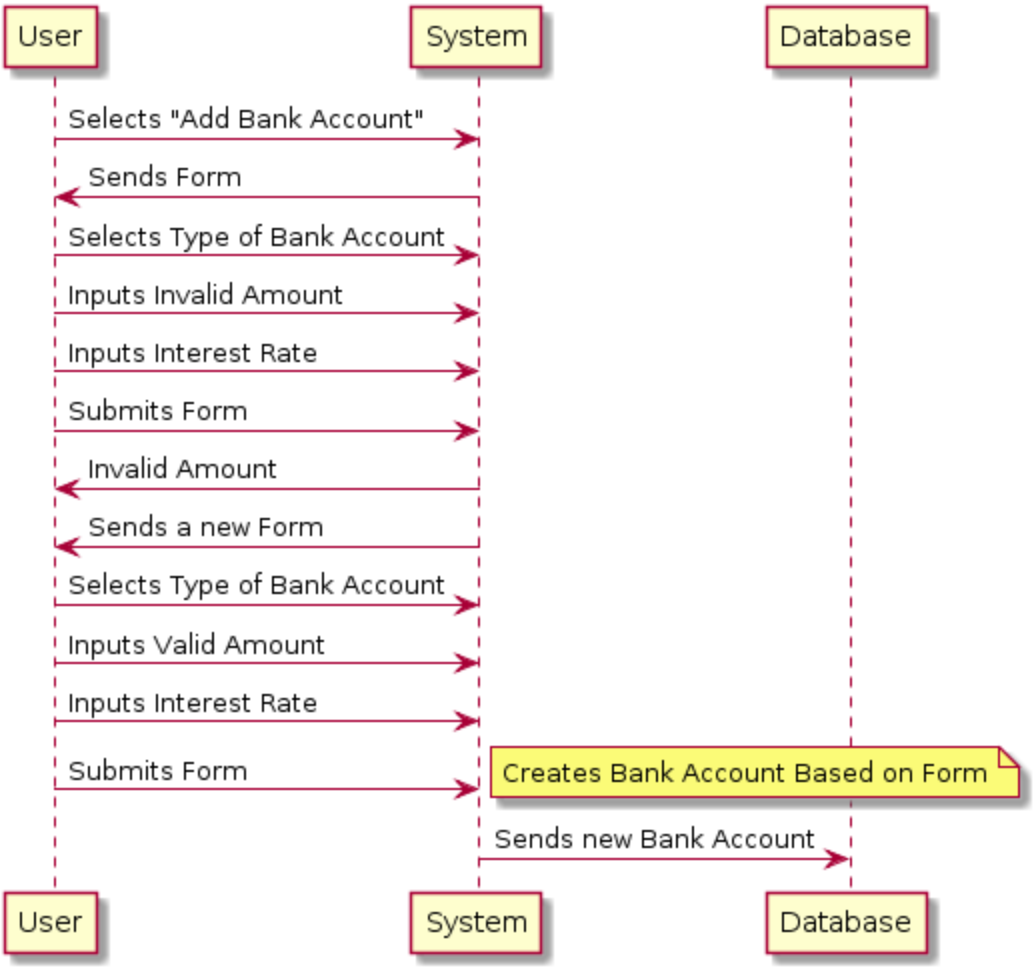
   User inputs a valid interest rate

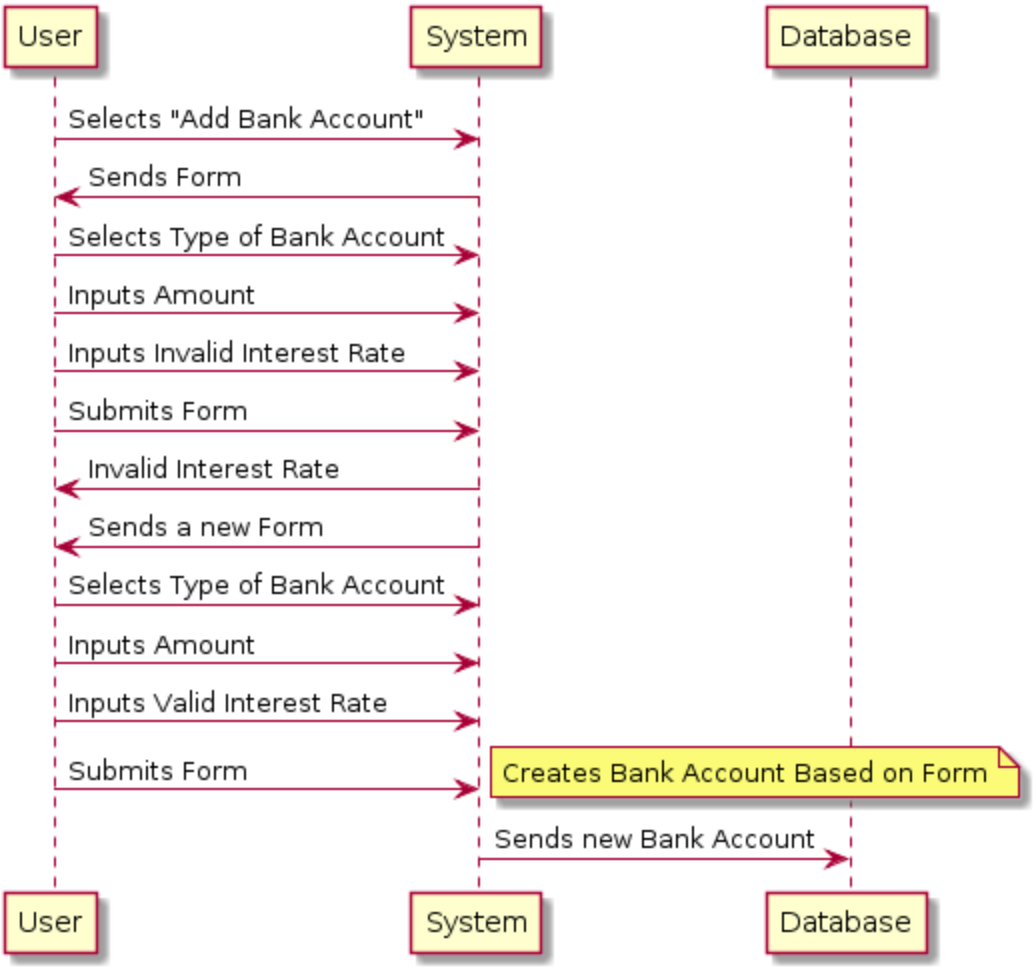
   User submits the form

   System takes the form and makes a bank account

   System sends the bank account and its info to the database




User
System
Database
Selects "Add Bank Account"
Sends Form
Selects Type of Bank Account
Inputs Invalid Amount
Inputs Interest Rate
Submits Form
Invalid Amount
Sends a new Form
Selects Type of Bank Account
Inputs Valid Amount
Inputs Interest Rate
Submits Form
Creates Bank Account Based on Form
Sends new Bank Account
User
System
Database


User
System
Database
Selects "Add Bank Account"
Sends Form
Selects Type of Bank Account
Inputs Amount
Inputs Invalid Interest Rate
Submits Form
Invalid Interest Rate
Sends a new Form
Selects Type of Bank Account
Inputs Amount
Inputs Valid Interest Rate
Submits Form
Creates Bank Account Based on Form
Sends new Bank Account
User
System
Database

Use Case: See Bank Account Info

Participants: User and Database Manager

Precondition: User is in the Bank Account Tab

Actor Intentions	**System Responsibility**
1. User selects a Bank Account to display 3. User closes when done viewing Bank Account	2. Gets all the Bank Account's transactions and displays them

Post condition: The Bank Account Tab returns to the state before viewing the Bank Account Info.

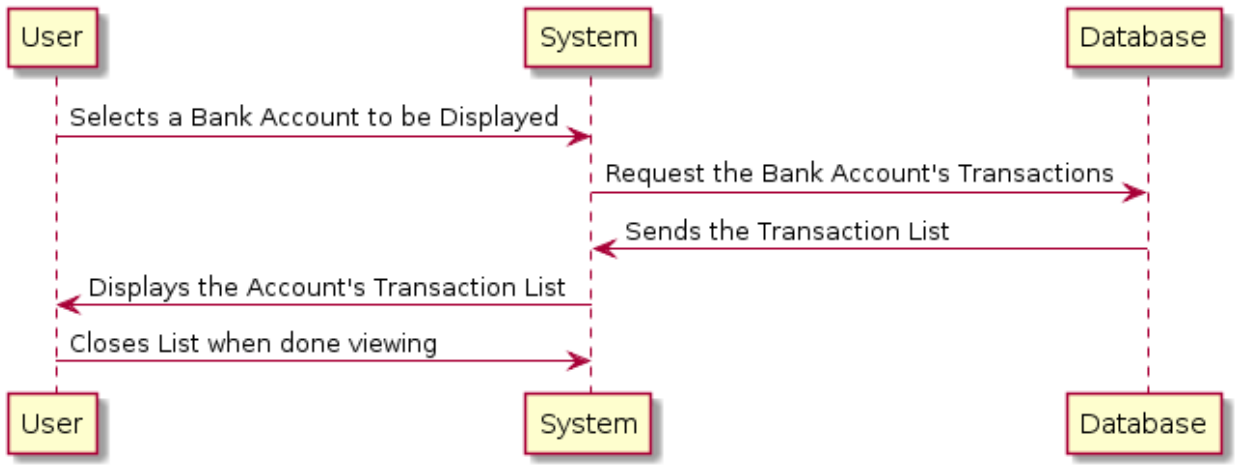
Scenario:

User selects one of their Bank Accounts to be displayed

System asks the database for the Bank Account and all of its transactions

System displays the Bank Account and all of the transactions

User closes the list when done viewing the Bank Account



Use Case: Look up stock/crypto

Participants: User and Investment Manager

Precondition: User is in the Investment Tab

Actor Intentions	**System Responsibility**
1. User selects the search bar 2. User inputs the stock/crypto symbol 4. User can scroll through list and cancel search when they want	3. Displays a list of stocks/cryptos with prices and a symbol similar to what the User input

Alternative Course:

Step 2: User inputs an invalid or nonexistent symbol and the System displays nothing in the search list.

Post condition: The Investment Tab returns to the state before searching for stock/crypto.

Scenarios:

1. User selects the search bar

   User inputs a symbol for the stock/crypto they are searching

   System displays a list of stocks/crypto that contain the symbol

   User cancels the search once they are done

2. User selects the search bar

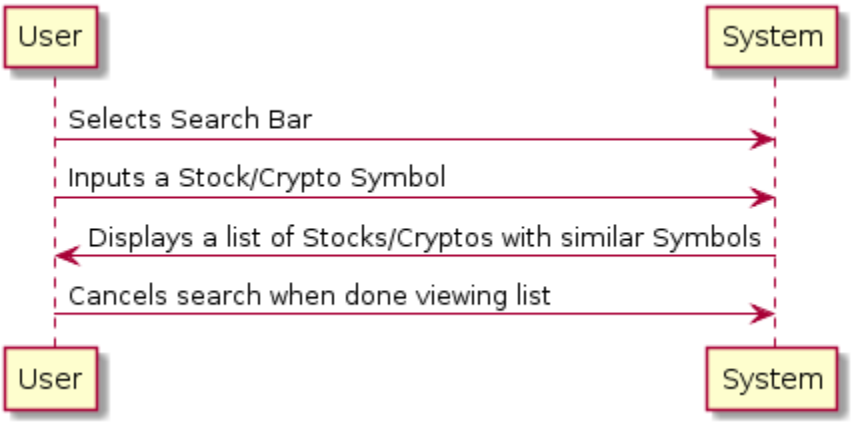
   User inputs an invalid/nonexistent symbol for the stock/crypto they are searching

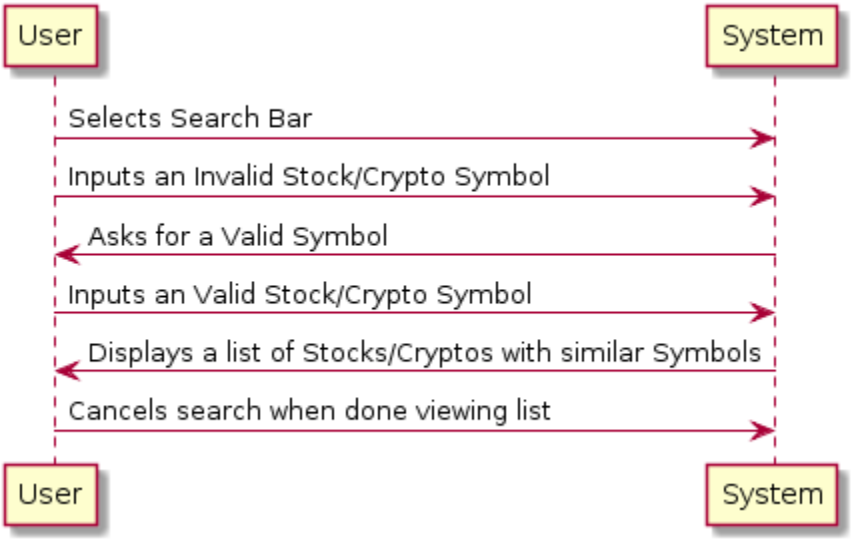
   System displays a list of nothing

   User inputs a valid symbol for the stock/crypto they are searching

   System displays a list of stocks/crypto that contain the symbol

User cancels the search once they are done





Use Case: Record Investment in stock/crypto

Participants: User and Database Manager

Precondition: User is in the Investment Tab

Actors Intentions	**System Requirements**
1. User selects to record a new stock/crypto investment 3. User inputs the symbol 5. User inputs the number of the stock/crypto	2. Displays a symbol input box 4. Asks for the number of the stock/crypto the User has 6. Creates and adds the investment to the investments list

Alternative Course:

Step 3: User inputs an invalid or nonexistent symbol and the System asks for a correct one.

Step 5: User inputs an invalid input and the System asks for a correct one.

Post condition: The Investment Tab returns to the state before adding the stock/crypto with the added stock/crypto on the investment list.

Scenarios:

1. User selects to record a new stock/crypto investment

   System asks for a stock/crypto symbol

   User inputs the symbol for the stock/crypto

   System asks for the number of shares for the stock/crypto

   User inputs the number of shares

   System creates the investment and sends it to the database to be added to the User's investment list

2. User selects to record a new stock/crypto investment

   System asks for a stock/crypto symbol

   User inputs an invalid/nonexistent symbol for the stock/crypto

   System asks for a valid symbol

   User inputs a valid symbol

   System asks for the number of shares for the stock/crypto

   User inputs the number of shares

   System creates the investment and sends it to the database to be added to the User's investment list

3. User selects to record a new stock/crypto investment

   System asks for a stock/crypto symbol

   User inputs the symbol for the stock/crypto

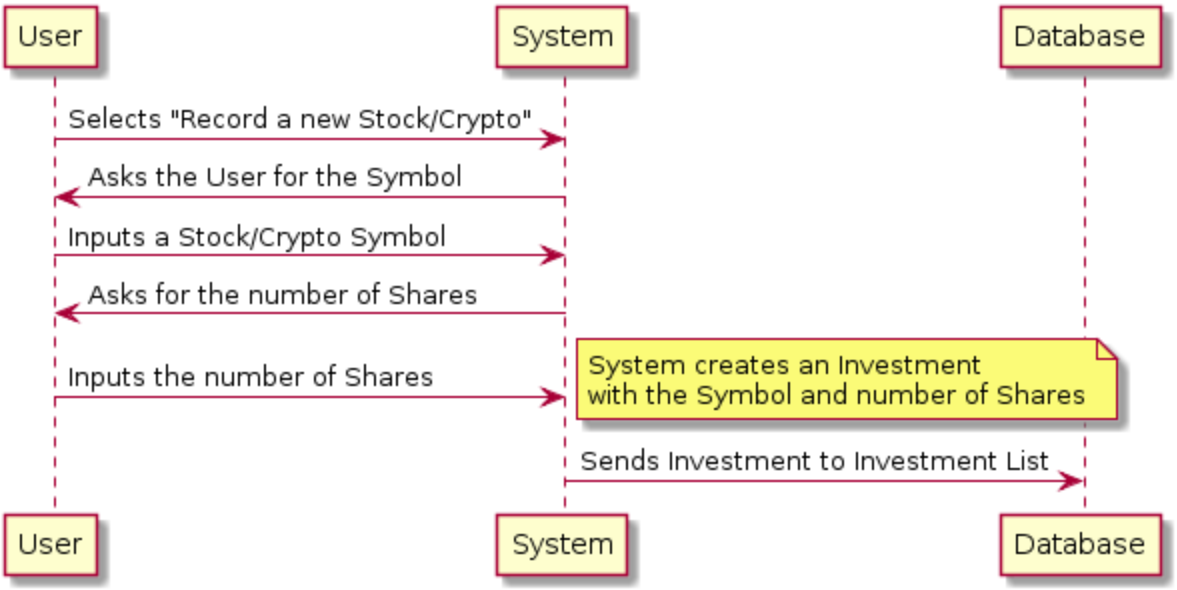
   System asks for the number of shares for the stock/crypto

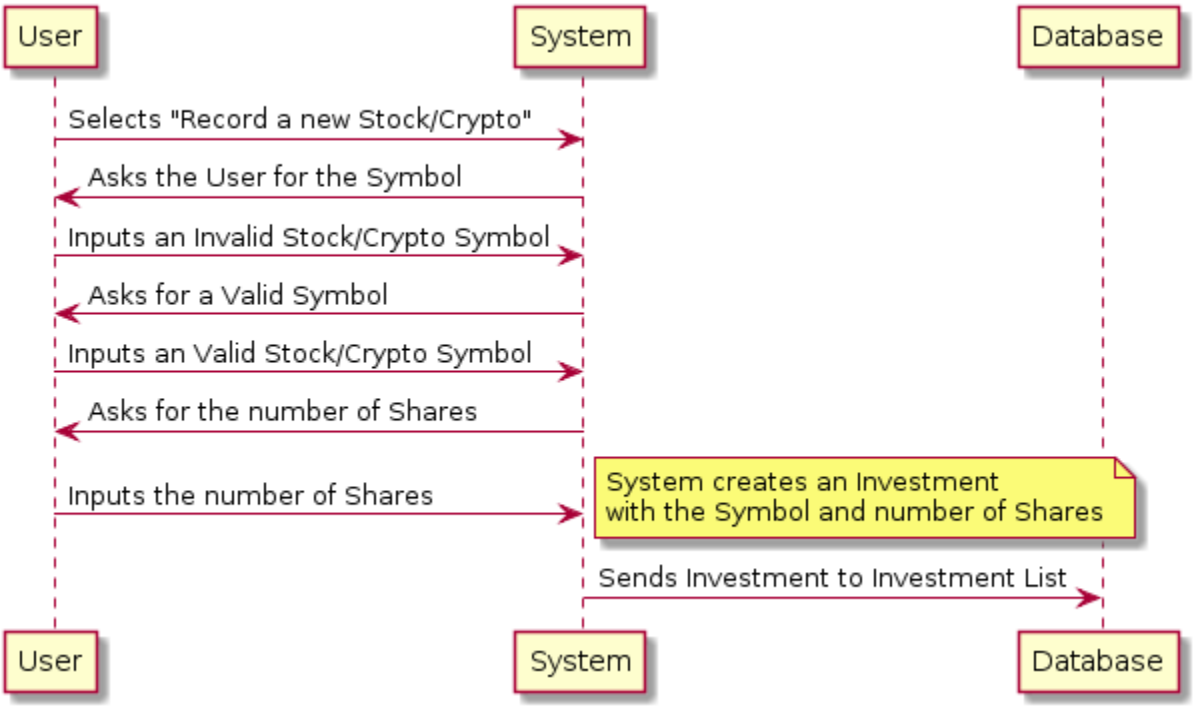
   User inputs an invalid input for the number of shares

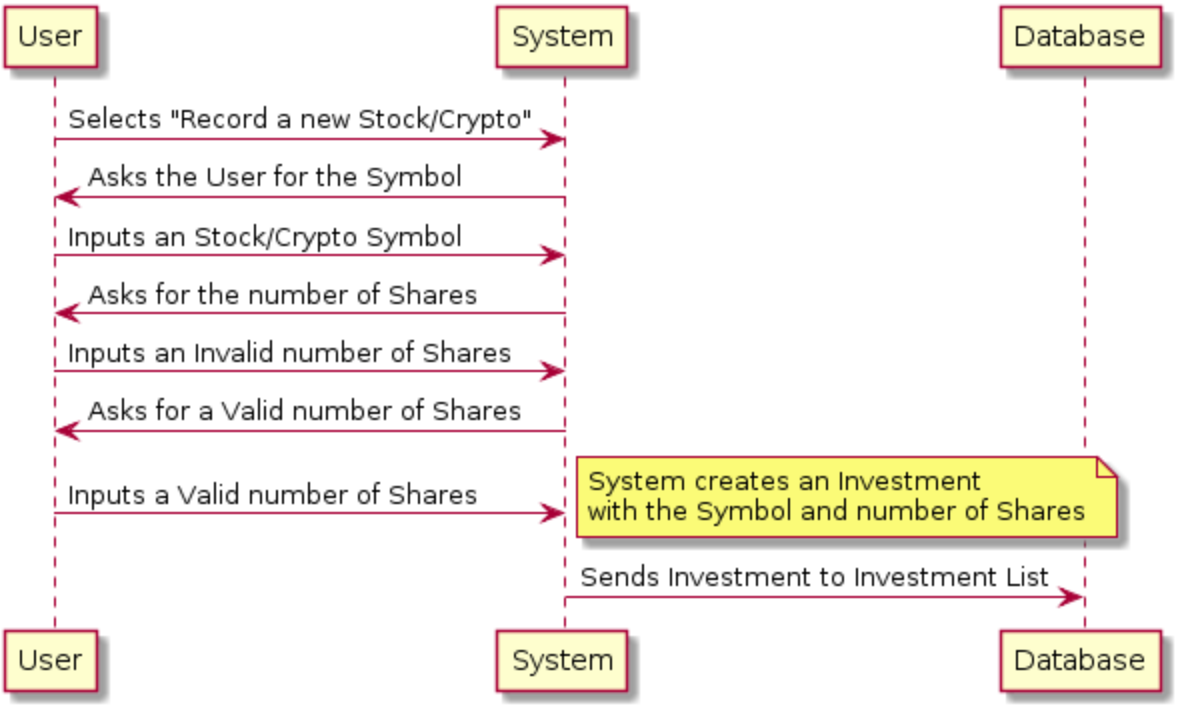
   System asks for a valid input for the number of shares

   User inputs a valid number of shares

   System creates the investment and sends it to the database to be added to the User's investment list


User
System
Database
Selects "Record a new Stock/Crypto"
Asks the User for the Symbol
Inputs a Stock/Crypto Symbol
Asks for the number of Shares
Inputs the number of Shares
System creates an Investment
with the Symbol and number of Shares
Sends Investment to Investment List
User
System
Database


User
System
Database
Selects "Record a new Stock/Crypto"
Asks the User for the Symbol
Inputs an Invalid Stock/Crypto Symbol
Asks for a Valid Symbol
Inputs an Valid Stock/Crypto Symbol
Asks for the number of Shares
Inputs the number of Shares
System creates an Investment
with the Symbol and number of Shares
Sends Investment to Investment List
User
System
Database


User
System
Database
Selects "Record a new Stock/Crypto"
Asks the User for the Symbol
Inputs an Stock/Crypto Symbol
Asks for the number of Shares
Inputs an Invalid number of Shares
Asks for a Valid number of Shares
Inputs a Valid number of Shares
System creates an Investment
with the Symbol and number of Shares
Sends Investment to Investment List
User
System
Database

Use Case: Record Investment in custom asset

Participants: User and Database Manager

Precondition: User is in the Investment Tab

Actors Intentions	**System Requirements**
1. User selects to record a new custom asset 3. User inputs the asset's current value 4. User inputs the asset's name 5. User inputs the asset's interest rate	2. Displays a form asking for the asset's current value, the asset's name, and the interest rate 6. Creates the asset and adds it to the investment list

Alternative Course:

Step 3: User inputs an invalid value and the system asks for a correct one.

Step 5: User inputs an invalid interest rate and the system asks for a correct one.

Post condition: The Investment Tab returns to the state before adding the asset with the added asset on the investment list.

Scenarios:

1. User selects to record a custom asset

   System displays a form to make a custom asset

   User inputs the asset's current value

   User inputs the asset's name

   User inputs the asset's interest rate

   System takes the form and makes a custom asset then sends it to the database to be added to the investment list

2. User selects to record a custom asset

   System displays a form to make a custom asset

   User inputs an invalid input for the asset's current value

   User inputs the asset's name

   User inputs the asset's interest rate

   System asks for a valid input for the current value

   User inputs a valid input for the asset's current value

   User inputs the asset's name

   User inputs the asset's interest rate

   System takes the form and makes a custom asset then sends it to the database to be added to the investment list

3. User selects to record a custom asset

   System displays a form to make a custom asset

   User inputs the asset's current value

   User inputs the asset's name

   User inputs an invalid input for the asset's interest rate

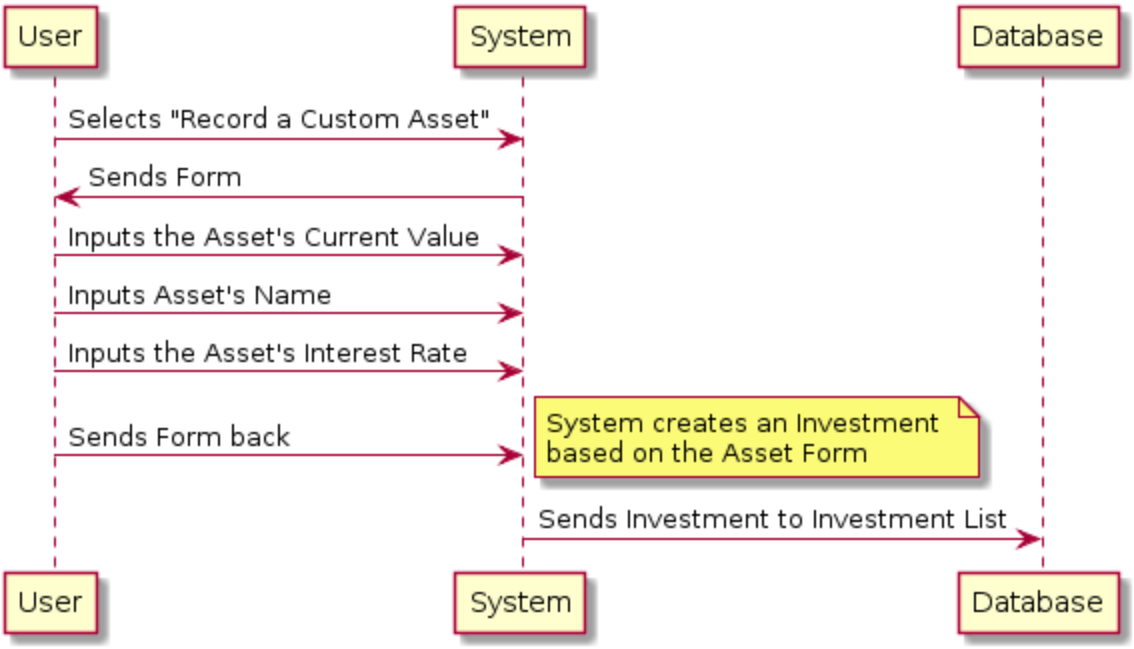
   System asks for a valid input for the interest rate

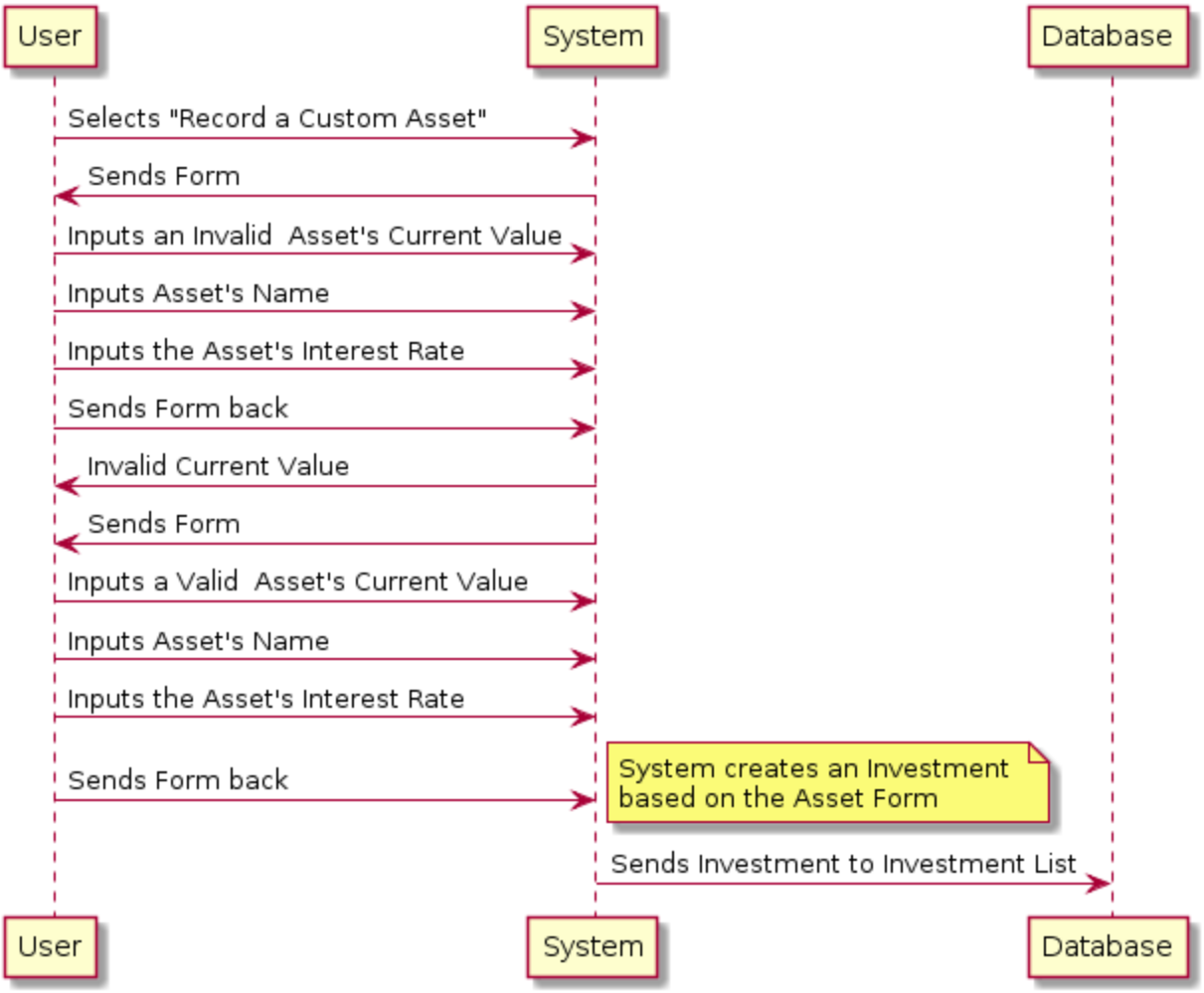
   User inputs the asset's current value

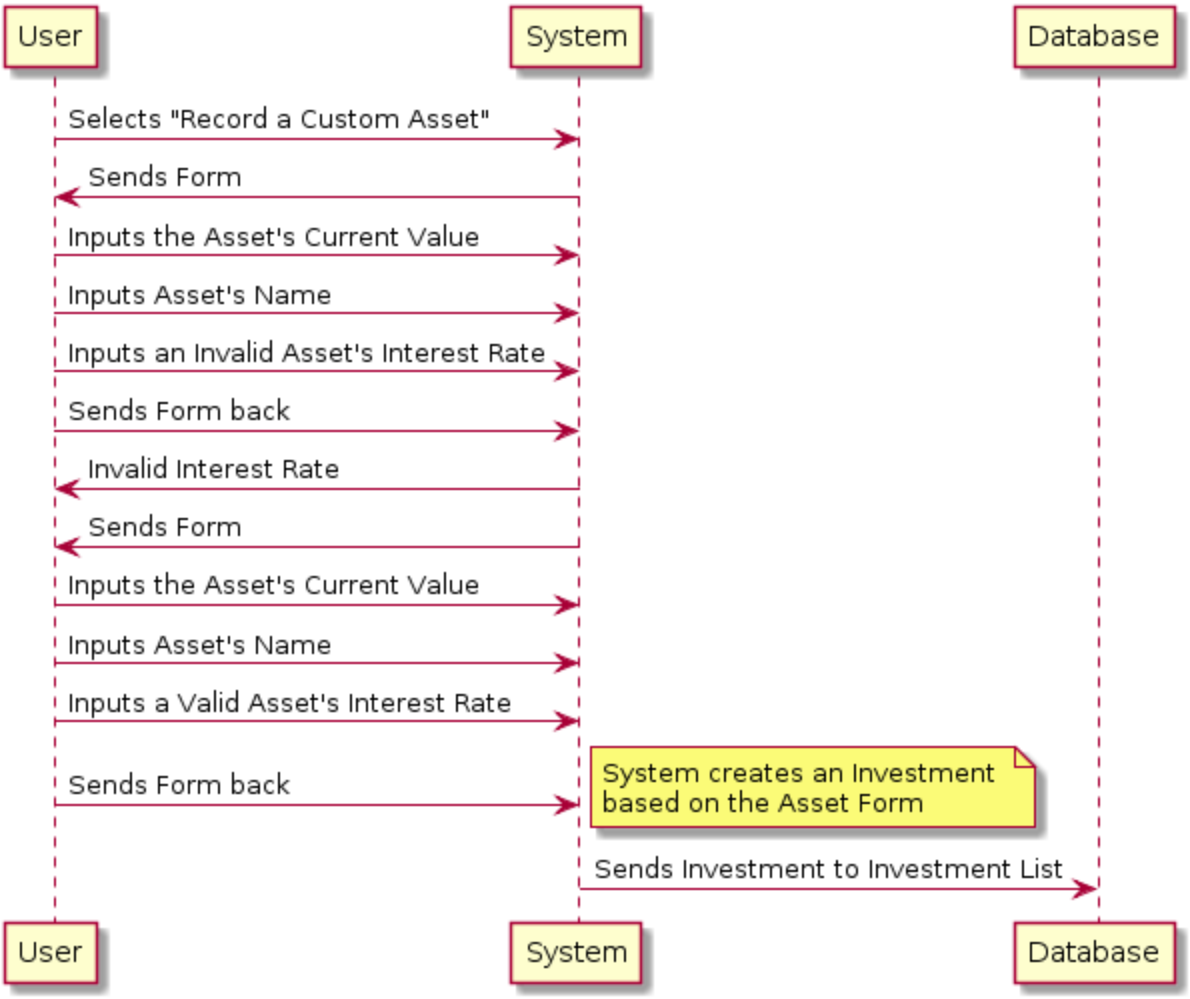
   User inputs the asset's name

   User inputs a valid input the asset's interest rate

   System takes the form and makes a custom asset then sends it to the database to be added to the investment list

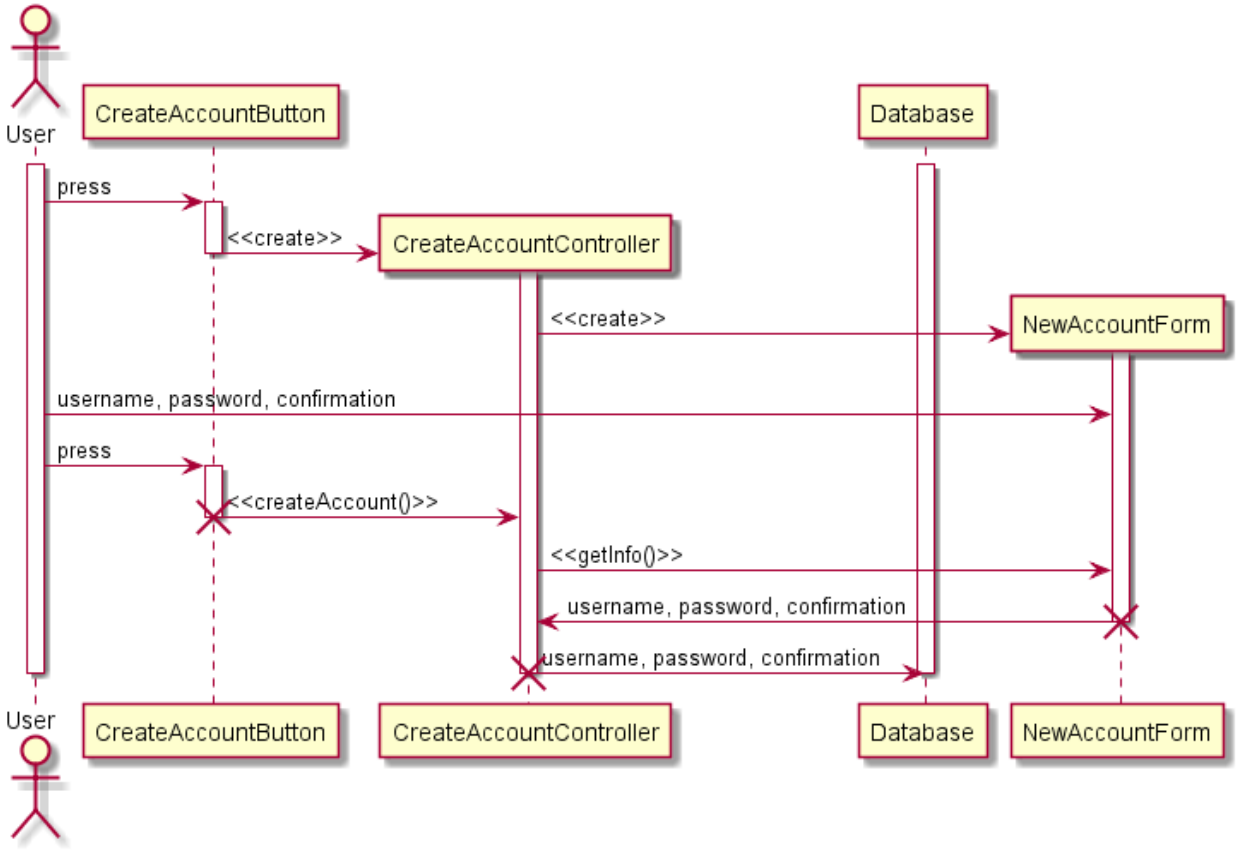

User
System
Database
Selects "Record a Custom Asset"
Sends Form
Inputs the Asset's Current Value
Inputs Asset's Name
Inputs the Asset's Interest Rate
Sends Form back
System creates an Investment based on the Asset Form
Sends Investment to Investment List
User
System
Database


User
System
Database
Selects "Record a Custom Asset"
Sends Form
Inputs an Invalid Asset's Current Value
Inputs Asset's Name
Inputs the Asset's Interest Rate
Sends Form back
Invalid Current Value
Sends Form
Inputs a Valid Asset's Current Value
Inputs Asset's Name
Inputs the Asset's Interest Rate
Sends Form back
System creates an Investment based on the Asset Form
Sends Investment to Investment List
User
System
Database

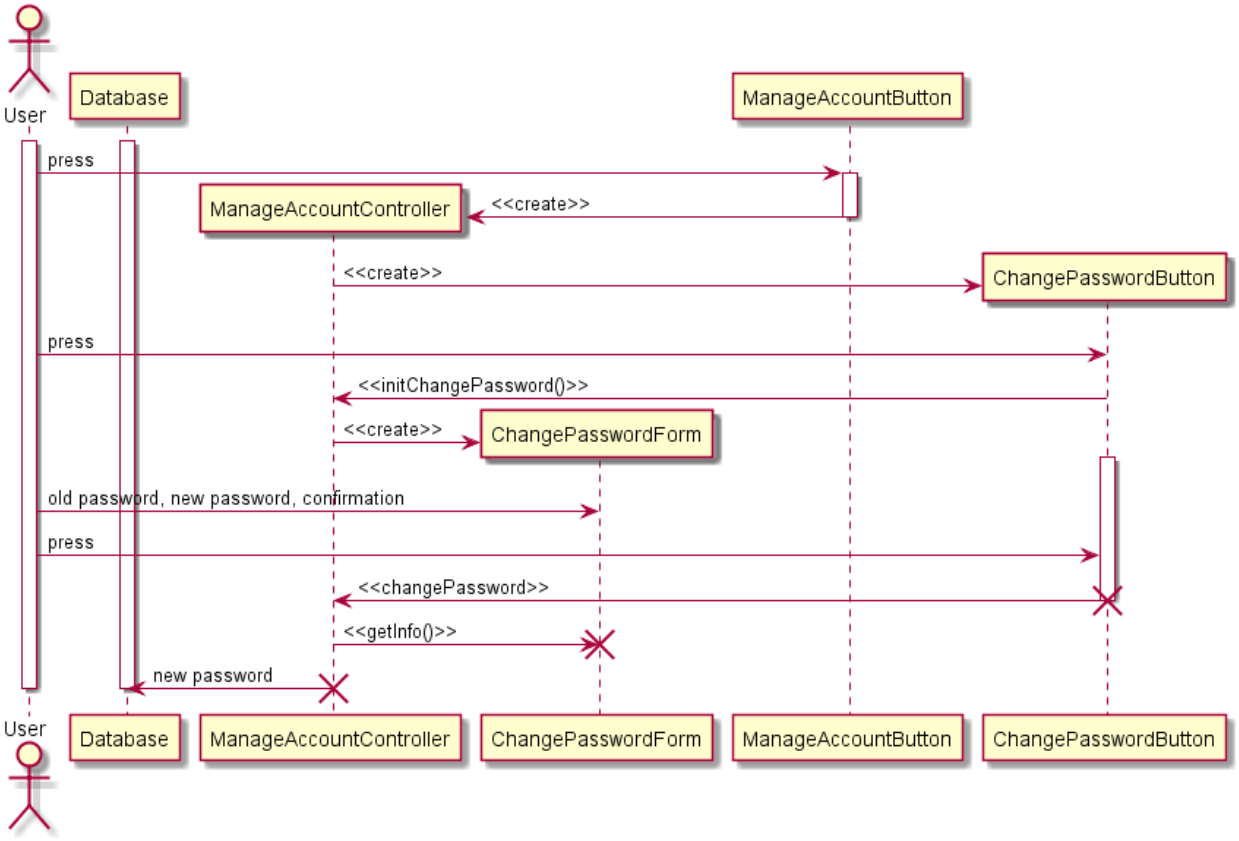

User
System
Database
Selects "Record a Custom Asset"
Sends Form
Inputs the Asset's Current Value
Inputs Asset's Name
Inputs an Invalid Asset's Interest Rate
Sends Form back
Invalid Interest Rate
Sends Form
Inputs the Asset's Current Value
Inputs Asset's Name
Inputs a Valid Asset's Interest Rate
Sends Form back
System creates an Investment based on the Asset Form
Sends Investment to Investment List
User
System
Database

Concrete Use Cases, DSSD:

Use case name: Create User Account	
Entry condition:	1) The is currently viewing the login screen and selects "create account"
Flow of events:	2) CreateAccountController responds by presenting a form to the user. The form includes username, password, and confirm password fields. 3) User fills out the form to specifications. The username is valid, and both passwords are valid and match each other. Once the form is complete, the user selects "create account". 4) CreateAccountController validates the information then passes the it to Database.
Exit condition:	5) CreateAccountController displays confirmation and returns user to login screen.

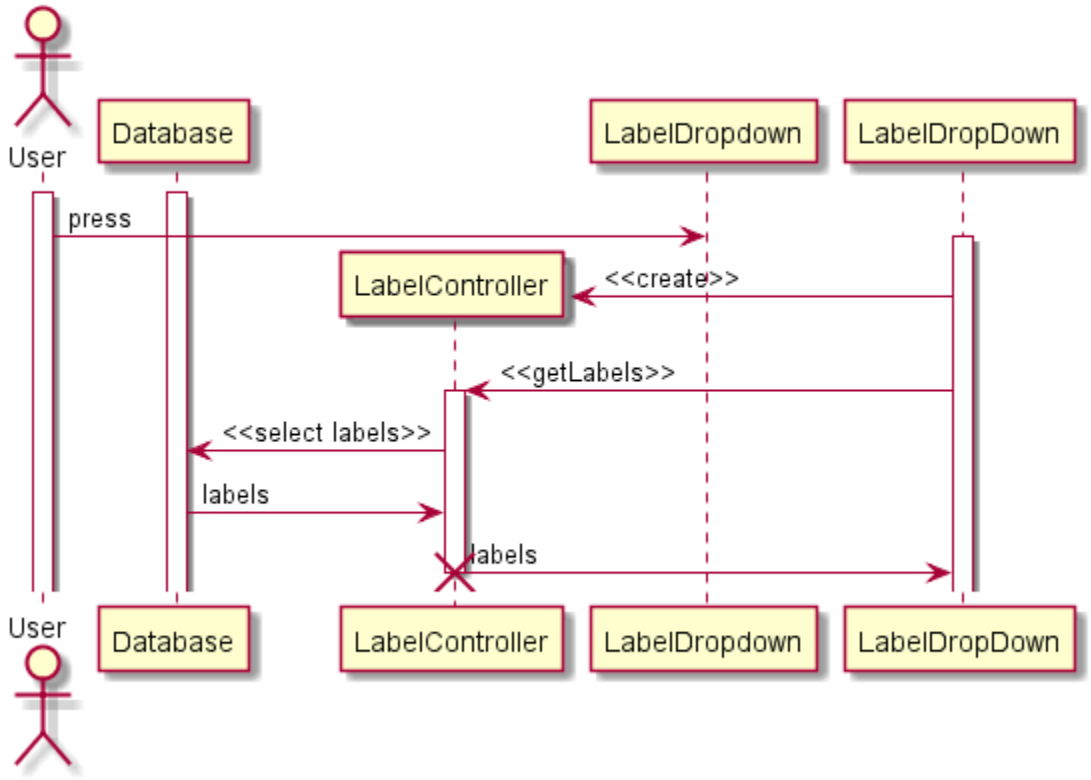


Use case name:	Change Password
Entry Condition:	1) User is viewing the home screen and selects "manage account"
Flow of Events:	2) AccountMangerController responds by presenting the user with the account management screen. 3) User selects "change password" 4) AccountManagerController presents the user a form including fields for their old password, new password and confirmation of new password. 5) User completes the fields and presses "change password". 6) AccountManagerController updates the Database.
Exit Condition:	7) AccountManagerController displays confirmation and returns user to homescreen.



Use case name:	View Labels
Entry Condition:	1) User is viewing the transaction tab and selects the "label" dropdown.

Flow of Events:	2) LabelController queries Database for the list of labels. 3) Database returns the list of labels.
Exit Condition:	4) LabelController populates the list of labels in the dropdown.



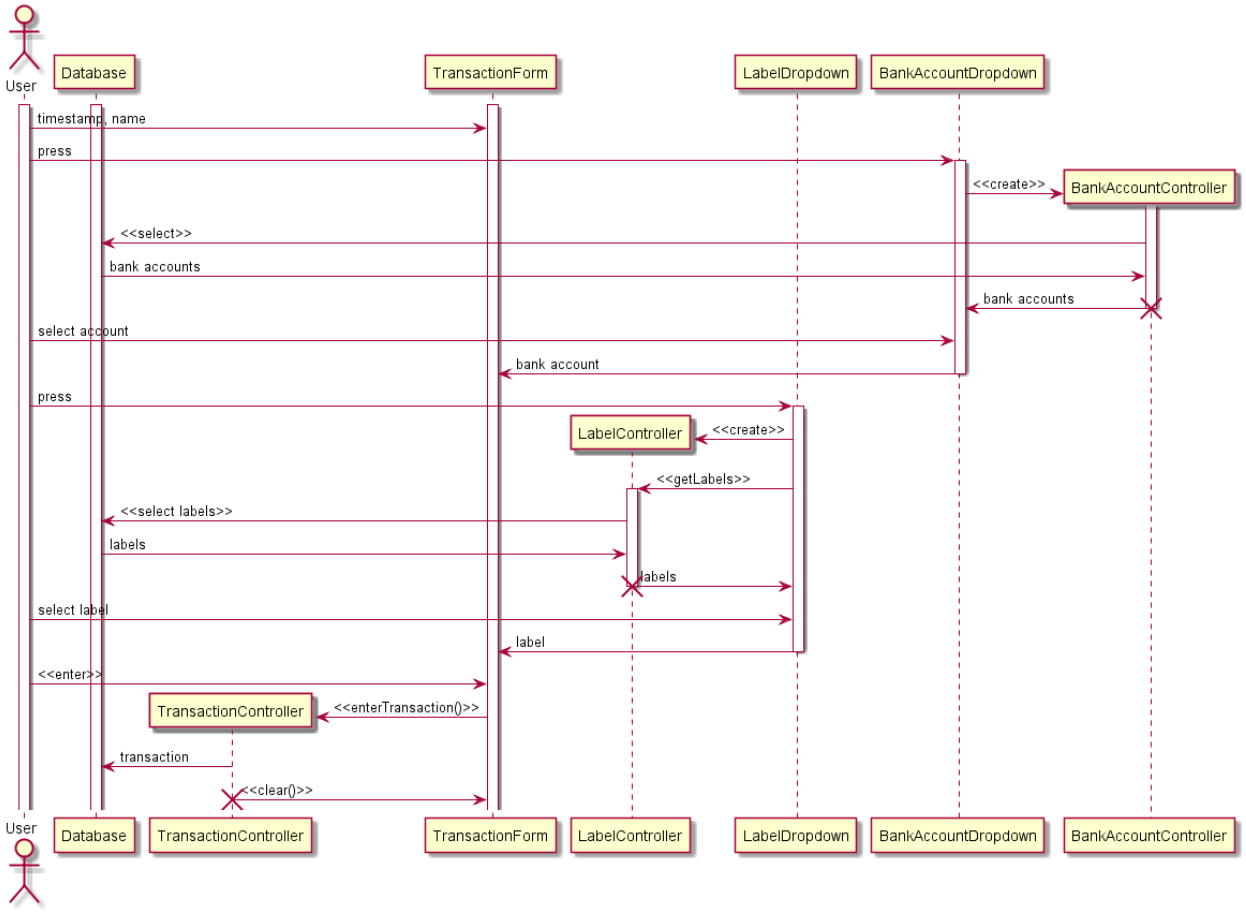
Use case name:	Input Transaction
Entry Condition:	1) User is viewing the Transaction Tab.
Flow of Events:	2) User fills in the fields at the top row of the transaction list. 3) User selects the labels dropdown. 4) LabelController queries Database for the list of labels. 5) Database returns the list of labels. 6) LabelController populates the list of labels in the dropdown. 7) User selects a label in the list. 8) LabelController closes the dropdown and displays the selected label. 9) User selects the bank accounts dropdown 10) BankAccountController queries Database for the list of bank accounts. 11) Database returns the list of bank accounts. 12) BankAccountController populates the list of bank accounts in the dropdown. 13) User selects a bank account in the list.

14) BankAccountController closes the dropdown and displays the selected bank account

15) User presses “enter” to submit the transaction.

16) TransactionController adds the transaction to the Database.

Exit Condition:	17) TransactionController shows the transaction in the list.



Use case name:	Input Recurring Transaction

Entry Condition: 1) User is viewing the transaction tab.

Flow of Events:

2) User fills in the fields at the top row of the transaction list.

3) User selects the labels dropdown.

4) LabelController queries Database for the list of labels.

5) Database returns the list of labels.

6) LabelController populates the list of labels in the dropdown.

7) User selects a label in the list.

8) LabelController closes the dropdown and displays the selected label.

9) User selects the bank accounts dropdown

10) BankAccountController queries Database for the list of bank accounts.

11) Database returns the list of bank accounts.

12) BankAccountController populates the list of bank accounts in the dropdown.

13) User selects a bank account in the list.

14) BankAccountController closes the dropdown and displays the selected bank account

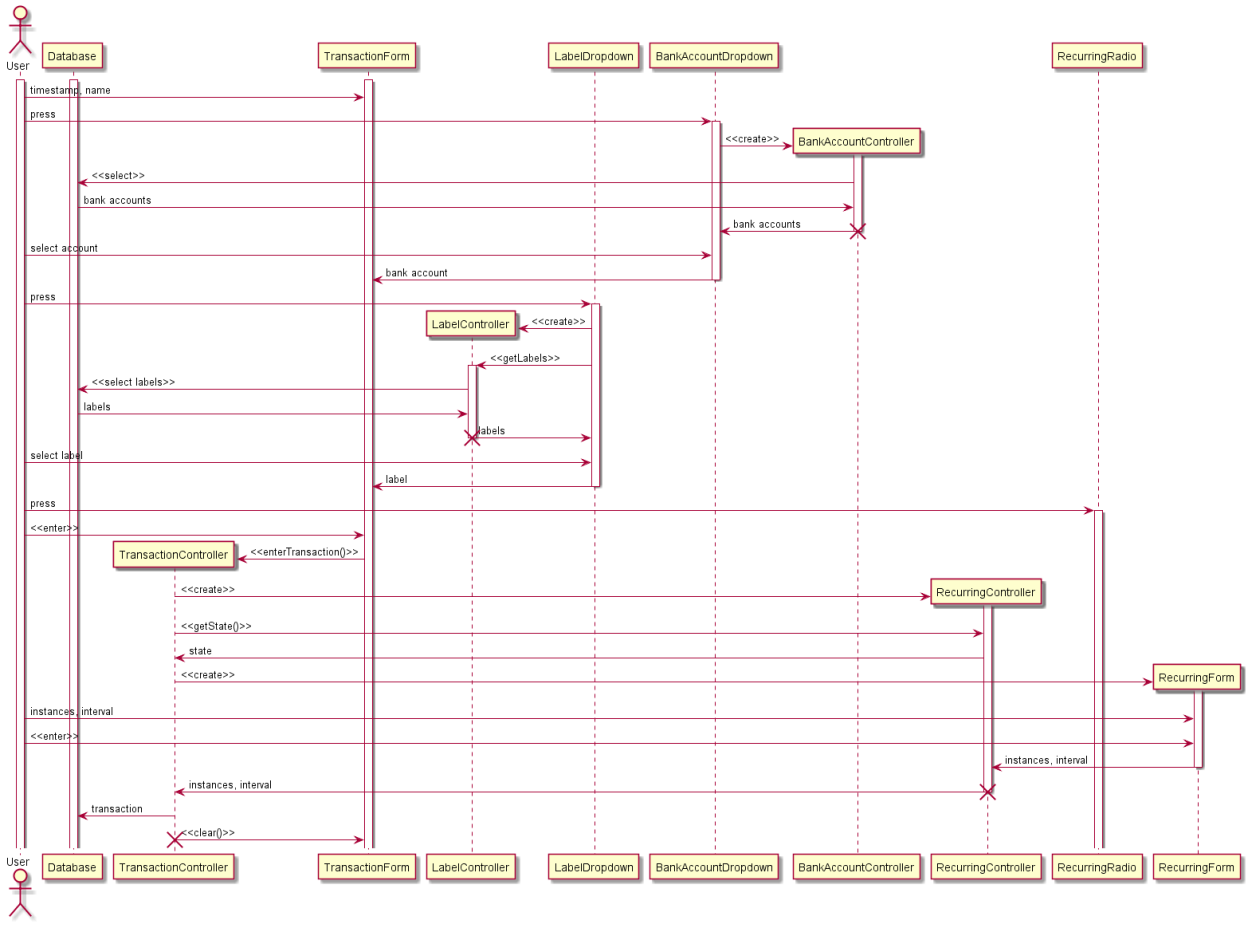
15) User marks that the transaction is recurring.

16) TransactionController asks the user if the perpetual and gives a field for a number of instances along with a field for the interval.

17) User enters the information and presses “create transaction”.

18) TransactionController adds the transaction to Database.

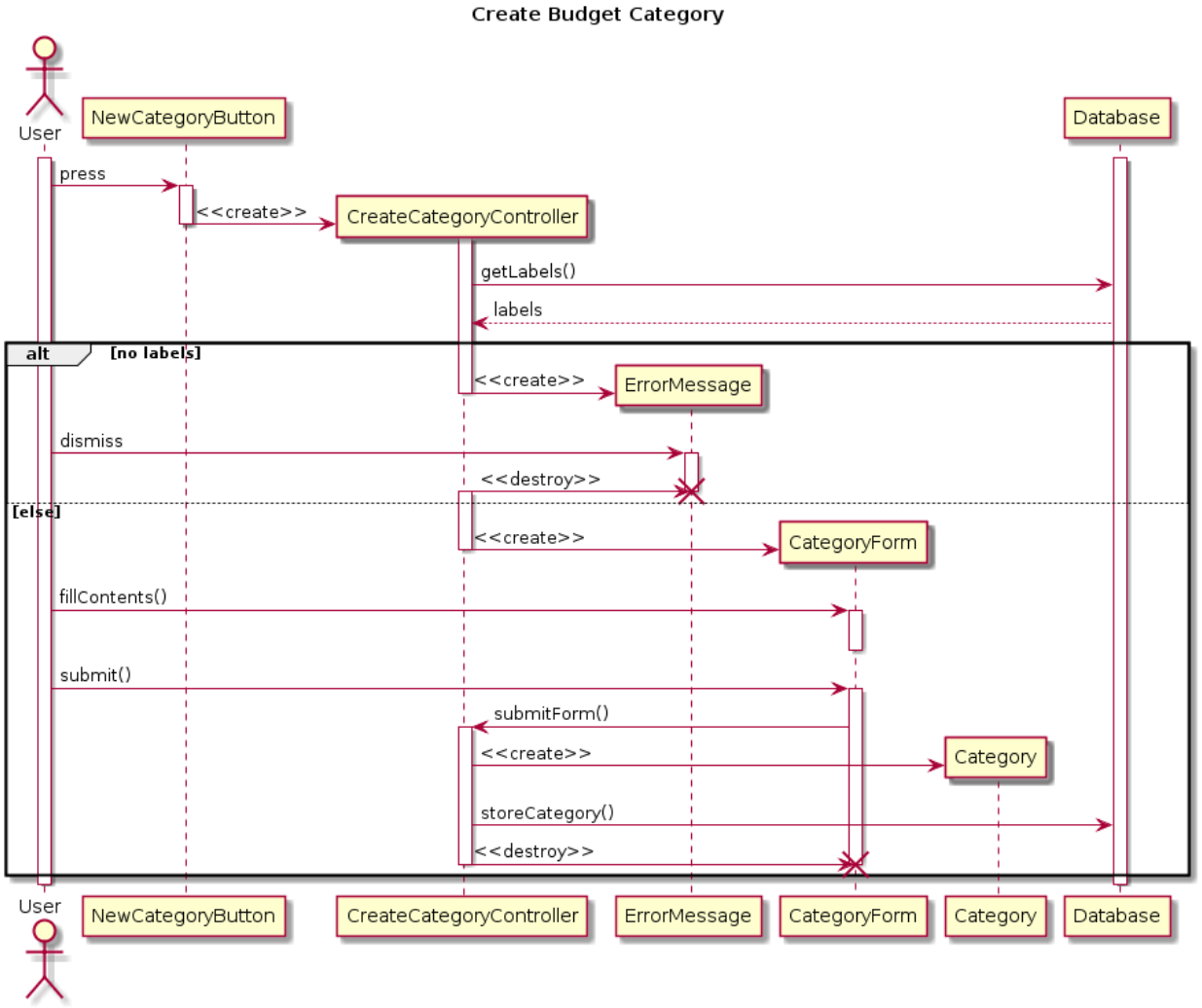
Exit Condition:	19) TransactionController displays the transaction in the list.



Use case name:	Create Budget Category

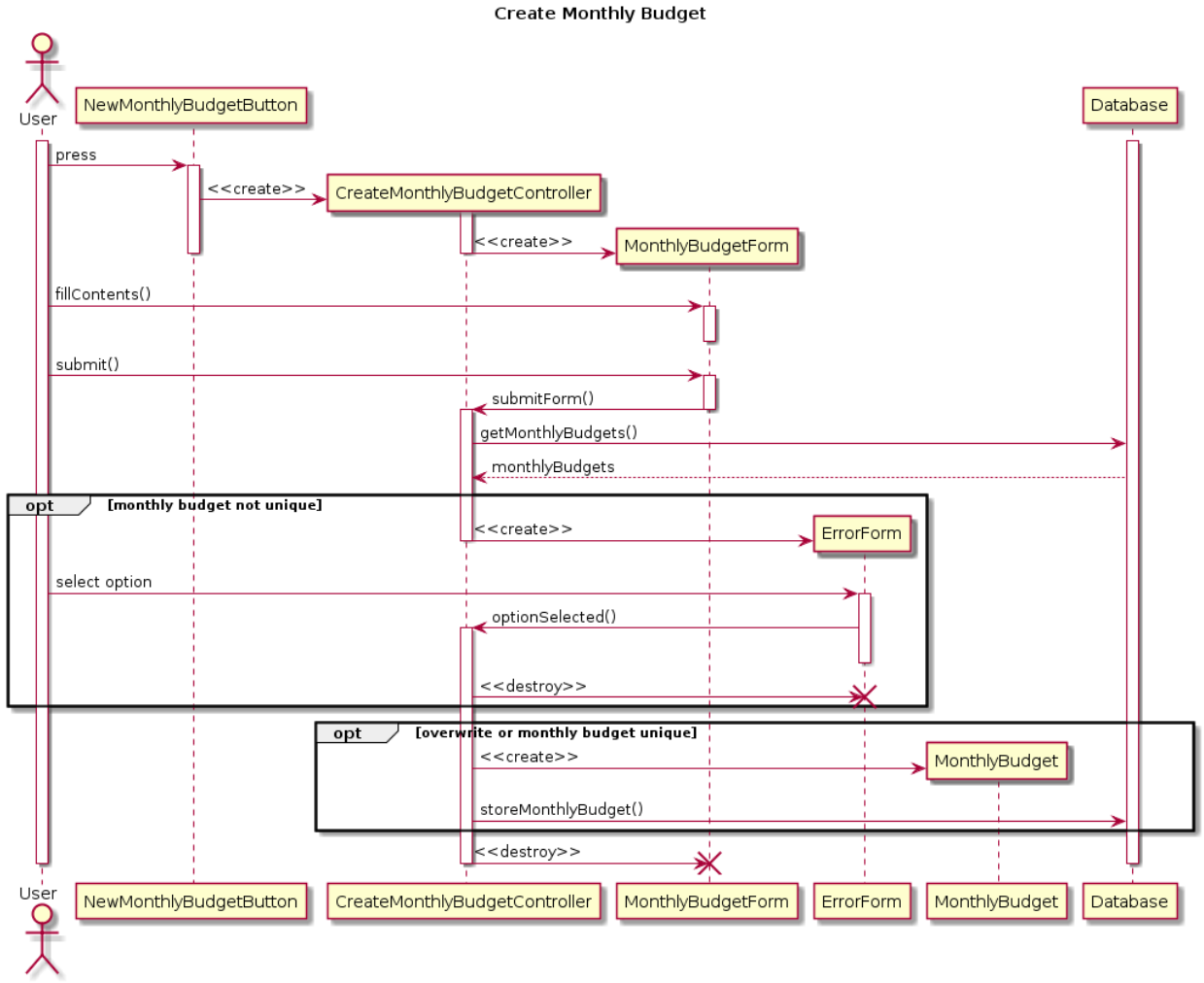
Entry condition:	1. The user is in the Budget Tab, and a MonthlyBudget to add the category to has been selected. The User then presses the NewCategoryButton.
Flow of events:	2. Shark Byte responds by querying the Database for all of the labels. If none are available, Shark Byte displays an ErrorMessage telling the User to create at least one label before creating a Category and the CreateBudgetCategory use case ends. 3. Shark Byte then presents a CategoryForm containing a drop-down menu of the available labels and fields for the name and price limit of the Category. The user selects a label and enters the fills the fields for the name and price limit, then submits the form. 4. Shark Byte then creates the Category from the information in the form and stores the Category in the Database, overwriting any other Category sharing the same label and MonthlyBudget.
Exit condition:	5. The Category has been successfully added to the MonthlyBudget and stored in the Database

Detailed System Sequence Diagram:



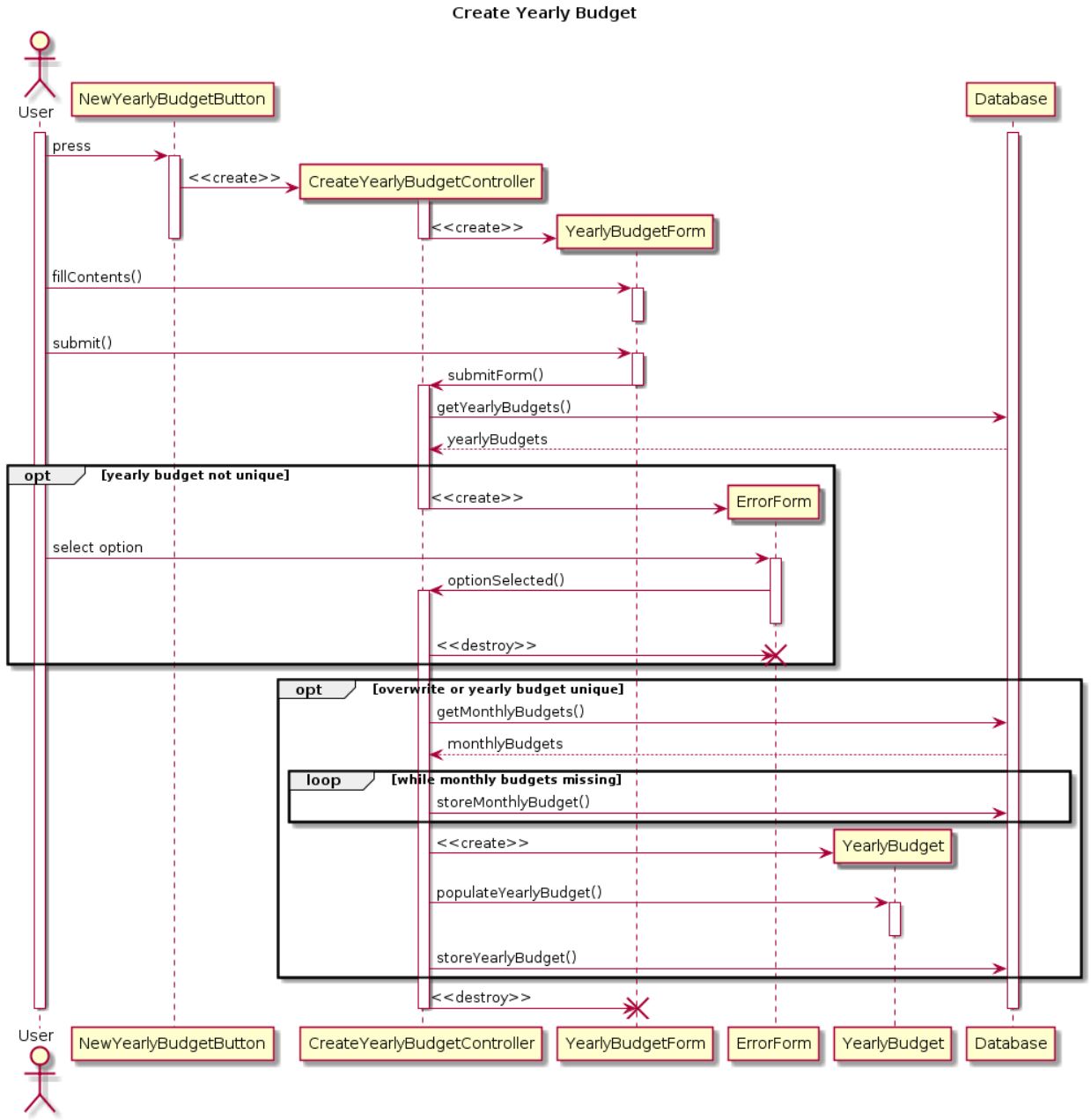
Use case name:	Create Monthly Budget
Entry condition:	1. The user is in the Budget Tab, and the User has pressed the NewMonthlyBudgetButton.
Flow of events:	2. Shark Byte responds by displaying a MonthlyBudgetForm to the User containing fields for the month and year for the MonthlyBudget. The User fills both fields in the MonthlyBudgetForm and then submits the form. 3. Shark Byte queries the Database to check if a MonthlyBudget with that month and year exists in the Database. If the MonthlyBudget exists in the Database (is not unique), then an ErrorForm telling the User that the MonthlyBudget already exists is displayed. The User may choose to overwrite or to cancel the CreateMonthlyBudget use case. 4. If the User chose to overwrite the MonthlyBudget or if the MonthlyBudget is unique, then Shark Byte creates the MonthlyBudget and stores it in the Database.
Exit condition:	5. The MonthlyBudget has been successfully created and added to the Database.

Detailed System Sequence Diagram:



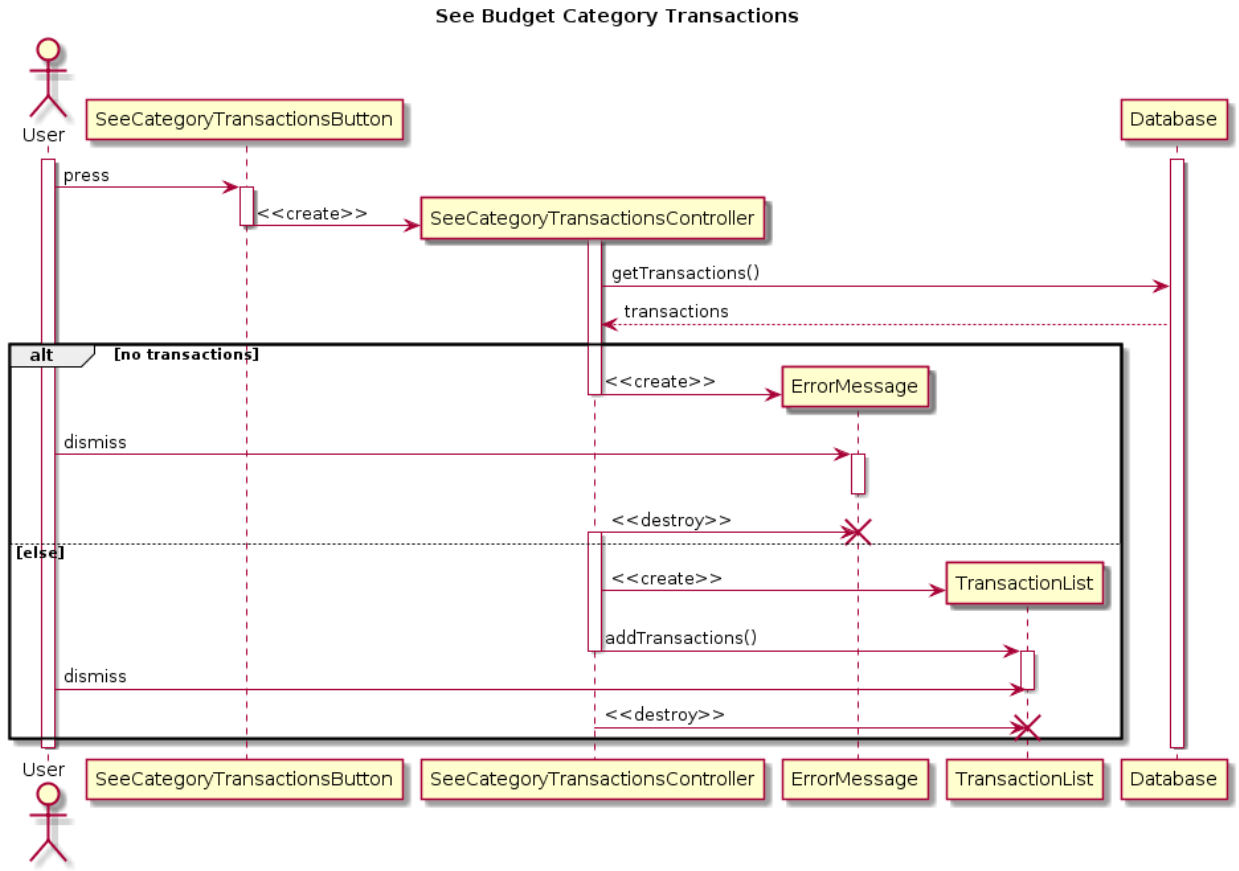
Use case name:	Create Yearly Budget
Entry condition:	1. The user is in the Budget Tab, and the User has pressed the NewYearlyBudgetButton.
Flow of events:	2. Shark Byte responds by displaying a YearlyBudgetForm to the User containing a field for the year of the YearlyBudget. The User then fills in the year field of the form. 3. Shark Byte queries the Database to check if a YearlyBudget with that year exists in the Database. If the YearlyBudget exists in the Database (is not unique), then an ErrorForm telling the User that the YearlyBudget already exists is displayed. The User may choose to overwrite or to cancel the CreateYearlyBudget use case. 4. If the User chose to overwrite the YearlyBudget or if the YearlyBudget is unique, then Shark Byte queries the Database for MonthlyBudgets sharing a year with the YearlyBudget. If MonthlyBudgets are missing, then the YearlyBudget is populated with MonthlyBudgets for the missing months (MonthlyBudgets are stored in the Database). Shark Byte then stores the YearlyBudget in the Database.
Exit condition:	5. The YearlyBudget has been created and successfully stored in the Database.

Detailed System Sequence Diagram:



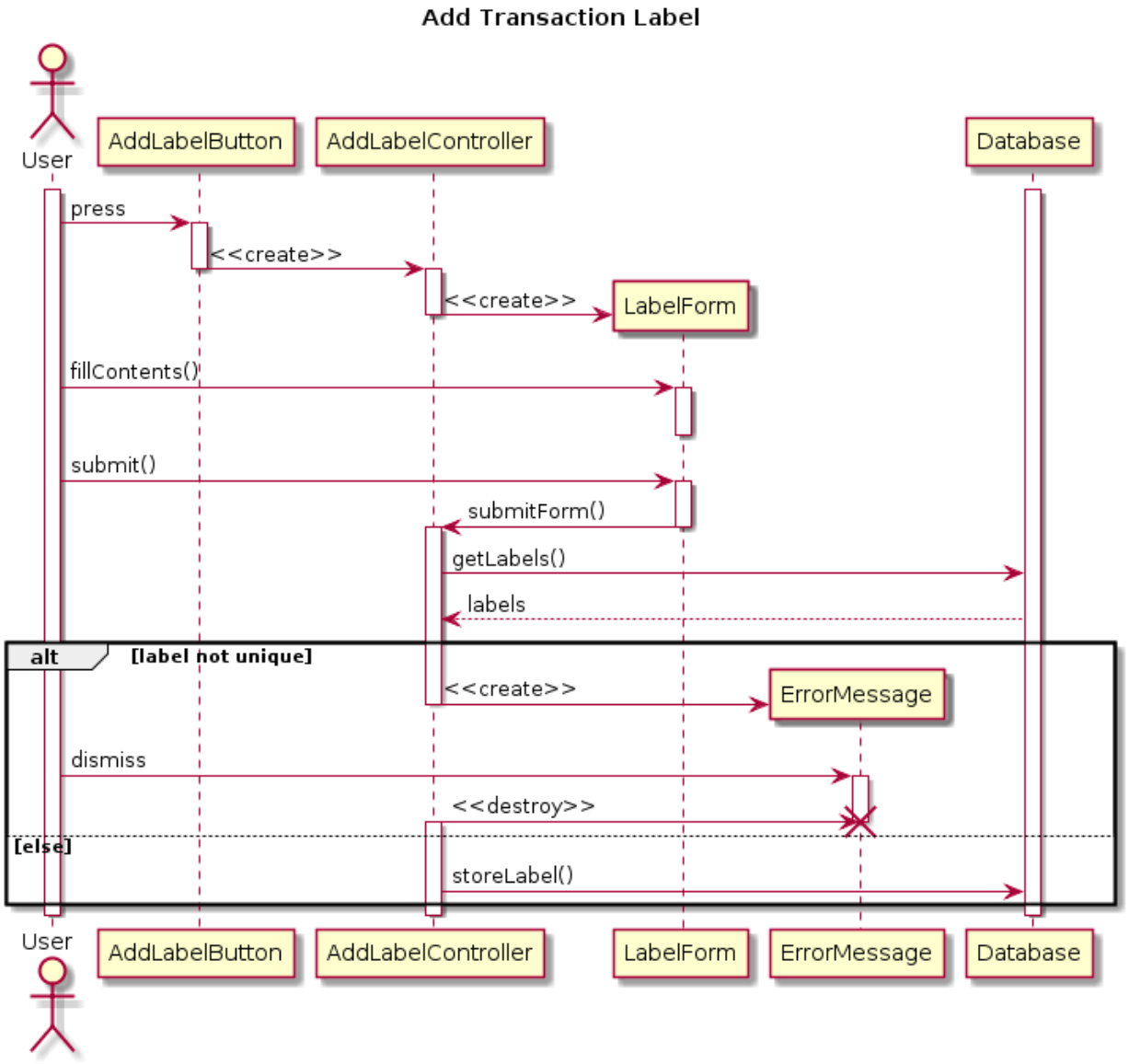
Use case name:	See Budget Category Transactions
Entry condition:	1. The User is in the Budget Tab, and the User has selected a Category of a budget. The User then presses the SeeCategoryTransactionsButton.
Flow of events:	2. Shark Byte queries the Database for transactions belonging to the selected budget Category (same label, month, and year). The Database then provides the transactions. 3. If no such transactions exist, an ErrorMessage is displayed to notify the User. The use case ends when the user dismisses the ErrorMessage. 4. If transactions do exist, a TransactionList containing the transactions is displayed to the User. The use case ends when the User dismisses this list.
Exit condition:	5. The User has successfully viewed the transactions belonging to a budget Category.

Detailed System Sequence Diagram:



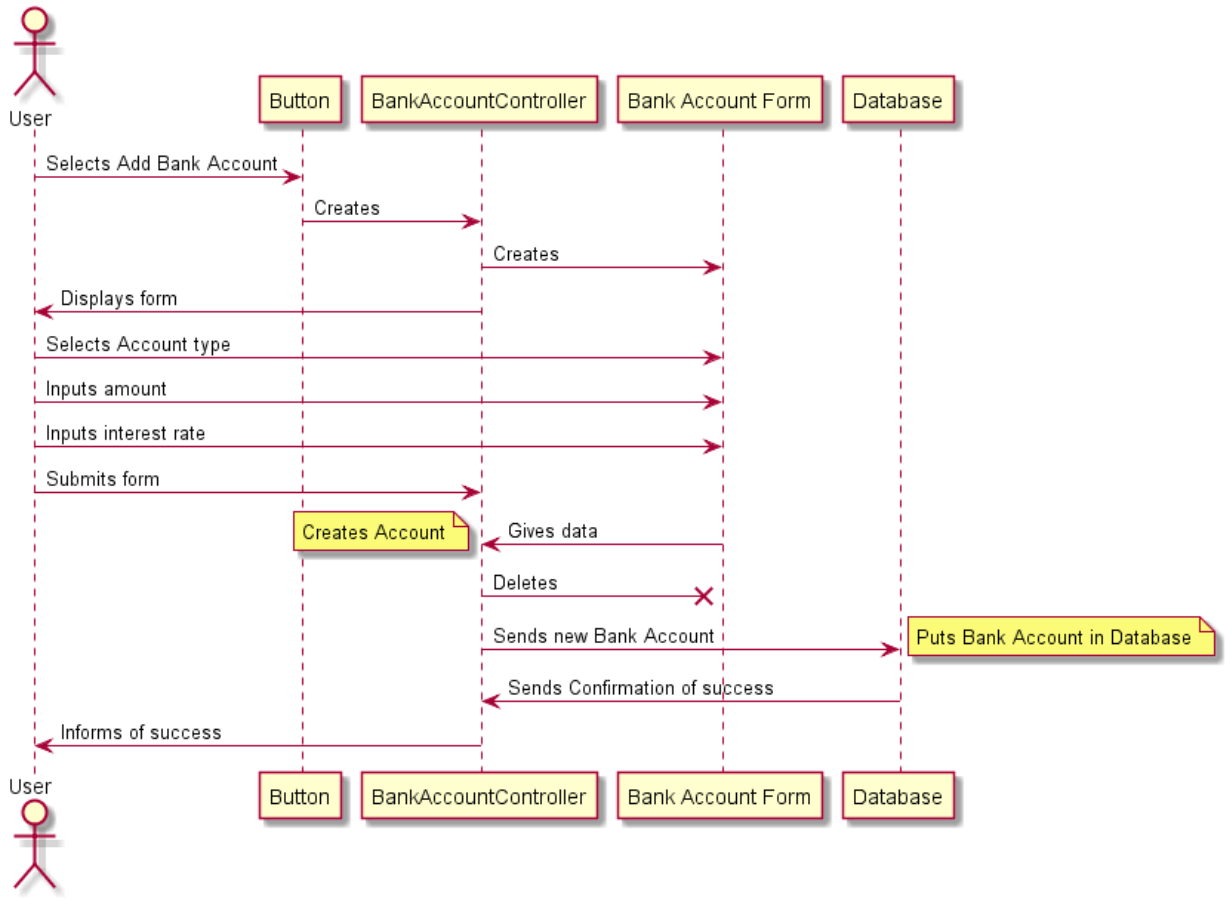
Use case name:	Add Transaction Label
Entry condition:	1. The User is in the Transactions Tab, and the User has pressed the AddLabelButton.
Flow of events:	2. Shark Byte creates a LabelForm containing a field for the name of the label. The User enters the name field and then submits the LabelForm. 3. Shark Byte queries the Database for a label sharing the name entered by the User. If such a label exists, then an ErrorMessage is displayed notifying the User the label already exists. The use case ends when the User dismisses the ErrorMessage. 4. If the label did not exist, then Shark Byte stores the label in the Database.
Exit condition:	5. The User has successfully created the label and the label has been stored in the Database.

Detailed System Sequence Diagram:

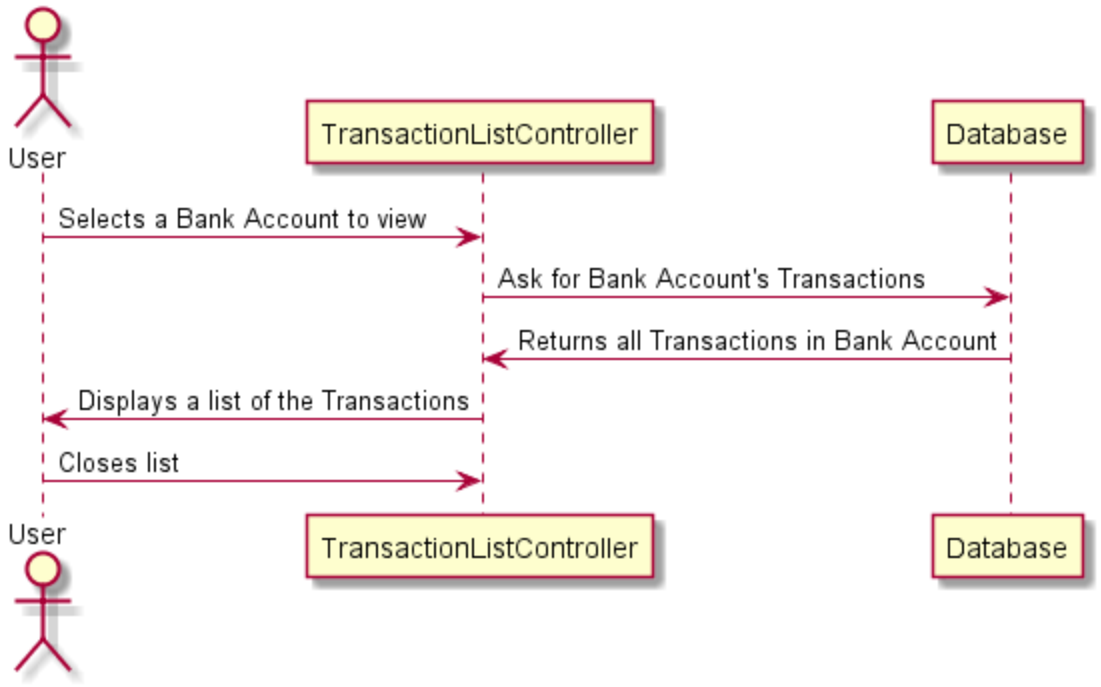


Use Case Name:	Add Bank Account
Entry Condition:	1) The User is currently viewing the Bank Account Tab and selects "Add Bank Account" button.
Flow of Events:	2) The BankAccountController displays a Bank Account form that the User must fill out. This form includes a box to select the bank account type and two input boxes for the amount and the interest rate. 3) The User selects the account type and inputs the amount and interest rate. The user then submits the form. 4) The BankAccountController checks that the amount and interest are valid.

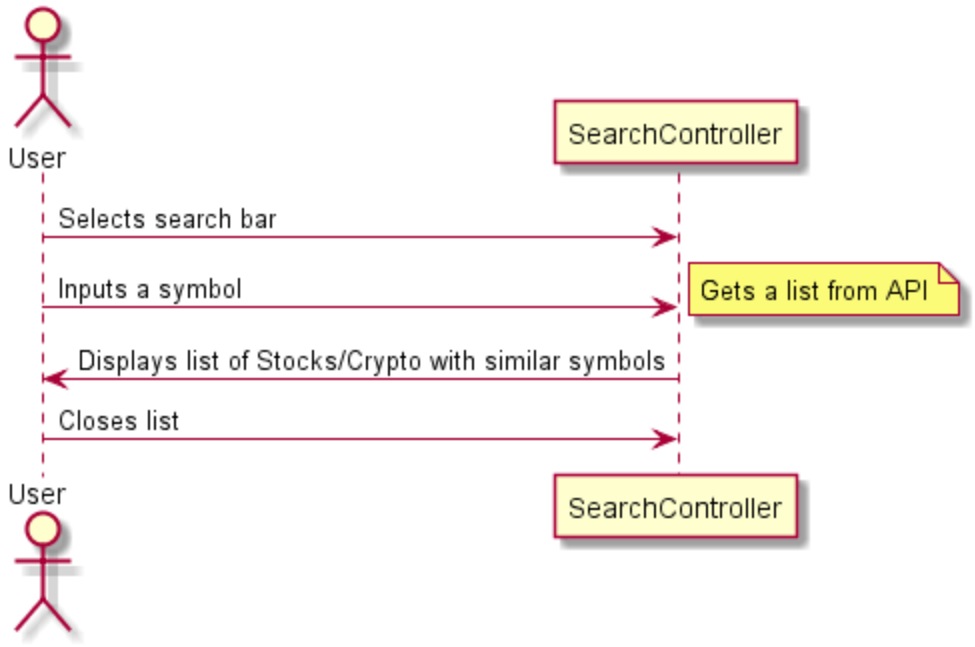
	5) The BankAccountController creates a bank account and sends it to the Database. 6) The Database adds this bank account to the User's bank account list and sends the BankAccountController a confirmation that is succeeded.
Exit Condition:	7) The BankAccountController informs the User that the addition was successful and adds the new bank account to the ones displayed on the Bank Account Tab.



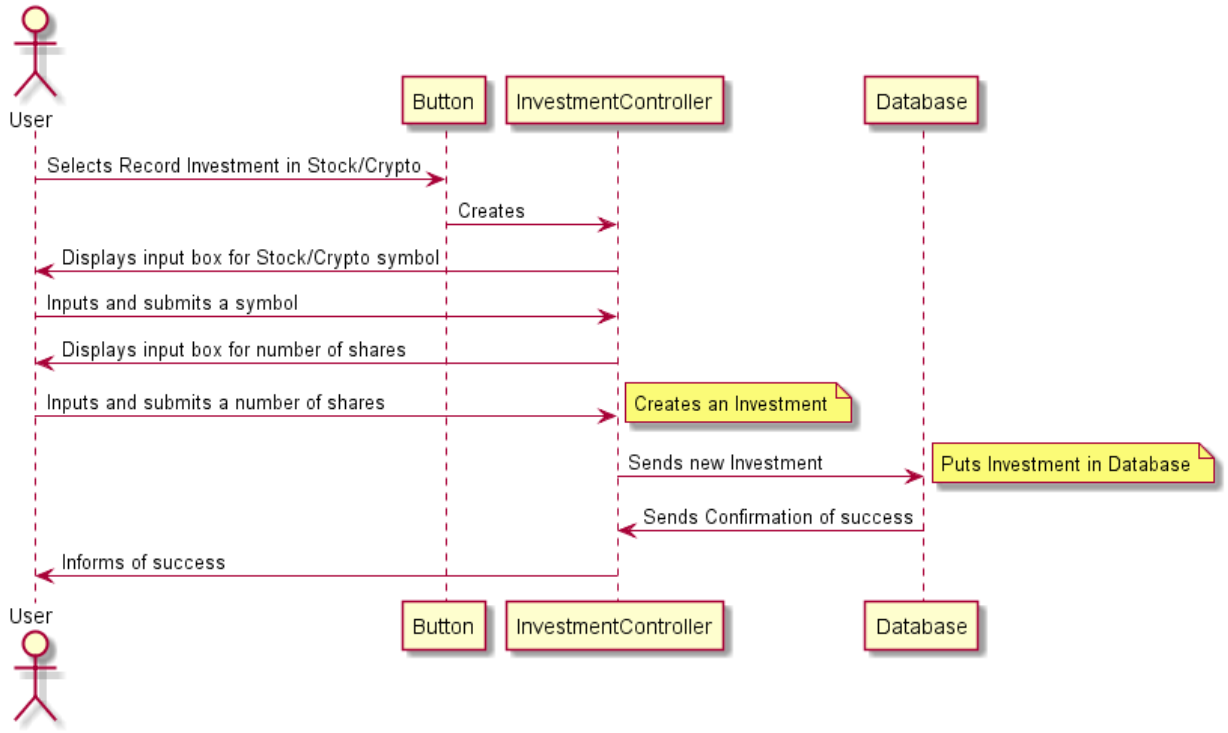
Use Case Name:	See Bank Account Info
Entry Condition:	1) The User is currently viewing the Bank Account Tab, selects a Bank Account, and chooses to view the Bank Account Info.
Flow of Events:	2) The TransactionListController asks the Database for the Bank Account and its' transactions. 3) The TransactionListController then displays the list of the Bank Account's transactions.
Exit Condition:	4) The User looks at and closes the list when they are done.



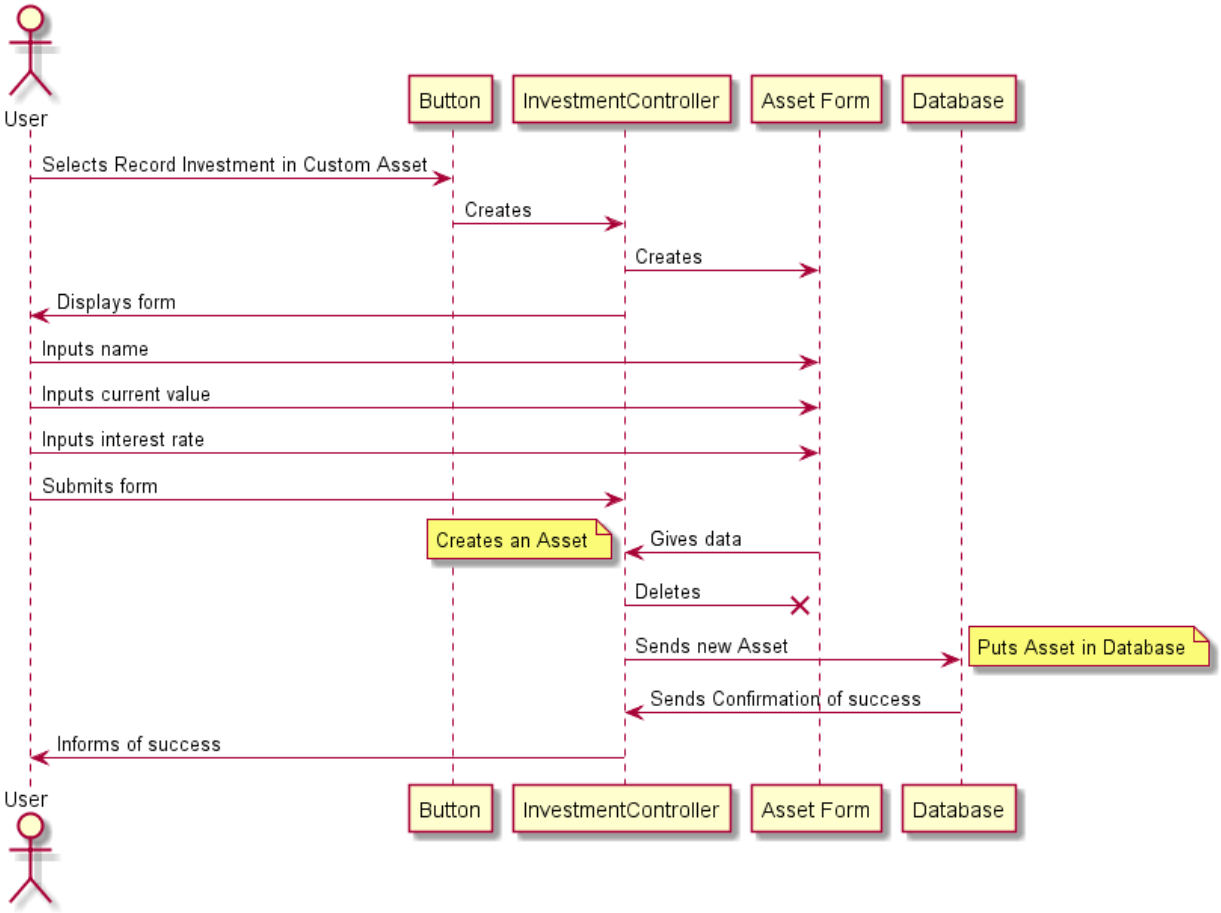
Use Case Name:	Look up Stock/Crypto
Entry Condition:	1) The User is currently in the Investment Tab and selects the search bar.
Flow of Events:	2) The User types in the symbol for the Stock/Crypto they want to look up. 3) The SearchController displays a list of Stocks/Crypto with similar symbols to the input.
Exit Condition:	4) The User looks at and closes the list when they are done.



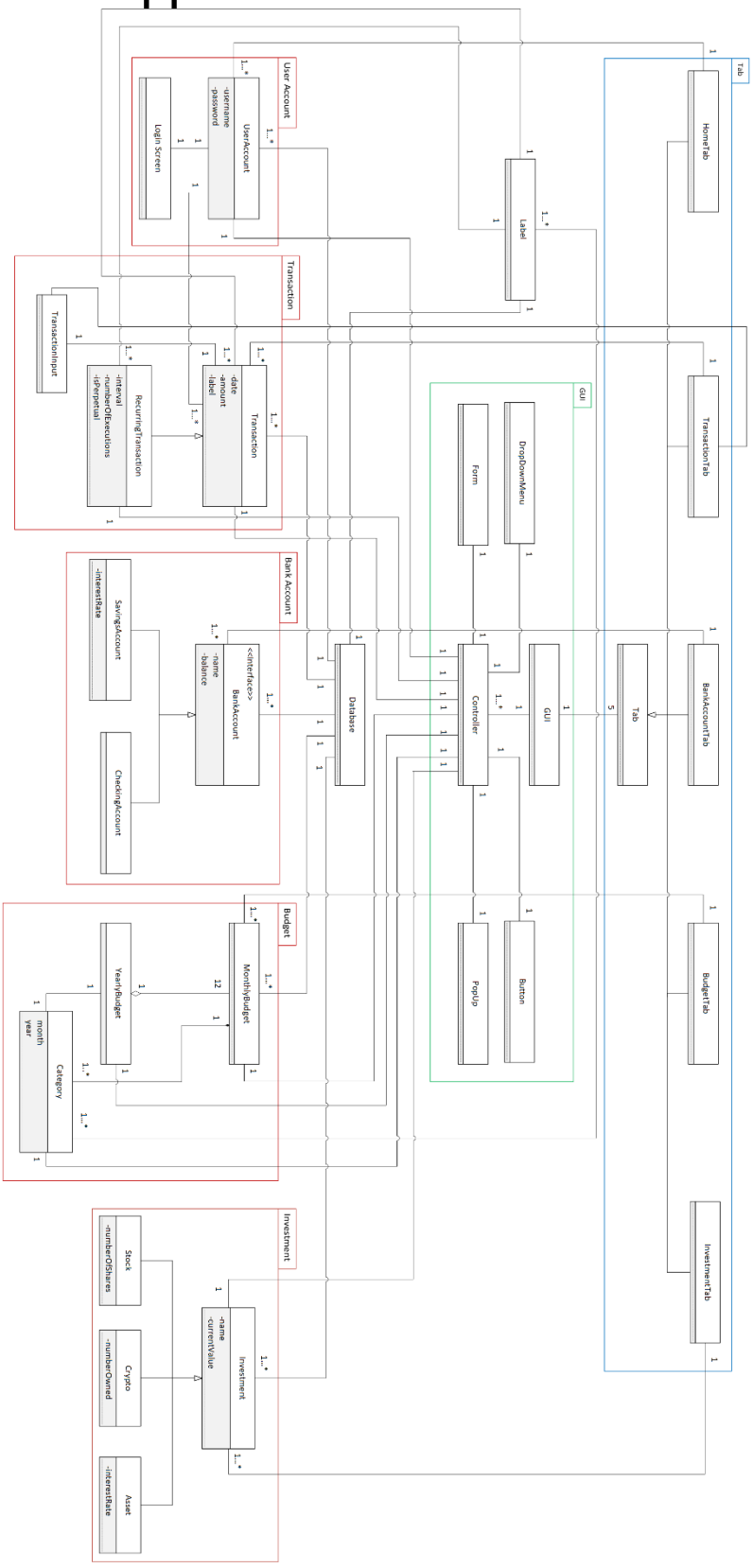
Use Case Name:	Record Investment in Stock/Crypto
Entry Condition:	1) The User is currently in the Investment Tab and selects "Record Investment in Stock/Crypto" button.
Flow of Events:	2) The InvestmentController displays an input box for the User to input a symbol. 3) The User inputs a symbol and submits it. The InvestmentController then verifies the symbol exist and displays another input box for the number of Stock/Crypto the User has. 4) The User inputs a number and submits it. The InvestmentController checks if it is a valid number. 5) The InvestmentController creates a new Investment with the information and sends the Investment to the Database. 6) The Database adds the Investment to the User's Investment list and sends a confirmation back to the InvestmentController.
Exit Condition:	7) The InvestmentController informs the User that the addition was successful and adds the Investment to the owned Investment list displayed on the Investment Tab.



Use Case Name:	Record Investment in Custom Asset
Entry Condition:	1) The User is currently in the Investment Tab and selects "Record Investment in Custom Asset" button.
Flow of Events:	2) The InvestmentController displays a form for the User to input a custom asset. 3) The User inputs the asset's current value, name, and interest rate. 4) The User submits the form, and the InvestmentController checks if all the information is valid. 5) The InvestmentController creates a new Asset with the information and sends the Asset to the Database. 6) The Database adds the Asset to the User's Investment list and sends a confirmation back to the InvestmentController.
Exit Condition:	7) The InvestmentController informs the User that the addition was successful and adds the Asset to the owned Investment list displayed on the Investment Tab.


User
Button
InvestmentController
Asset Form
Database
Selects Record Investment in Custom Asset
Creates
Creates
Displays form
Inputs name
Inputs current value
Inputs interest rate
Submits form
Creates an Asset
Gives data
Deletes
Sends new Asset
Puts Asset in Database
Sends Confirmation of success
Informs of success
User
Button
InvestmentController
Asset Form
Database

Application Class Model



Application Class Model OCL Constraints

```
Context BankAccount

        inv: self.classes -> forall (<x, y> | x.name != y.name and x != y)
```

```
Context UserAccount

        inv: self.password.size() > 6
```

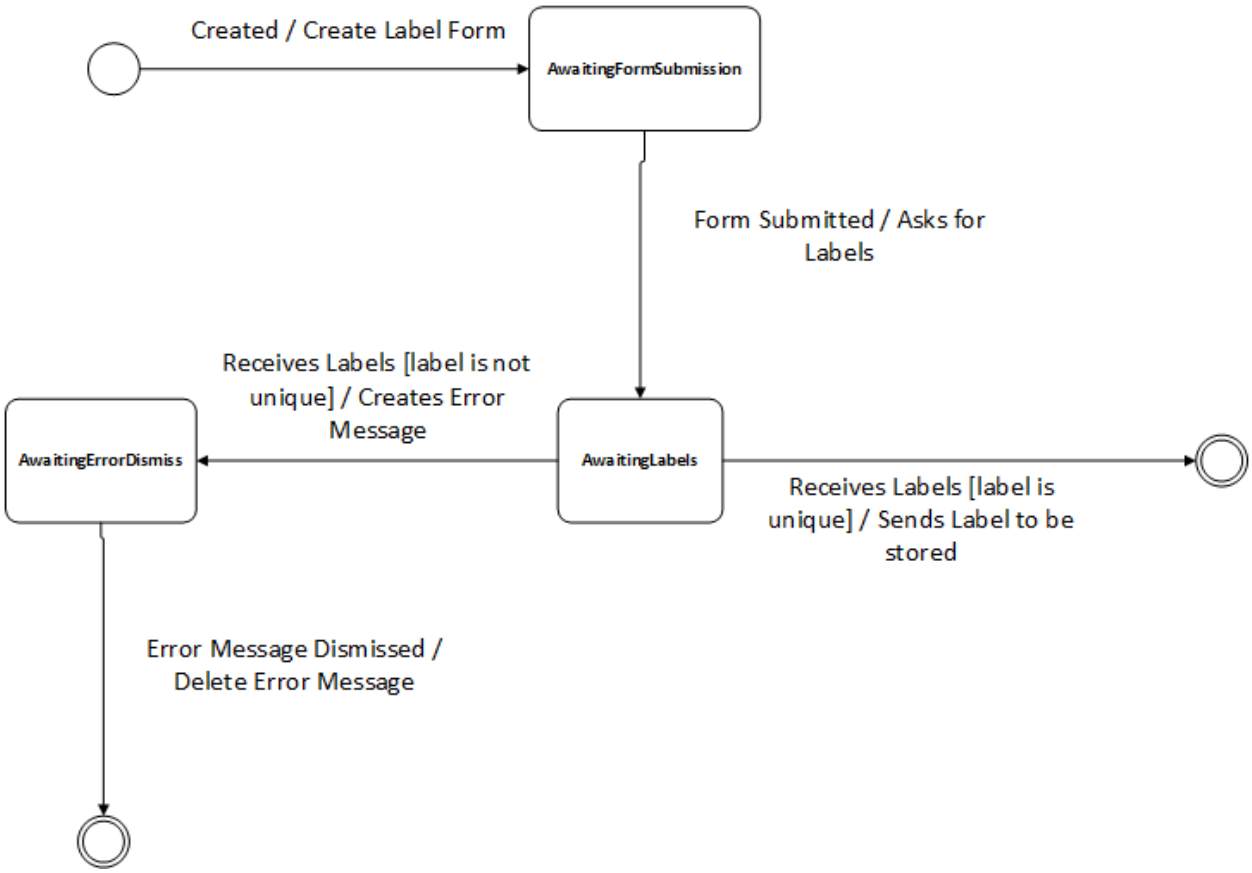
```
Context Investment

        inv: self.name.size() > 0
```

```
Context Investment

        inv: self.currentValue >= 0
```

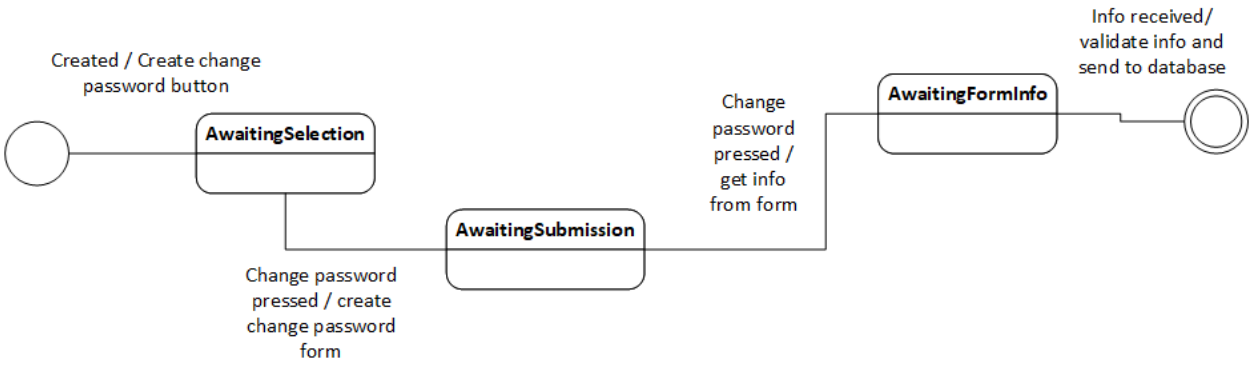
```
Context Crypto

        inv: self.numberOwned > 0
```

Application State Model

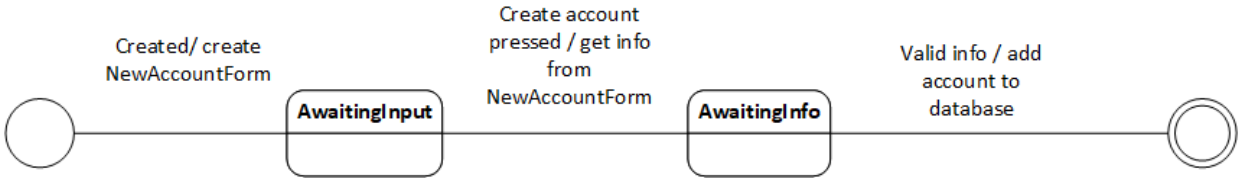
AddLabelController



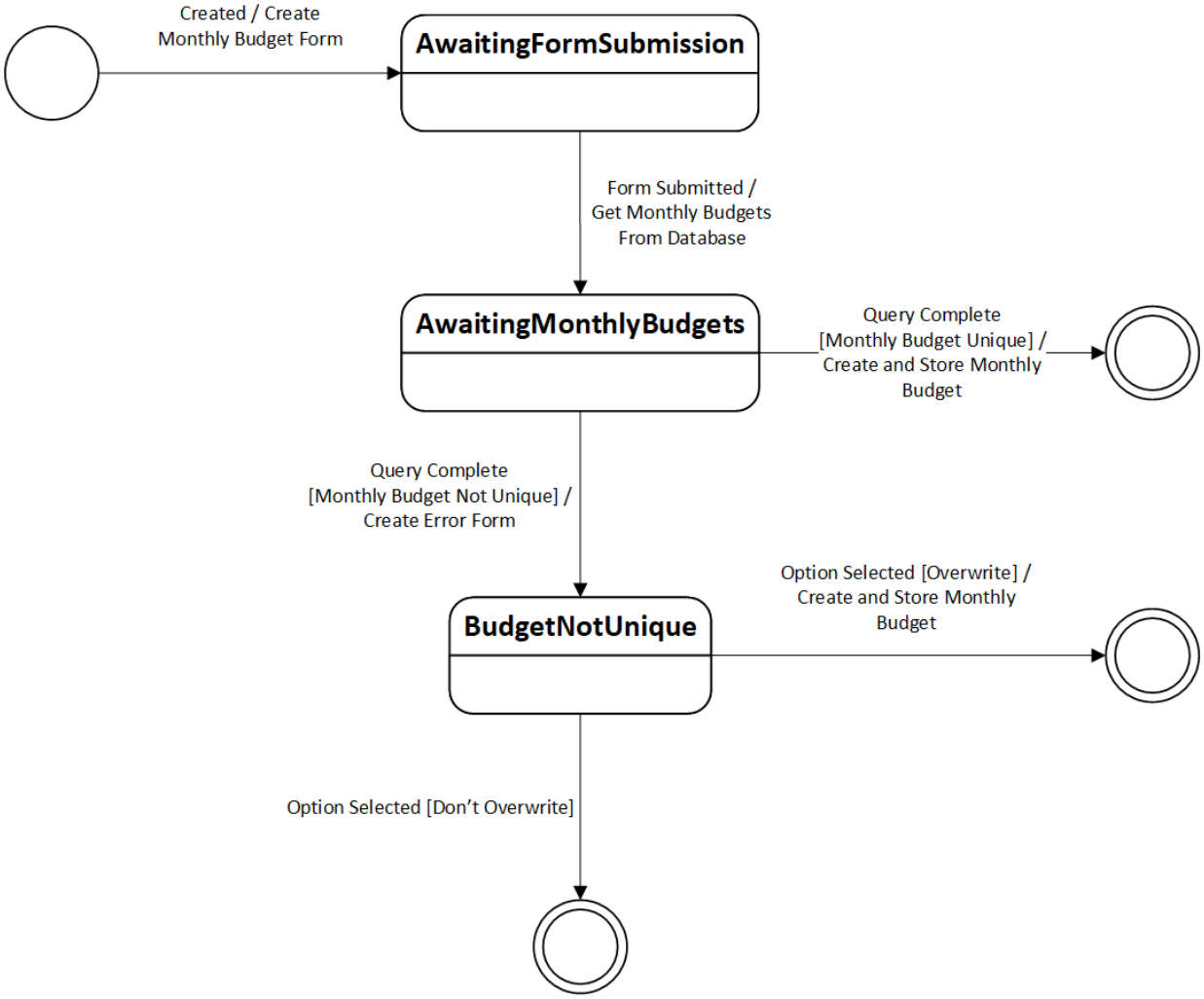
ManageAccountController



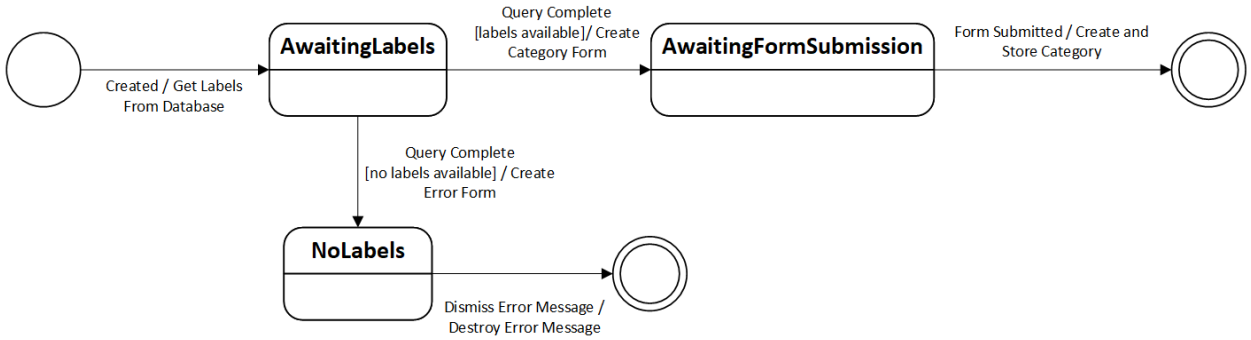
CreateAccountController



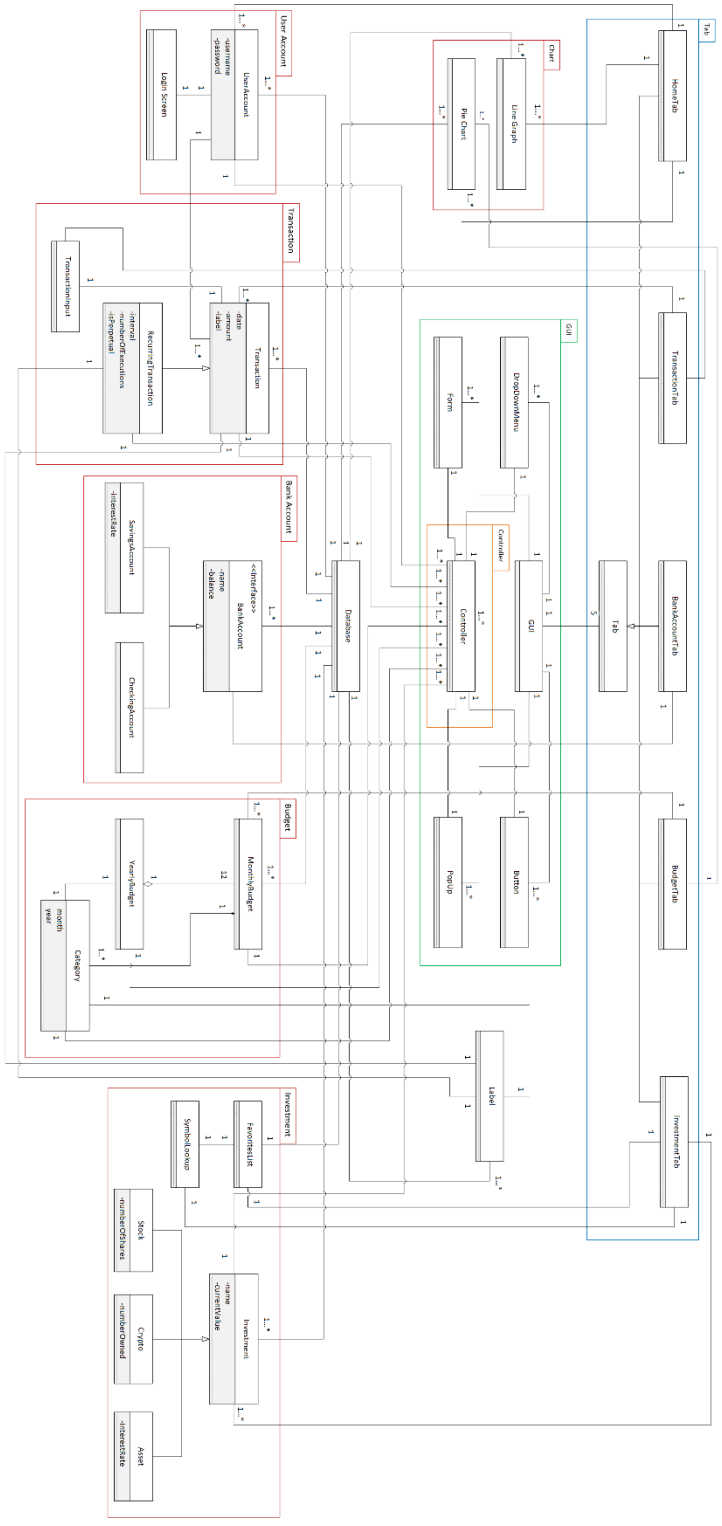
CreateMonthlyBudgetController

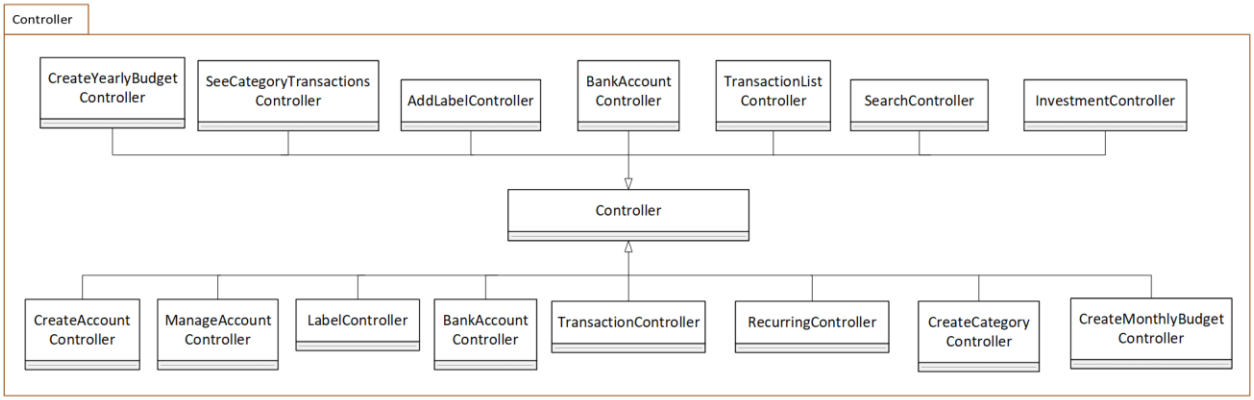


CreateBudgetCategoryController



Consolidated Class Model




Controller
CreateYearlyBudget Controller
SeeCategoryTransactions Controller
AddLabelController
BankAccount Controller
TransactionList Controller
SearchController
InvestmentController
Controller
CreateAccount Controller
ManageAccount Controller
LabelController
BankAccount Controller
TransactionController
RecurringController
CreateCategory Controller
CreateMonthlyBudget Controller

Model Review

The domain class model properly represents the domain described in the concept statement; the domain model covers accounts, budgets, investments, transactions, and labels. The domain class model contains the fundamental components of Shark Byte that were described in the concept statement. Because these concepts in the domain model were derived from the concept statement, if the user were to read the domain model, they would be able to clearly understand the domain model. Additionally, the domain model is consistent with the application model. Although the application model was derived from the use cases, the application model has significant overlap with the domain model because the use cases themselves derive from the concept statement. A couple of objects are not in the application class model because some of the objects were not used in the created use cases, but they are integrated into the consolidated class model. Many of the associations in our class model are used in the transfer of data from the user through the GUI to the database. These associations are traversed when the user interacts with a GUI object in order for data to be sent to the database or for the retrieval of data.

Regarding the completeness of the use cases in our model, the use cases cover all of the major uses of Shark Byte covered in the concept statement; however, one notable use case that is excluded from out model is the visualization of a budget with graphs. This was excluded because this use case is not rich in interactions with the user. For the use cases that were modeled, the important error cases were covered. For instance, the use cases cover when an incorrect password is entered, or if the user attempts to add data that already exists in the database. However, any interaction that was not covered by a use case is still represented in the consolidated class model.

The detailed system sequence diagrams are consistent with the high-level system sequence diagrams. The interactions crossing the system boundary in the high-level system sequence diagrams are represented in the detailed system sequence diagrams as user interactions with boundary objects. Due to the scale of some of the detailed system diagrams, some of the alternative flows were intentionally omitted to retain readability of the graphics.

The biggest weakness in our analysis is the inclusion of the database in our models. We feel that this was a necessity due to how the user would need to store information that would be required by many other objects; however, this decision does restrict us to the repository architectural design pattern. Throughout the analysis phase we followed a couple different models: using manager classes to handle I/O of their relevant data to the database, and using controllers for GUI objects which directly interact with the database. Using controllers attached to our GUI objects means that we'll have some hybrid of the model-view-controller model and the repository model. We also found that we naturally needed many software objects, which meant large, complex class diagrams. Other than this, our models focus very little on implementation, meaning the designers will have more freedom to make important design decisions.

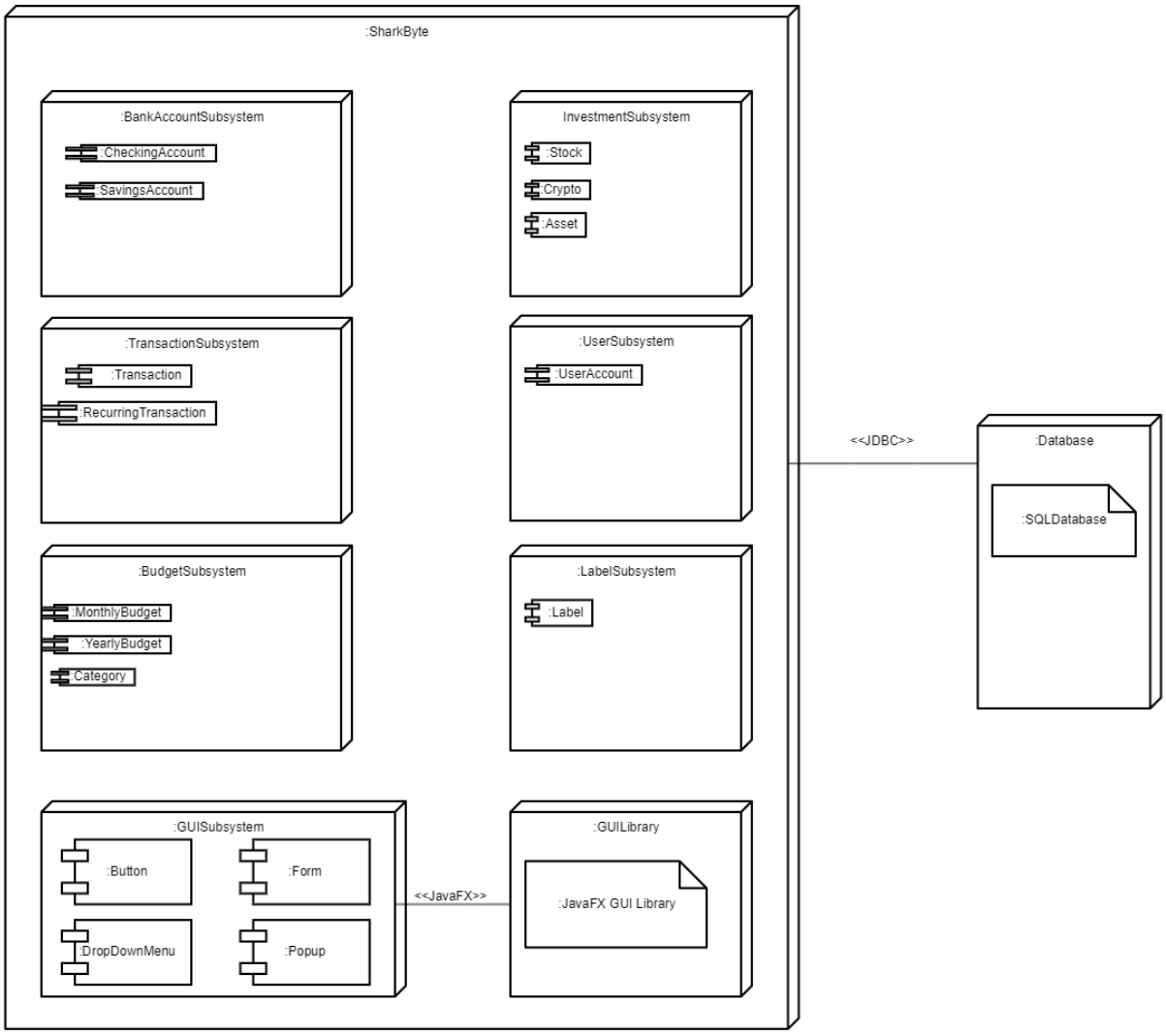
In addition, since we followed the repository model, all of our GUI rendering will require queries to the database for the information to display. Also, it may not be the most effective to represent investments, transactions, and bank accounts as software objects, since they all exist as rows of information within a database. Manipulating data in the database means that these objects will need to be reconstructed/modified based on any changes to the

repository. The objects themselves might not serve any real purpose since data flows directly between the database and GUI, rather than interacting with separate objects. While helpful for understanding what our data is and how it works, we will likely not implement these concepts as software objects in the design phase. Also, over time, the database will contain a potentially lengthy list of transactions. This means that searching the database for specific transactions by date, label, or bank account might become an inefficient process.

PHASE II - DESIGN

Architectural Design

UML Deployment Diagram



Deployment Diagram Explanation

In the Deployment Diagram for Shark Byte financial suite, we demonstrated the existence of eight different subsystems. Each use case was mapped to a more general subsystem; such as "Create User Account" being allocated to the "User" subsystem and "Create Bank Account" being allocated to the "Bank Account" subsystem. With this in mind, we created subsystems for Bank Account, Investment, Transaction, User, Budget, Label, GUI and GUI Library.

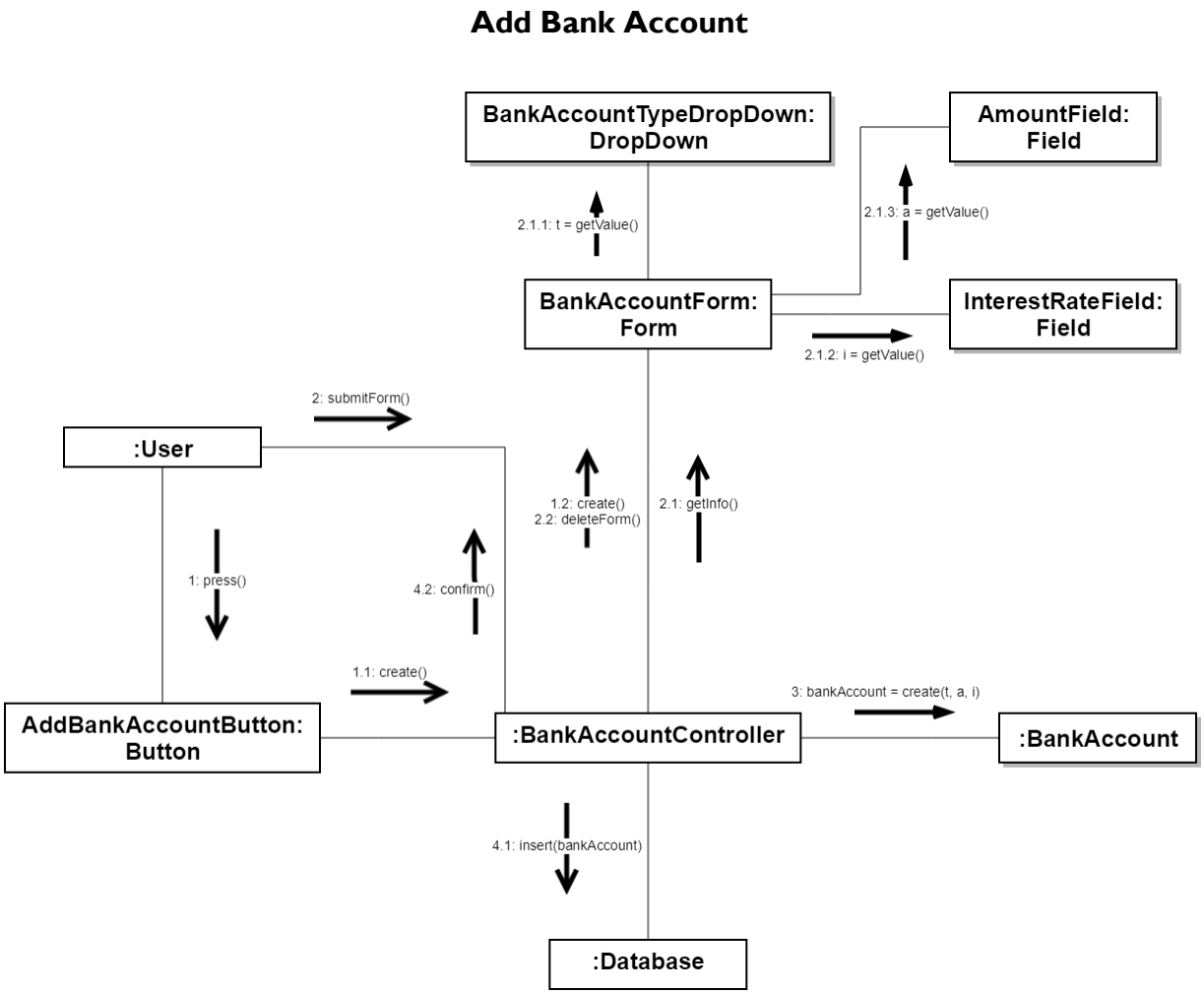
In our model, every subsystem will directly access the database. Because of this, we have selected the **Repository Model** as the primary architectural style for Shark Byte.Through the Repository Model, the subsystems will be able to efficiently share data while storing and retrieving information will be trivial. But, the Repository Model will mandate that all subsystems agree upon how to access the database and changing data within the database will be costly.

Along with the Repository Model, we have decided to use aspects of a **Model-View-Controller** Architecture as well since Shark Byte will be heavily dependent on a GUI. The database information will be contained in the model, and the controllers will be able to handle all requests from the user. Though the MVC Architecture will take a bit of time to implement, we believe it will compartmentalize the program, making it easier to test, alter, and maintain.

Detailed Design

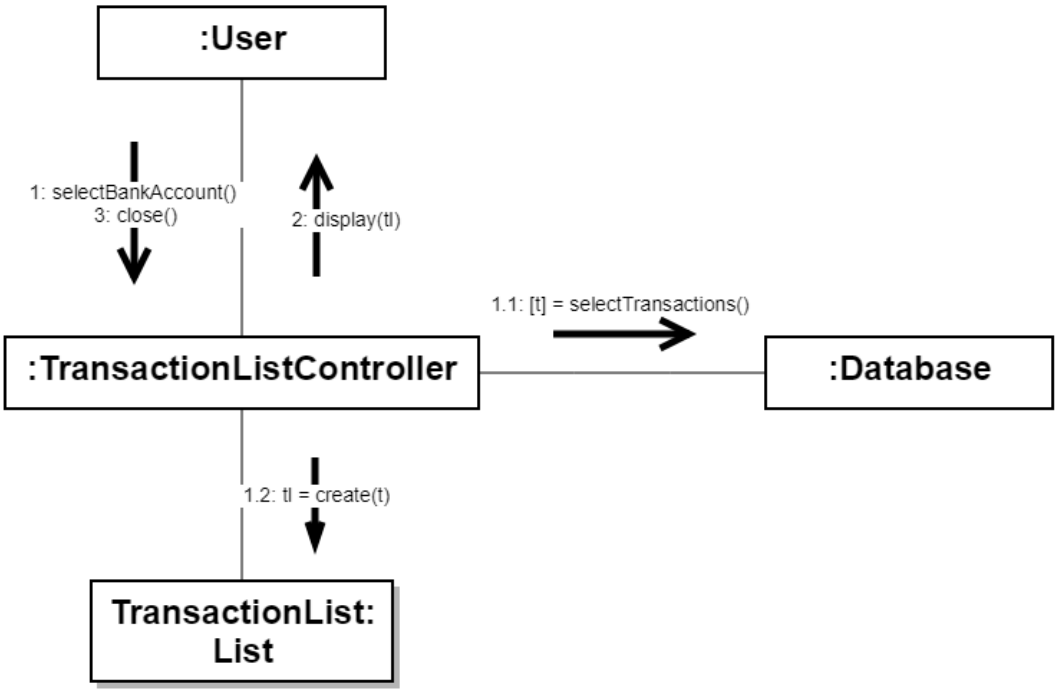
Interaction Design

Collaboration Diagrams:



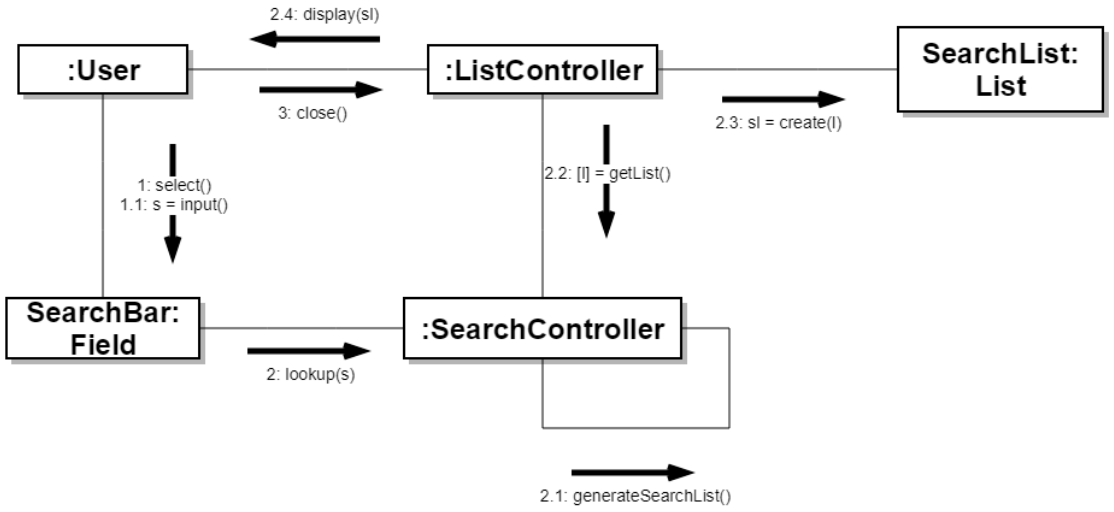
The BankAccountController lets the user interact with the system and database so they can change the data in the way they want. Therefore, BankAccountController uses a **controller** pattern. It also creates a form and a bank account after the form is filled, so it also uses the **creator** pattern.

See Bank Account Info



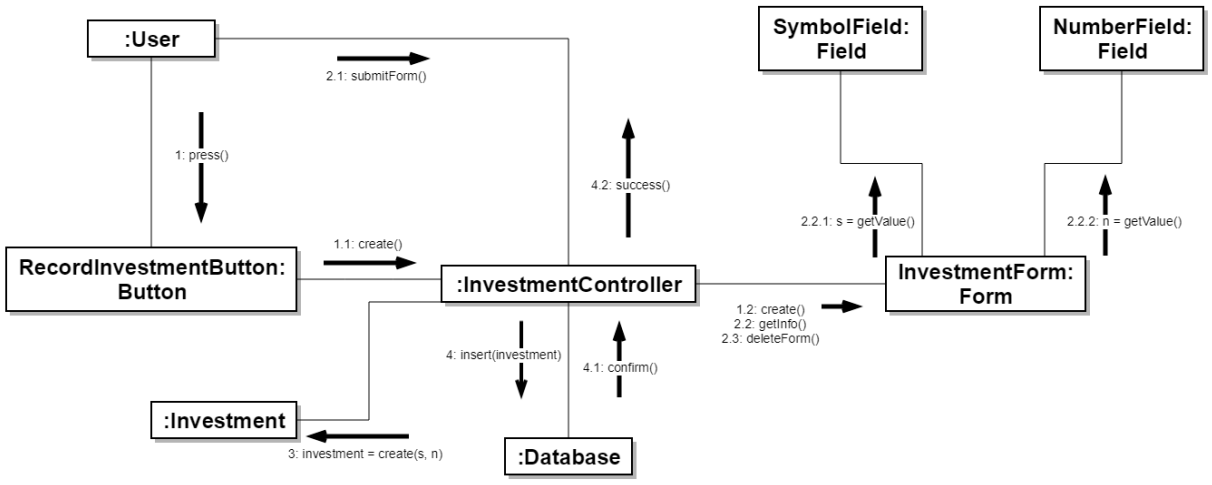
The TransactionListController displays a list of transactions from a bank account the user selects. Since the class name represents what the use case does, it is a **controller** pattern. It also creates a List of transactions making is also use the **creator** pattern. TransactionListController has **high cohesion** because it's only purpose is to take the transactions from the database and displays them for the user.

Lookup Stock or Crypto



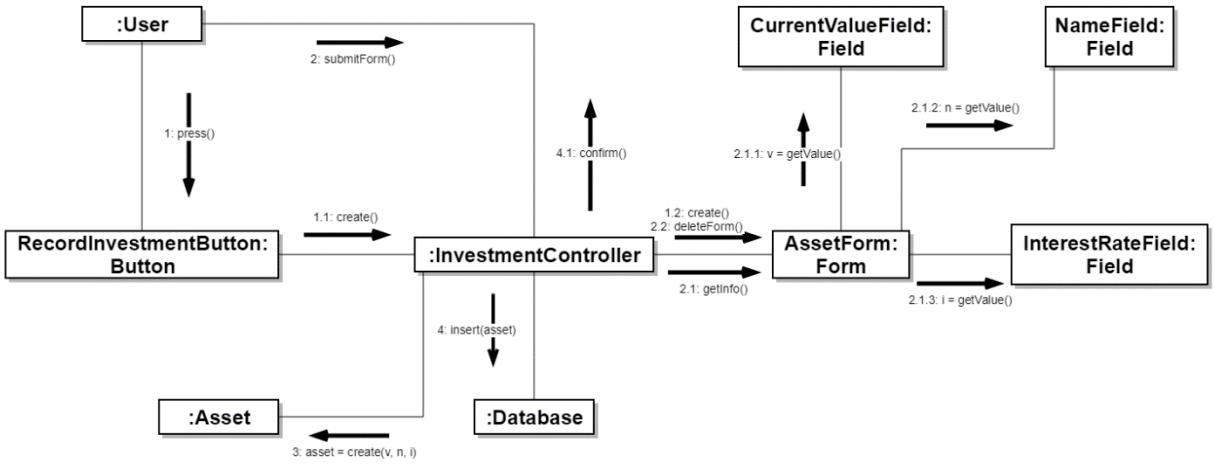
The SearchController looks up the symbol that the user inputs into the UI. Since the class name represents what the use case does, it is a **controller** pattern. The SearchController sends the list of the search items to a ListController that handles the displaying of the information. Thus, this case uses an **indirection** pattern. The ListController creates a List using a **creator** pattern.

Record Investment in Stock or Crypto



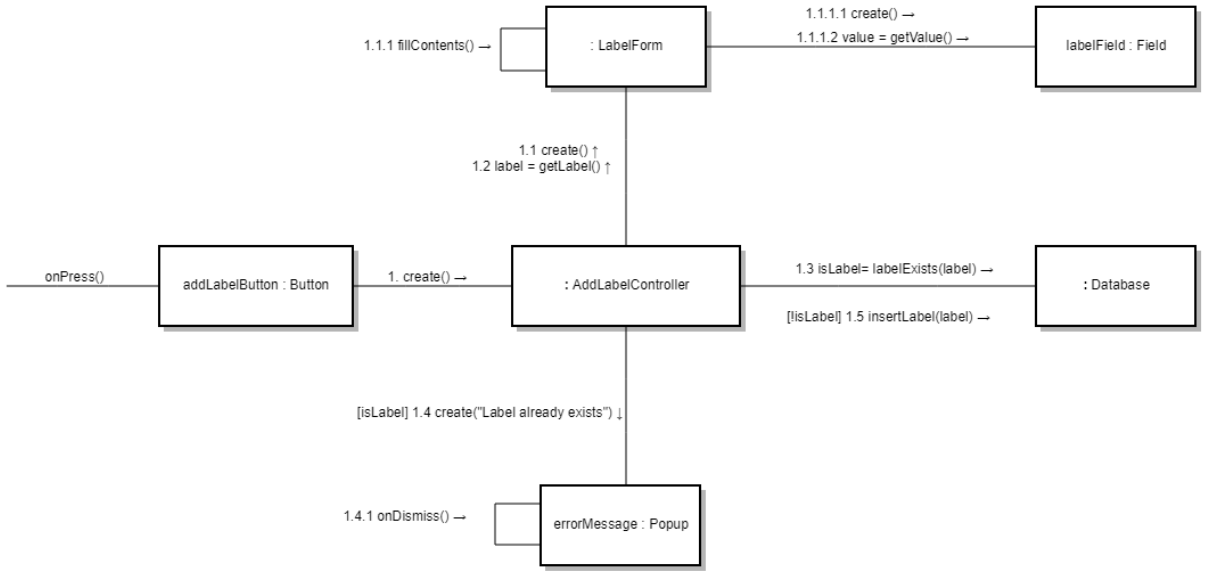
The InvestmentController lets the user interact with the system and database so they can change the data in the way they want. Therefore, InvestmenttController uses a **controller** pattern. It also creates a form and a stock or crypto after the form is filled, so it also uses the **creator** pattern.

Record Investment in Custom Asset



The InvestmentController lets the user interact with the system and database so they can change the data in the way they want. Therefore, InvestmentController uses a **controller** pattern. It also creates a form and an asset after the form is filled, so it also uses the **creator** pattern.

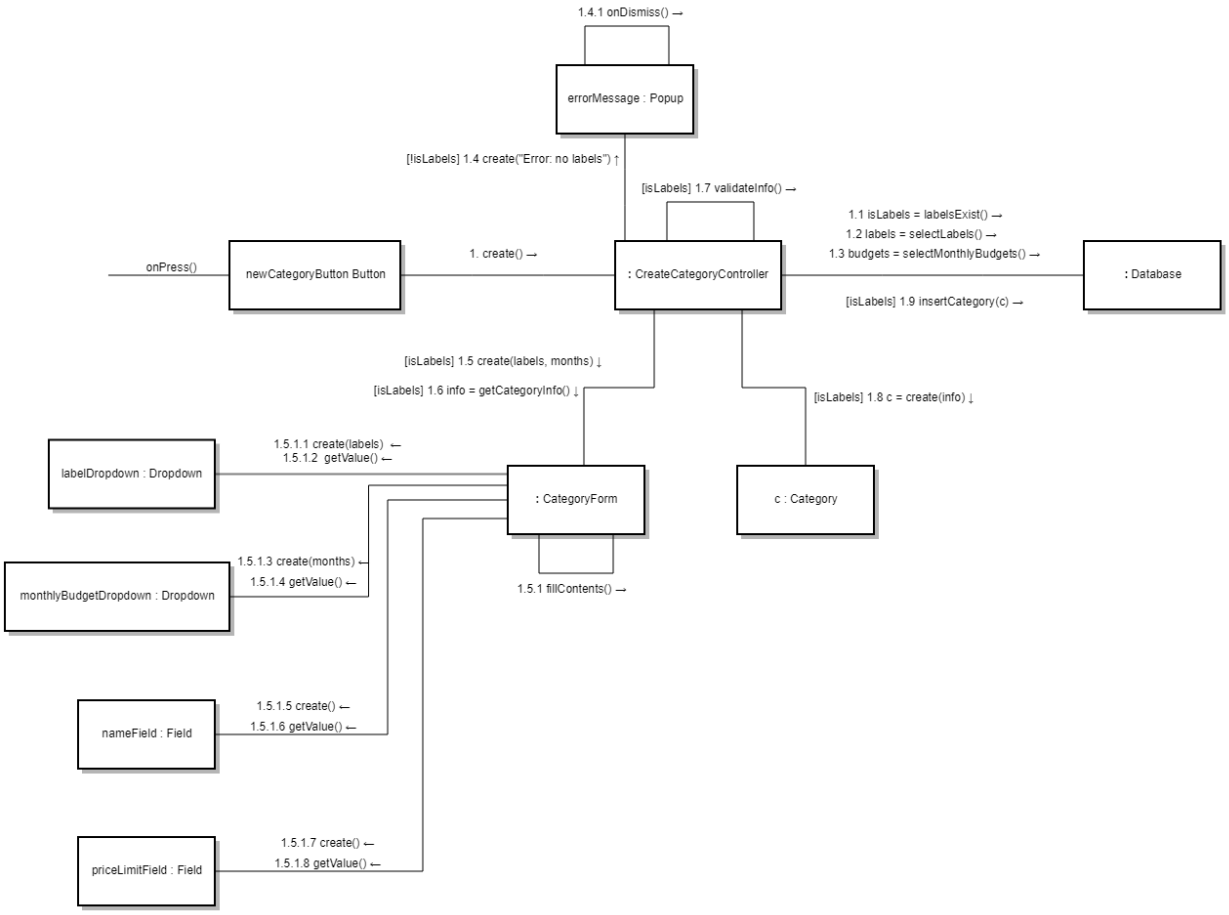
Add Label



This collaboration diagram involves all the steps required to add a label to the system. The instance of the AddLabelController class is assigned the responsibility of the **controller**, coordinating the other objects to complete the use case. To add a level of **indirection** between the user input and the AddLabelController, the LabelForm class has been created. It serves the purpose of handling the user input, and the controller does not require knowledge of how this is accomplished. Similarly, once the user has input the label, the Database class is assigned the responsibility of performing queries on the

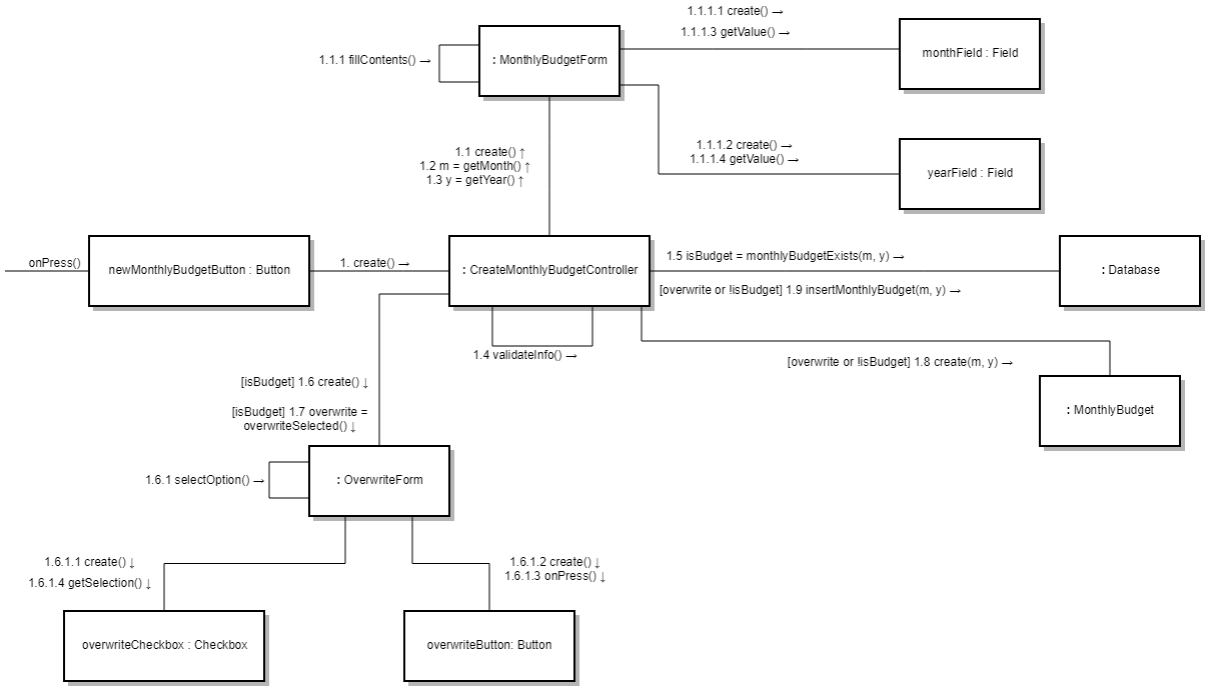
database, since it is the database **expert**. This separation of concerns decreases **coupling** between objects and increases **cohesion** within the objects.

Create Budget Category



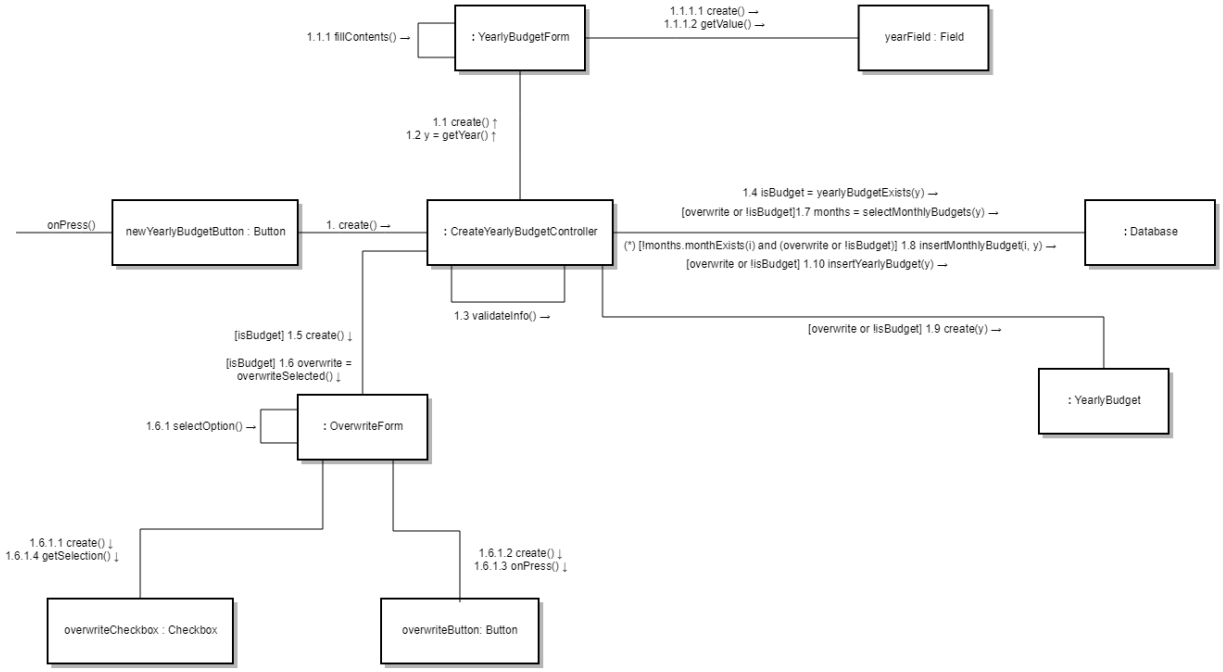
This collaboration diagram involves all the steps required to add a budget category to the system. The CreateCategoryController object is assigned the responsibility of the **controller**, coordinating the boundary objects and the database in order to complete the use case. To add a level of **indirection** between the user input and the CreateCategoryController, the CategoryForm class has been created. The CategoryForm class handles the user input independent of the controller. Similarly, once the user has input the proper data, the Database class is assigned the responsibility of performing the actual queries on the database, since it is the **expert** in regards to database management. This separation of concerns decreases **coupling** between objects and increases **cohesion** within the objects.

Create Monthly Budget



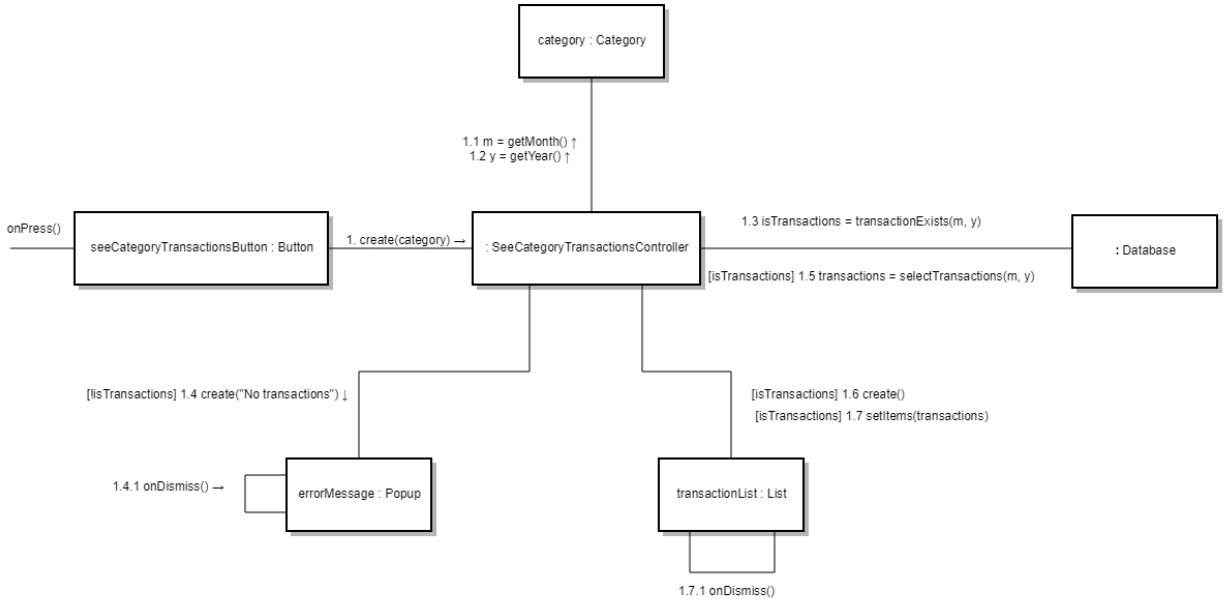
This collaboration diagram involves all the steps required to add a monthly budget to the system. The CreateMonthlyBudgetController object is assigned the responsibility of the **controller**, coordinating the forms and the database in order to complete the use case. To add a level of **indirection** between the user input and the CreateMonthlyBudgetController, the MonthlyBudgetForm class has been created. The MonthlyBudgetForm class handles the user input independent of the controller. Additionally, should the monthly budget be a duplicate, the OverwriteForm allows for an additional level of indirection. Once the user has input the proper data, the Database class is assigned the responsibility of performing the actual queries on the database, since it is the **expert** in regards to database management. This separation of concerns decreases **coupling** between objects and increases **cohesion** within the objects.

Create Yearly Budget



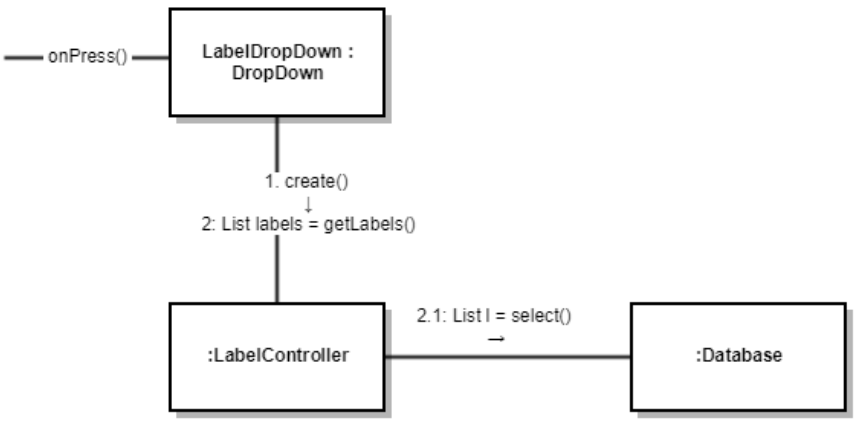
This collaboration diagram involves all the steps required to add a monthly budget to the system. The CreateYearlyBudgetController object is assigned the responsibility of the **controller**, coordinating the forms and the database in order to complete the use case. To add a level of **indirection** between the user input and the CreateYearlyBudgetController , the YearlyBudgetForm class has been created. The YearlyBudgetForm class handles the user input independent of the controller. Additionally, should the monthly budget be a duplicate, the OverwriteForm allows for an additional level of indirection. Once the user has input the proper data, the Database class is assigned the responsibility of performing the actual queries on the database, since it is the **expert** in regards to database management. This separation of concerns decreases **coupling** between objects and increases **cohesion** within the objects.

See Budget Category Transactions



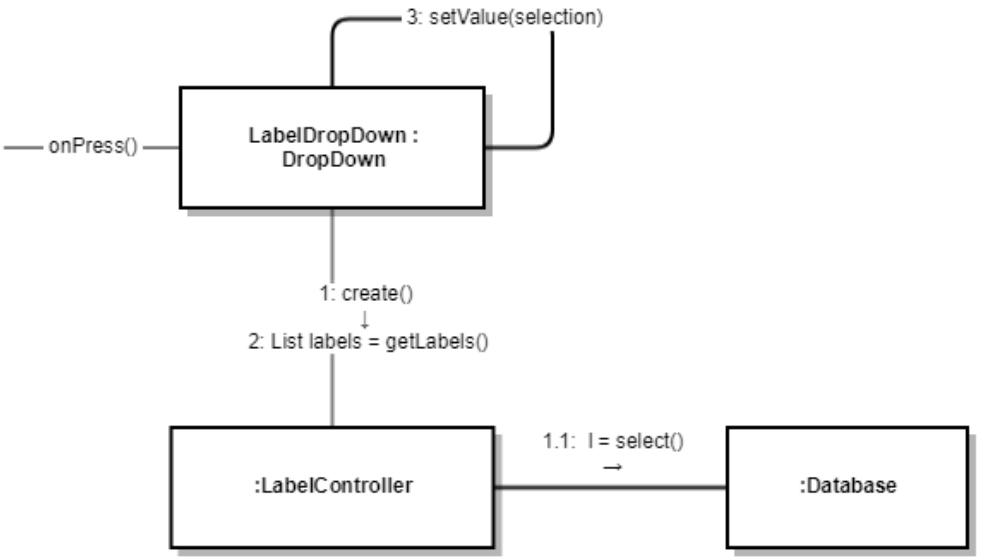
This collaboration diagram involves all the steps required to add a monthly budget to the system. The SeeCategoryTransactionsController object is assigned the responsibility of the **controller**, coordinating the boundary objects and the database in order to complete the use case. The Database class is assigned the responsibility of performing the actual queries on the database, since it is the **expert** in regards to database management. After the transactions are read from the database, the controller sets the transactions to be displayed in a List object. This List object increases the level of **indirection**, since the controller doesn't need to know how the transactions are displayed to the user. This separation of concerns decreases **coupling** between objects and increases **cohesion** within the objects.

View Labels



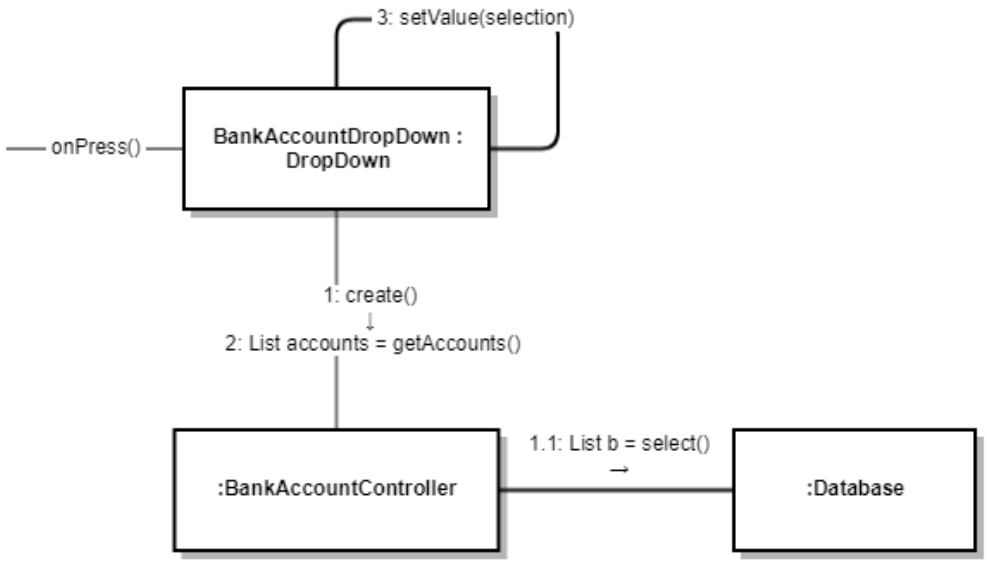
This collaboration diagram for the view labels operation features a **controller** object which handles the operations between the GUI and the database. Since the controller object only interacts with a couple of objects, it has **low coupling**. And since it serves one purpose, it is **highly cohesive**.

Input Transaction Label



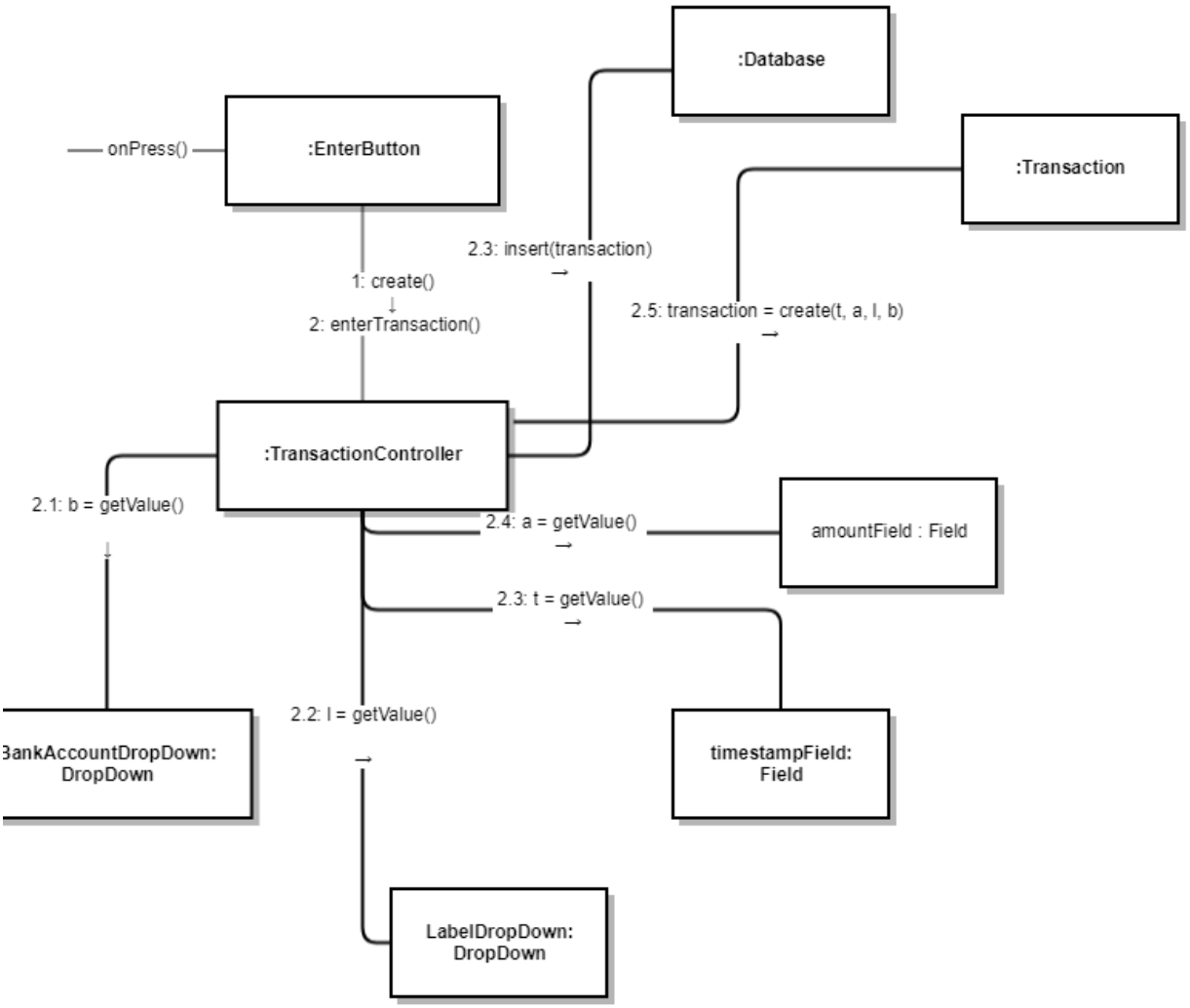
The LabelController object is assigned the **controller** responsibility in order to handle the transfer of data between the GUI and the Database. The object only communicates with two objects, so it is **low in coupling**, and since it only handles that specific passing of a data from the database to the GUI it is **highly cohesive**.

Input Transaction Bank Account



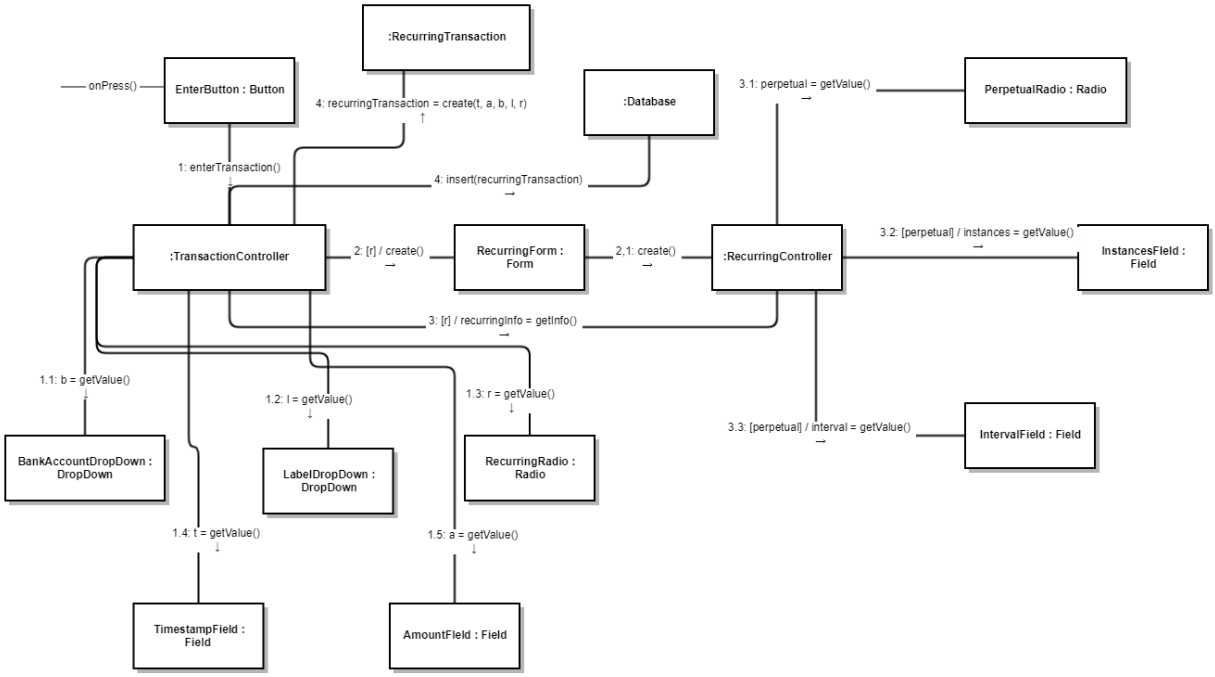
The BankAccountController object is assigned the **controller** responsibility in order to handle the transfer of data between the GUI and the Database. The object only communicates with two objects, so it is **low in coupling**, and since it only handles that specific passing of a data from the database to the GUI it is **highly cohesive**.

Enter Transaction



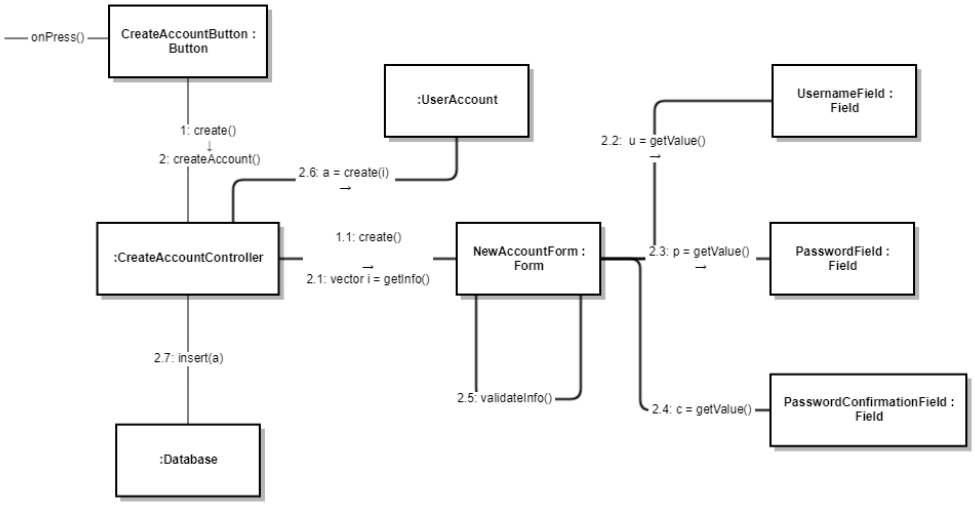
The TransactionController is assigned the **controller** responsibility in order to handle the creation of the transaction and sending it to the database. In addition, since it serves that one purpose, it is **highly cohesive.** The TransactionController also acts as a **creator** since it has the data required to create the transaction object.

Input Recurring Transaction



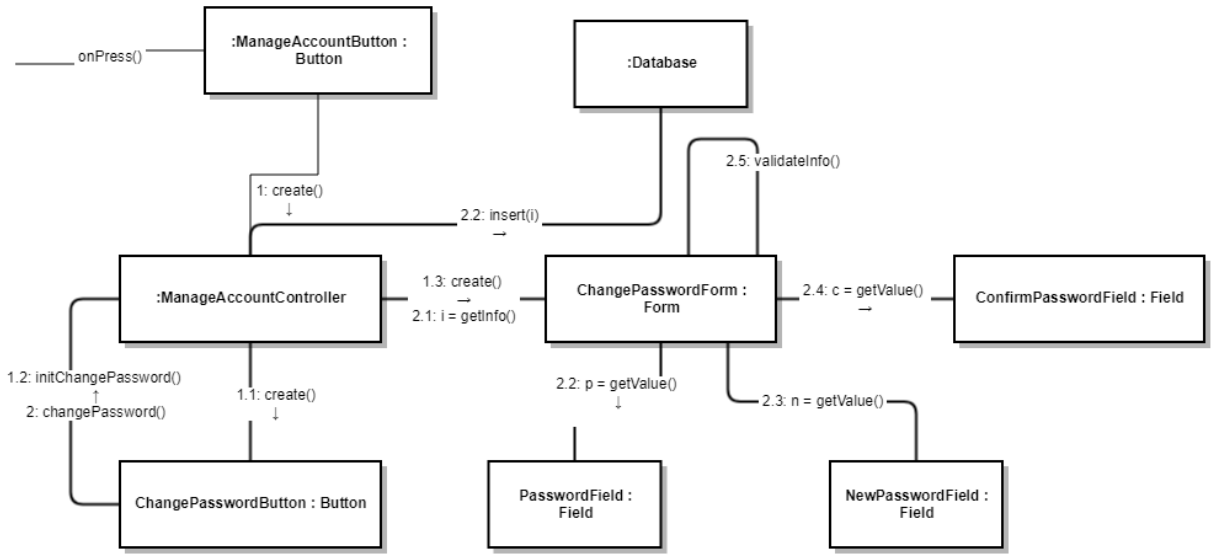
The TransactionController is assigned the **controller** responsibility in order to handle the creation of the transaction object and sending it to the database. **Indirection** is applied through the creation of the RecurringForm and RecurringController to ensure that we maintain **low coupling** in the TransactionController object. Since the TransactionController only exists as a **creator** for Transaction objects, it is **highly cohesive**.

Create Account



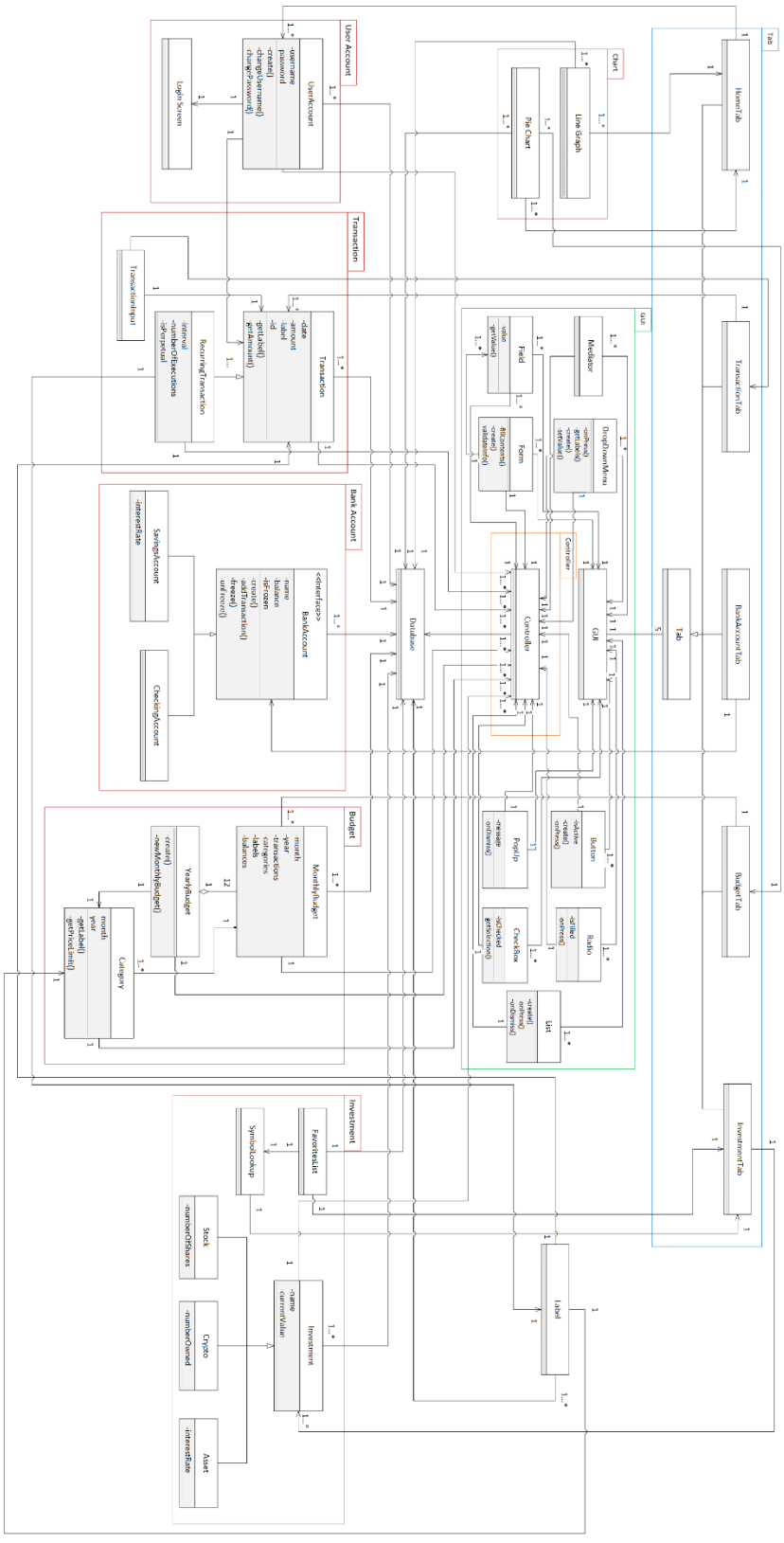
The CreateAccountController object implements the **controller** pattern by handling the operation on behalf on the Button object. Since it retrieves all of the necessary data, it implements the **creator** pattern to manufacture UserAccount objects. It is a **highly cohesive** class because it exists only to create UserAccount objects. In addition, it implements **indirection** by creating a NewAccountForm object to not need to contact every field on its own, making giving it **low coupling**.

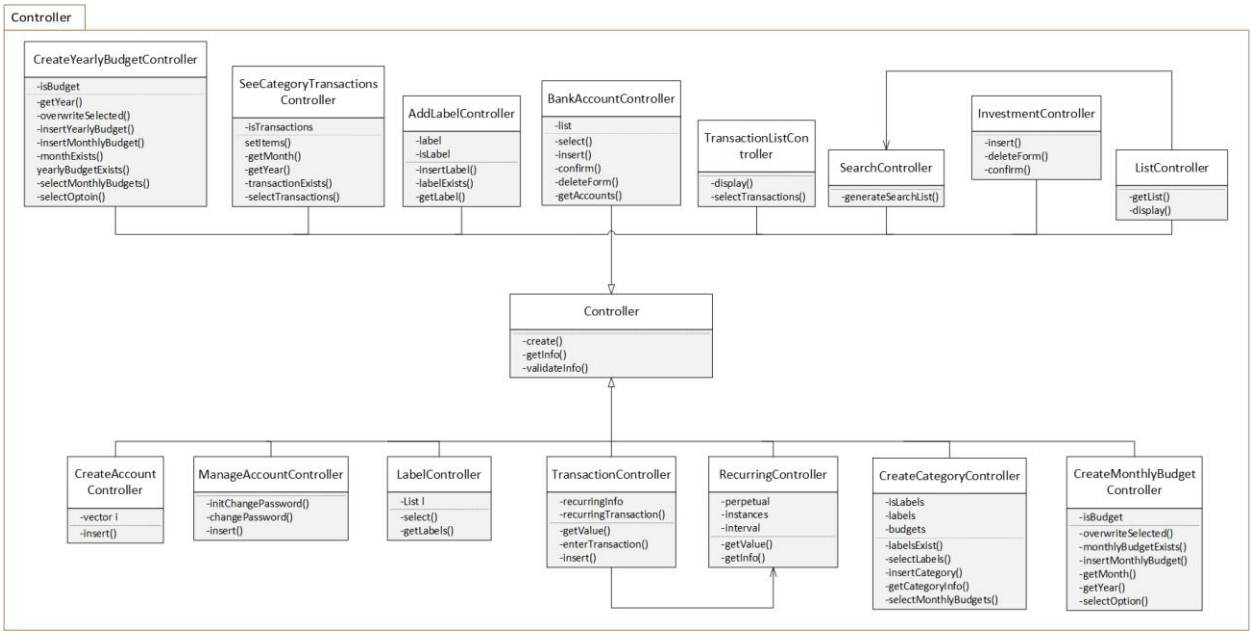
Change Password



The ManageAccountController object implements the **controller** pattern by handling the operation on behalf on the Button object. It is a **highly cohesive** class because it exists only to change the password. In addition, it implements **indirection** by creating a ChangePasswordAccountForm object to not need to contact every field on its own, making giving it **low coupling**.

Design Class Diagram



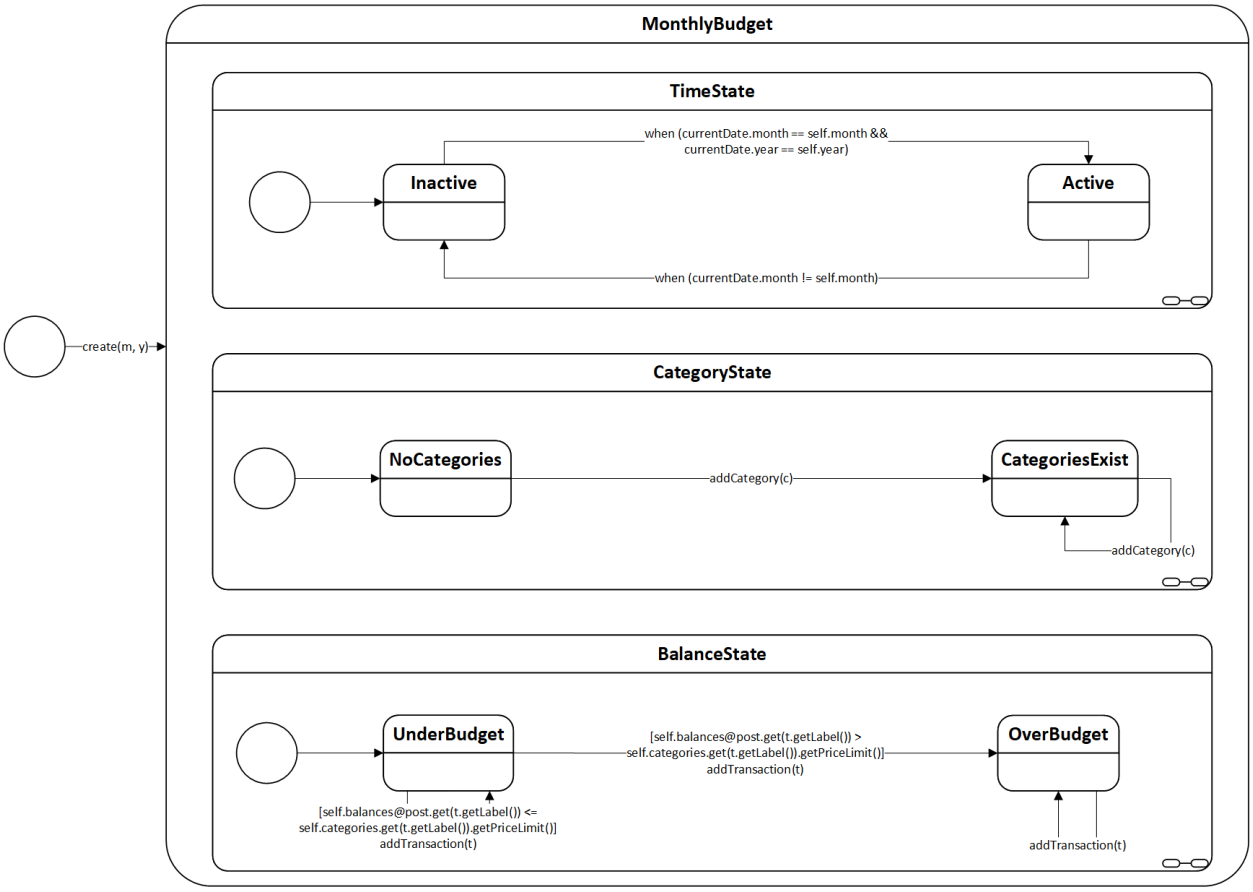


Design Pattern Discussion

We chose to use a mediator in our class design. In our transactions tab, we have a row of inputs that must be filled in before a new transaction can be added. Since all of the fields must be present, we are using a mediator to handle the behavior of the "enter transaction" button. When the fields are not full, the button is disabled. Once all of the information is present, the button becomes clickable. This way, the button doesn't need to communicate with every input.

Class Design

Monthly Budget



Pre/ Post Condition:

create(int m, int y)
Pre: transactions == null && categories == null && labels == null && 0 <= m <= 11 && y > 0
Post: transactions != null && categories != null && labels != null
Pseudocode:

```
month = m
year = y
transactions = new Set<Transaction>
categories = new Map<Category>
labels = new Set<String>
balances = new Map<String, Integer>
```

addTransaction(Transaction t):
Pre: self.transactions->excludes(t)
Post: self.transactions->includes(t) && self.labels.includes(t.getLabel()) && self.balances@post.get(t.getLabel()) == self.balances@pre.get(t.getLabel()) + t.getAmount()
Pseudocode:

```
        if (transactions.contains(t))
                Error "Duplicate transaction"
        else
                transactions.add(t)
                labels.add(t.getLabel())
                temp = balances.get(t.getLabel())
                if (temp != null)
                        balances.put(t.getLabel(), temp + t.getAmount())
```

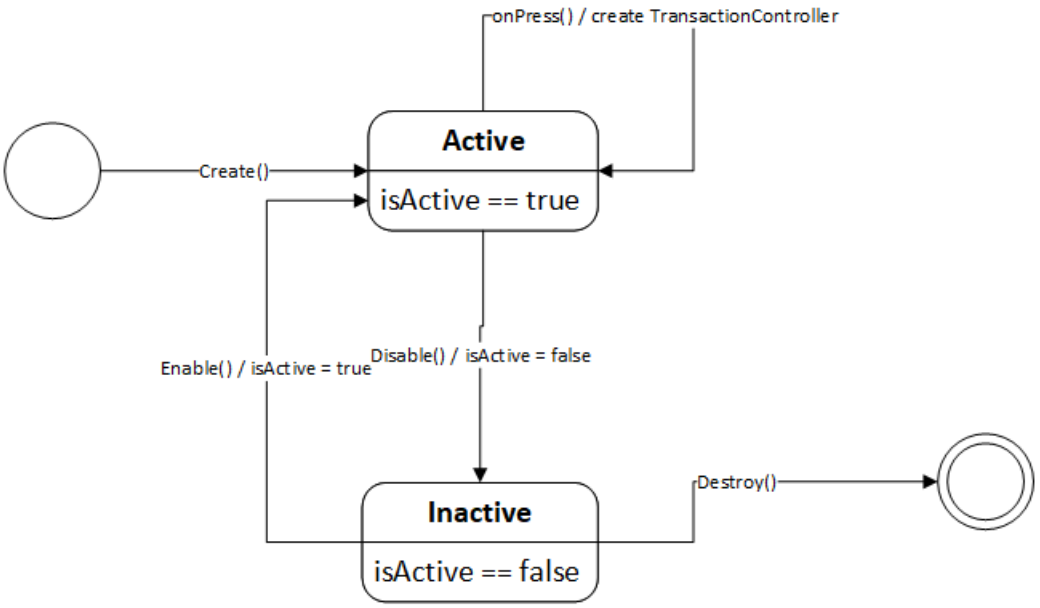
addCategory(Category c)
Pre: self.categories>excludes(c)
Post: self.categories>includes(c) && self.labels.includes(c.getLabel())
Pseudocode:

```
        if (categories.containsValue(c))
                Error "Duplicate category"
        else
                categories.put(c.getLabel(), c)
                labels.add(c.getLabel())
```

Invariants:

```
context : MonthlyBudget
    inv: self.transactions->forAll(t | self.labels->includes(t.getLabel()))
    inv: self.categories>forAll(c| self.labels->includes(c.getLabel()))
    inv: self.balances->forAll(<x, y> | self.transactions->select(t | t.getLabel() == x)->sum() = y)
```

New Transaction Button



Attributes:
bool isActive
TransactionController controller

Methods:
onPress()
enable()
disable()

onPress()
Pre: isActive == true
Post: isActive == true

```
if(isActive){
        controller = new TransactionController
        controller.newTransaction()
}
```

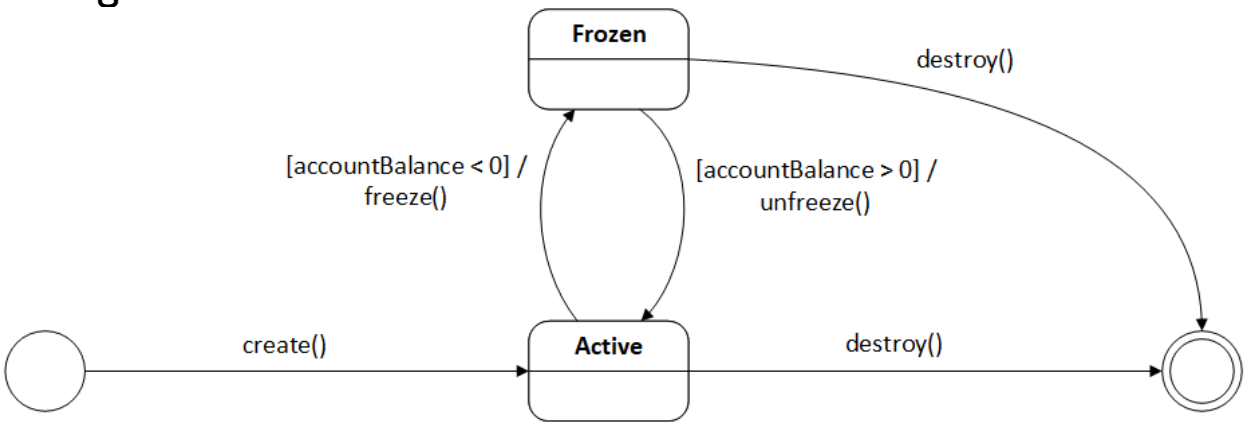
enable()
Pre: isActive == false
Post: isActive == true

```
isActive = true
```

disable()
Pre: isActive == true
Post: isActive == false

```
        isActive = false

inv: self.isActive == true || self.isActive == false
```

Savings Account



Pre/ Post Conditions:

create(String name, Double balance, Double interestRate)
Pre: name == null && balance == null && interestRate == null
Post: name != null && balance <= 0 && interestRate <= 0
Pseudocode:

```
Name = name
Balance = balance
Interest Rate = interestRate
transactions = new Set<Transaction>
```

addTransaction(Transaction t)
Pre: self.transactions->excludes(t)
Post: self.transactions->includes(t) && self.labels.includes(t.getLabel())
Pseudocode:

```
if (transactions.contains(t))
        Error "Duplicate transaction"
else
        transactions.add(t)
```

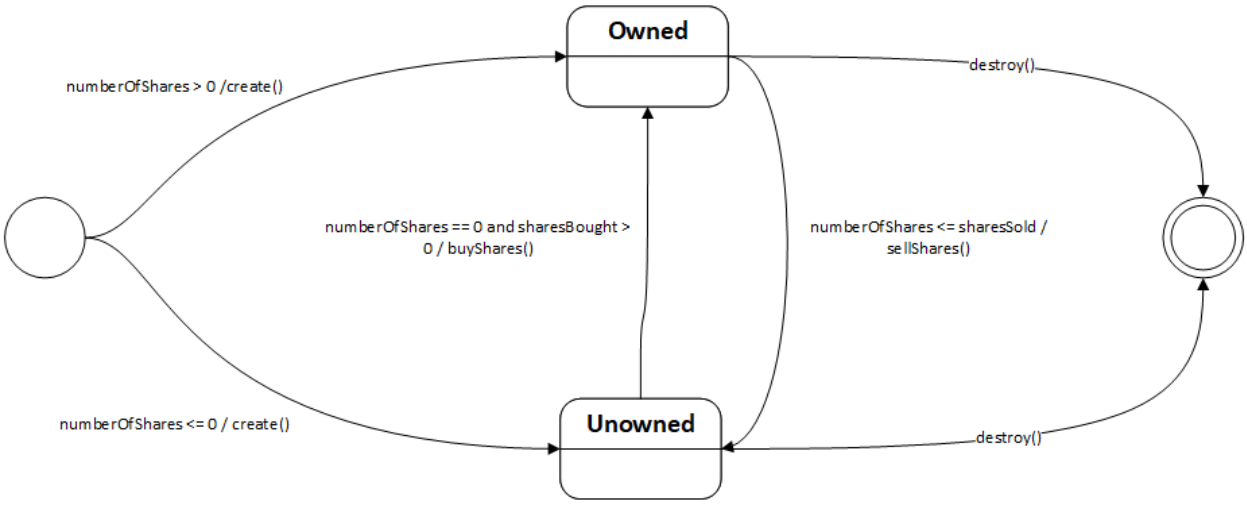
freezeAccount()
Pre: isFrozen == false
Post: isFrozen == true
Pseudocode:

```
isFrozen == true
```

Invariants:

```
context : SavingsAccount
    inv: self.transactions->forAll(t | self.labels->includes(t.getLabel()))
```

Investment



Pre/Post Condition:

create()

Pre: self.name == null && self.currentValue == null && self.numberOfShares == null
Post: self.name != null && self.currentValue != null && self.numberOfShares != null

getNumberOfShares()

Pre: self. numberOfShares != null
Post: self. numberOfShares != null

sellShares()

Pre: self.numberOfShares > 0 && self.sharesSold <= self.numberOfShares
Post: self.numberOfShares >= 0

buyShares()

Pre: self.numberOfShares >= 0 && self.sharesBought > 0
Post: self.numberOfShares == self.numberOfShares + self.sharesBought

Invariant:

context: Investment

inv: self. numberOfShares >= 0

Pseudocode:

create():

```
public create(string nameIn, double valueIn, int numberOfSharesIn)
    1.  name = nameIn
    2.  currentValue = valueIn
    3.  if (numberOfSharesIn <= 0)
    4.      numberOfShares = 0
    5.      isOwned = false
    6.  else
    7.      numberOfShares = numberOfSharesIn
    8.      isOwned = true
```

getNumberOfShares():

```
public int getNumberOfShares()
    1.  return numberOfShares
```

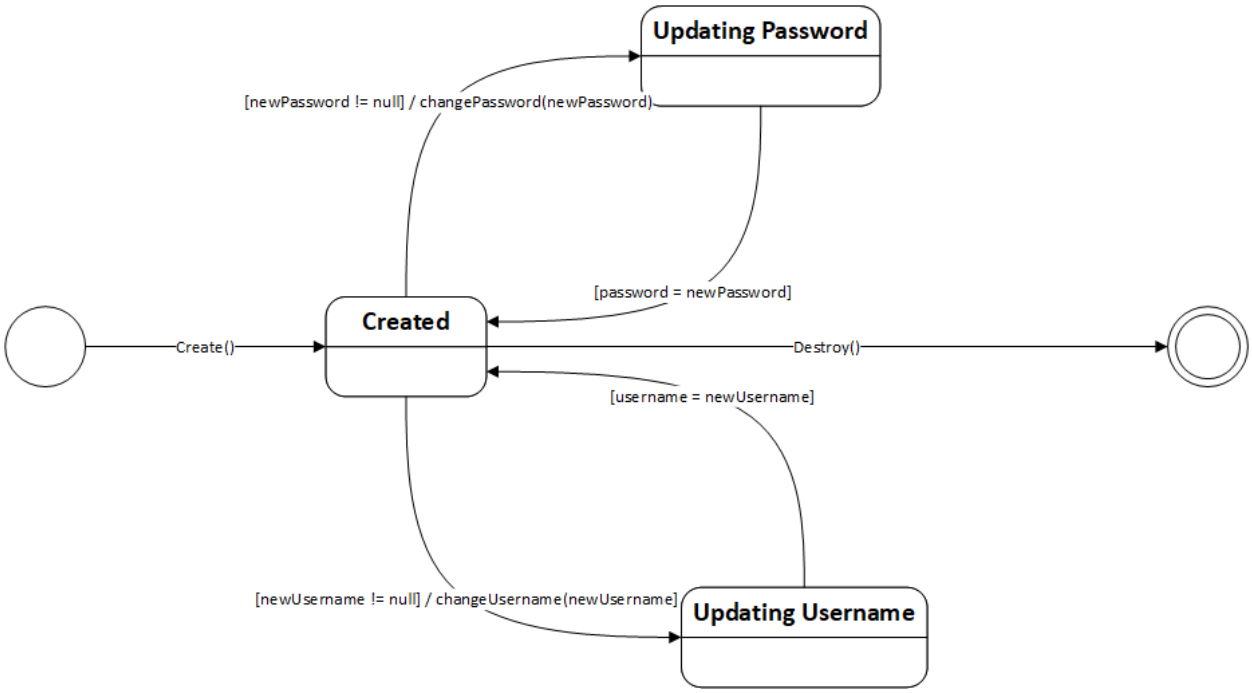
sellShares():

```
public void sellShares(int sharesSold)
    1.  if (shareSold >= numberOfShares)
    2.      numberOfShares = 0
    3.      isOwned = false
    4.  else
    5.      numberOfShares = numberOfShares – shareSold
```

buyShares():

```
public void buyShares(int sharesBought)
    1.  if (numberOfShares <= 0 && sharesBought > 0)
    2.      numberOfShares = numberOfShares + sharesBought
    3.      isOwned = true
    4.  else
    5.      numberOfShares = numberOfShares + sharesBought
```

User Account



Pre/ Post Condition:

create(usernameIn, passwordIn)
Pre: self.username == null && self.password == null
Post: self.username != null && self.password != null

username = usernameIn
password = passwordIn

changeUsername(newUsername)
Pre: self.username != null && newUsername != self.username && newUsername != null
Post: self.username = newUsername

Username = newUsername

changePassword()
Pre: self.password != null && newPassword != self.password && newPassword != null
Post: self.password = newPassword

Self.password = newPassword

inv: self.username != null && self.password != null

Design Quality

For phase II of our project, we made sure to keep all aspects of Phase I in mind. At the center of our entire program are our 15 use cases: Create account, Change Password, Input Transaction, Input Recurring Transaction, Create Monthly Budget, Create Yearly Budget, See Transactions from Budget, Create bank account, See bank account, Look up stock/crypto, Record investment in stock/crypto, Record investment in custom asset, Add transaction label, View Labels, and Create budget category. These use cases cover most of Shark Byte's essential operations.

We began Phase II by deciding which **Architectural Style** we wanted to use to create Shark Byte. After deliberation, we settled on a Repository w/ Model-View-Controller hybrid. We came up with this style because of two main reasons; First, each subsystem will directly access a shared database, making a Repository model necessary. Second, Shark Byte will be heavily dependent on its GUI, making a Model-View-Controller seem very applicable. With this settled, we began crafting our deployment diagram.

Our **UML Deployment Diagram** demonstrates its completeness and consistency through its subsystems. Each use case is allocated to a subsystem in the deployment diagram. The Create Account and Change Password use cases are allocated to the User subsystem; the Input Transaction, Input Recurring Transaction, and See Budget Category Transactions use cases are allocated to the Transaction subsystem; the Create Monthly Budget, Create Yearly Budget, and Create Budget Category use cases are allocated to the Budget subsystem; the Add Bank Account and See Bank Account use cases are allocated to the Bank Account subsystem; the Look Up Stock/Crypto, Record Investment in Stock/Crypto, and Record Investment in Custom Asset use cases are allocated to the Investment subsystem; the Add Label and View Labels use cases are allocated to the Label subsystem. All of the use cases are accounted for in the deployment diagram, so the diagram is consistent and complete with regard to the use cases. Additionally, the deployment diagram is consistent with the architectural model; JDBC will be used to connect to the database and support the repository model. With our Architectural Design completed, we moved on to Detailed Design.

We began the Detailed Design with the **Collaboration Graphs**. These diagrams show the relationships between every object necessary to fulfill each use case. Software objects, methods, data structures, and parameters are all present in the graphs. Each graph was created first by interpreting every message on the detailed system sequence diagrams and translated them in concrete methods with representations of the data passed between them. Since each graph was derived from the sequence diagrams, we know they are consistent. Any differences between the two means that additional objects were inserted to clarify the movement of data. Each collaboration graph accurately described the mechanisms behind each use case or the complex portions that make up a more complicated use case. With the Collaboration diagrams now completed, we moved on to the Design Class Diagram.

Our group spent a lot of time and effort on our **Design Class Diagram**. We used our Consolidated Class Diagram as a starting point and the collaboration diagrams to supplement the diagram. We stepped through each collaboration diagram and added all the attributes, methods and associations mentioned to the classes. Most of this had to do with our controller classes. We spent time thinking through how each class works and what behaviors it might

need. We added methods and attributes as needed. As we went further into Phase II, we added more associations, methods, and attributes to our classes. Our classes will certainly need more methods and attributes, but we believe these will become more apparent as we begin to implement Shark Byte. The Design Class Diagram was really the last thing to be fully completed, since it was always being changed, but we moved on to the Class Design when the bulk of it was completed.

After the overall Class Design was over we began to look back at the overall quality of our design, specifically the **Design Metrics**. Overall, our classes have a good number of methods in them. Since much of our functionality is handled by controller classes, each controller is very cohesive. As a result, each one has only the required methods to handle the operations. In addition, not many member variables are needed in these classes. However, for our objects that represent data, a handful of member variables are needed to contain the required data. The overall application does not use many global variables and methods, keeping true to object oriented design principles. Our design can mostly be broken into a few distinct, fundamental types: GUI elements, controllers, and data types. The GUI elements contain very few methods, since each serves a single purpose of handling a click event or collecting user input. These methods do not need any parameters. The data types we created are a bit more involved. These classes have a lot more member variables. However, these classes only have a few key methods, leaving the complicated operations to be handled by the classes that pass the data around. These classes have a degree of interfacing as their contents can be modified by other classes. The controller classes are the most complex. They don't have many member variables, because most of the application's data is stored within other classes. However, these classes have significantly more methods since they handle most of the application's operations. These methods usually also require several parameters since they handle the movement and manipulation of data throughout the application.

After the Design Metrics we went over the **Size Metrics** of our design. The total number of classes in our design class diagram is 83. The average number of methods per class is 1.57. Because of this, many of our classes will be easy to test. The inheritance depth is very shallow so there is not a lot of method sharing. This also makes it easier to test and understand. Most of the classes in the diagram use another class causing a high degree of coupling especially in the controller classes. This makes the program more complex and error prone, but the nature of our program requires high coupling so it is an unavoidable issue. We think that overall our design quality is very high, and we put a lot of effort into reducing any inconsistency.