Design Document - Iteration 2

Team PA-PI-a

18 March 2018

Table 1: Team

Name	ID Number
Melanie Taing	40009850
Laurie Gagnon	22943433
Wayne Yiel Leung	26586988
Jordan Rutty	27300107
Alice Barkhouse	27486782
Michael Foo	40000225
Pierre-Andre Leger	40004010
Colin Greczkowski	40001600

Contents

List of Figures

1 Introduction

The purpose of this document is to describe and provide details for the design and implementation of the second iteration of the MyMoney application.

The following pages will cover the rational of the architectural design as well as the subsystem interface specifications and details on their implementation. Finally, we will describe three dynamic design scenarios based on use cases specified in the documentation for iteration 1.

2 Architectural Design

The Mymoney application is implemented using a model-view-controller (MVC) architecture. this section will cover the architectural diagram for the MVC as well as the subsystem interface specifications.

2.1 Architectural Diagram

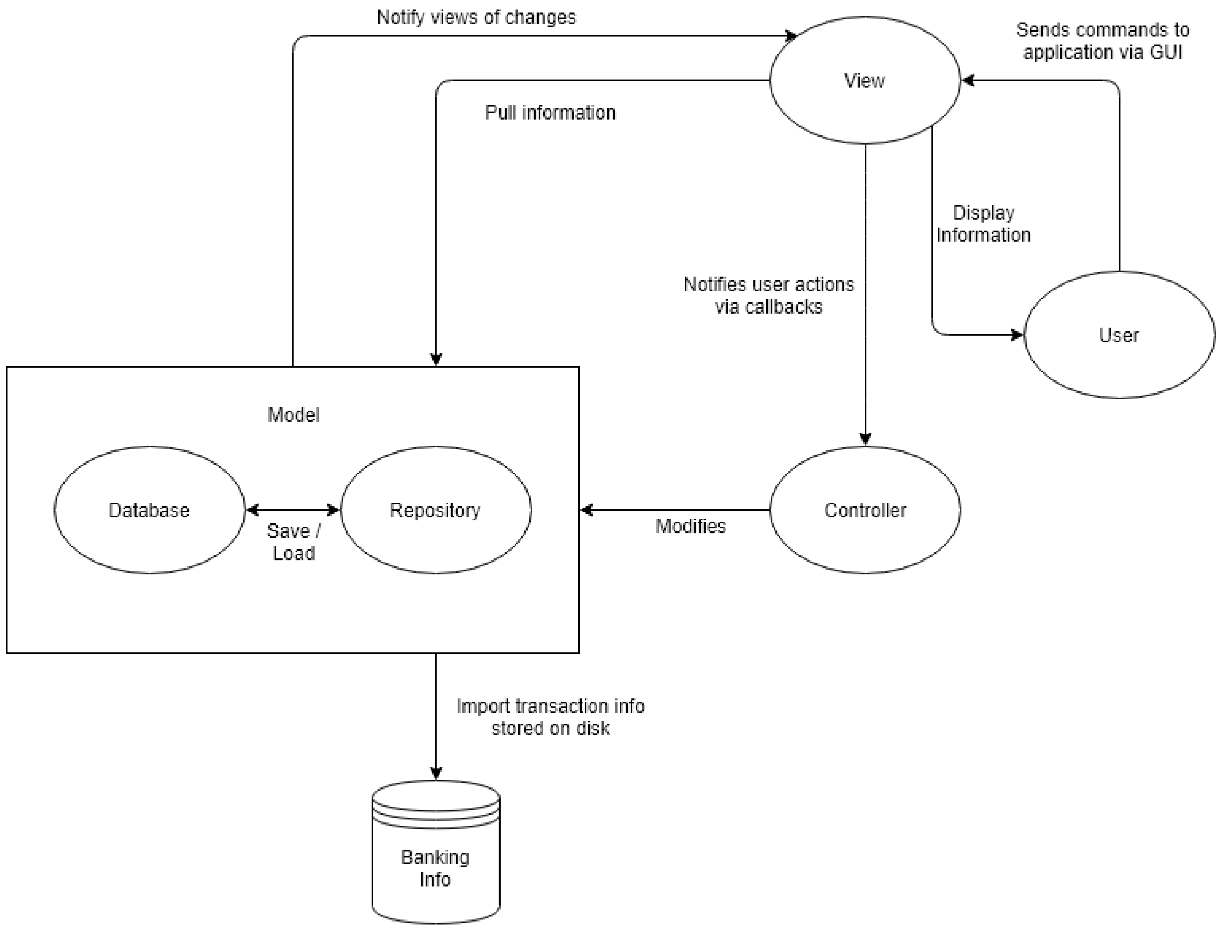


Figure 1: High level structure of MVC architecture

The model contains all the information related to the transactions and the accounts that the user wishes to track. It consists of an SQL database used to serialize and deserialize the information between user sessions and a repository with which the rest of the application interacts. Modifications to the repository are saved on-the-fly to the database while the program is running. The view displays the accounts and transactions loaded into the model (repository) and offers interactive elements that the user can interact with. In essence, it is a GUI. The controller handles user input from the view (GUI) and then acts on the model accordingly by adding, modifying or deleting transactions or accounts.

The main advantage of using an MVC pattern is the separation of concerns. As will be demonstrated in the next section, by enforcing each subsystem to depend strictly on `Interface` types when communicating with each other, we can reduce dependencies and greatly increase modularity.

2.2 Subsystem Interface Specifications

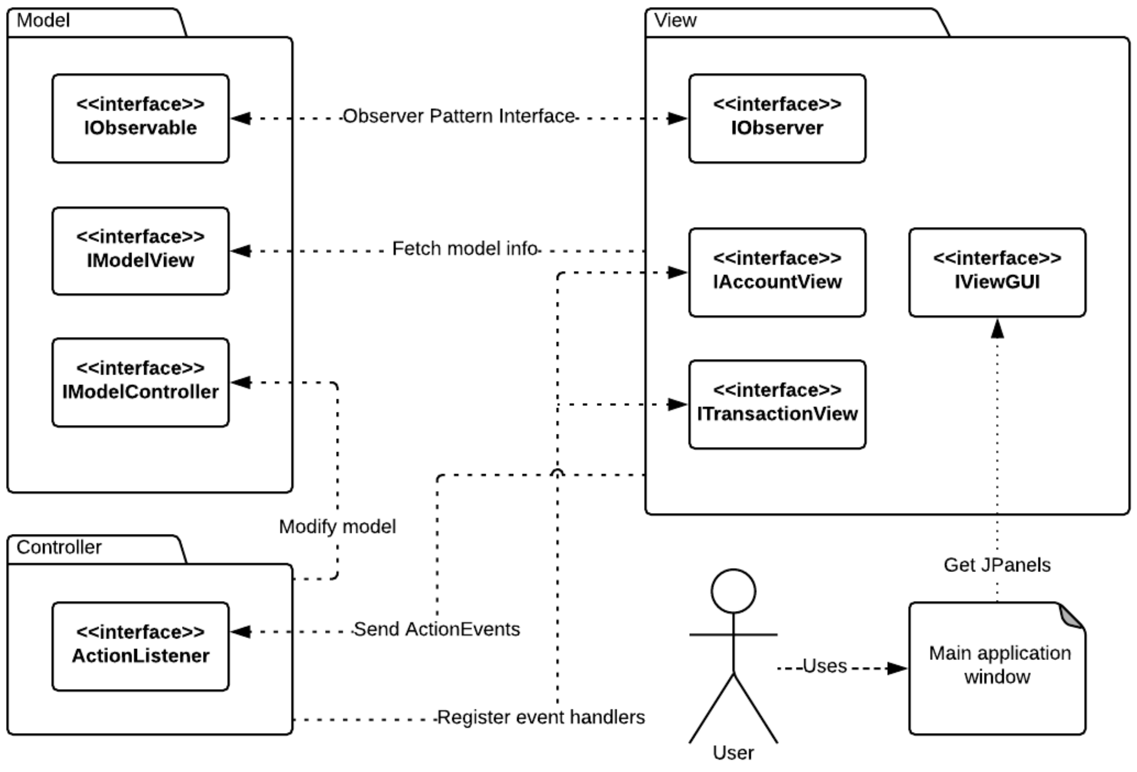


Figure 2: Subsystem specification diagram

2.2.1 Model - View : Observer Pattern

The interfaces `IObservable` and `IObserver` form the observer pattern between the model and the view.

`IObserver`

- `update()` : Called by an `IObservable` object. This should trigger internal logic in the observer to allow it to update its view on the model.

`IObservable`

- `attachObserver(IObserver)` : Attach an observer to this object
- `detachObserver(IObserver)` : Detach an observer from this object
- `notifyObservers()` : Call the `update()` method on all attached observers. Whenever the state of the model changes, it should call this function to allow its attached observers to update their views.

2.2.2 Model : IModelView Interface

The `ImodelView` interface exposes methods to allow the view to fetch information from the the model.

- `getTransactions(Integer accountId)` : Returns a list of `Transaction` objects belonging to the specified accountId. If there are no transactions for the account or the account does not exist, it will return an empty list.
- `getAllAccounts()` : Returns a list of all the `Account` objects for the current user.

2.2.3 Model : IModelController Interface

The `IModelController` interface exposes methods to allow the controller to modify the model.

- `saveTransaction(Transaction)` : Save the given `Transaction` object to the repository and update the SQL database. If the ID of the transaction is 0, create a new entry. Otherwise update the existing one.
- `saveAccount(Account)` : Save the given `Account` object to the repository and update the SQL database. If the ID of the account is 0, create a new entry. Otherwise update the existing one.
- `deleteTransaction(Transaction)` : Delete the specified transaction from both the repository and the SQL database.
- `deleteAccount(Account)` : Delete the specified account from both the repository and the SQL database.
- `importTransactions(String path, Integer accountId)` : Construct and save `Transactions` objects to the repository and SQL databse from a .csv file located at the specified path. The format of the .csv file should be well defined.

2.2.4 View : IAccountView

The `IAccountView` interface exposes methods to allow the controller to register event listeners for user actions (buttons clicks) and have access to the content of the form fields filled by the user.
The callback system uses Java's `ActionEvent` class.

- `registerAddActionCallback(ActionListener, String)` : Attach the specified listener to the GUI element that should trigger the "Add" action and set the event's action command to the specified string (should be "Add").

- `registerUpdateActionCallback(ActionListener, String)` : Attach the specified listener to the GUI element that should trigger the ”Update” action and set the event’s action command to the specified string (should be ”Update”).
- `registerDeleteActionCallback(ActionListener, String)` : Attach the specified listener to the GUI element that should trigger the ”Delete” action and set the event’s action command to the specified string (should be ”Delete”).
- `getBankInput()` : Return a string consisting of the content of the BankName field in the GUI.
- `getNicknameInput()` : Return a string consisting of the content of the Nickname field in the GUI.
- `getBalanceInput()` : Return an Integer consisting of the content of the Balance field in the GUI.
- `getSelectedAccountId()` : Return a Integer consisting of the id of the account currently selected by the user.
- `setSelection(Integer)` : Overrides the user’s current account selection. This method mostly improves user experience (for example, automatically selects a new account when it is created)

2.2.5 View : ITransactionView

The `ITransactionView` interface exposes methods to allow the controller to register event listeners for user actions (buttons clicks) and have access to the content of the form fields filled by the user.
The callback system uses Java’s `ActionEvent` class.

- `registerAddActionCallback(ActionListener, String)` : Attach the specified listener to the GUI element that should trigger the ”Add” action and set the event’s action command to the specified string (should be ”Add”).
- `registerUpdateActionCallback(ActionListener, String)` : Attach the specified listener to the GUI element that should trigger the ”Update” action and set the event’s action command to the specified string (should be ”Update”).
- `registerDeleteActionCallback(ActionListener, String)` : Attach the specified listener to the GUI element that should trigger the ”Delete” action and set the event’s action command to the specified string (should be ”Delete”).
- `registerImportActionCallback(ActionListener, String)` : Attach the specified listener to the GUI element that should trigger the ”Import” action and set the event’s action command to the specified string (should be ”Import”).

- `getTypeInput()` : Return a string consisting of the content of the Type field in the GUI.
- `getDateInput()` : Return a string consisting of the content of the Date field in the GUI.
- `getDescriptionInput()` : Return a string consisting of the content of the Description field in the GUI.
- `getAmountInput()` : Return an Integer consisting of the content of the Amount field in the GUI.
- `getSelectedAccountId()` : Return a Integer consisting of the id of the account currently selected by the user.
- `getSelectedTransactionId()` : Return a Integer consisting of the id of the transaction currently selected by the user.
- `setSelection(Integer)` : Overrides the user's current transaction selection. This method mostly improves user experience (for example, resetting the current selection when a transaction is deleted)

2.2.6 View : IViewGUI

The `IViewGUI` interface exposes a single method that returns a `JPanel` object. It is only used by the main application window.

- `getPanel()` : Returns the topmost parent `JPanel` of this view. It is meant to be used by the main application window to populate its frame.

2.2.7 Controller : ActionListener

The controller only needs to listen to events triggered by the user's input. From the other subsystems' perspective, it implements a single interface with a single method.

- `actionPerformed(ActionEvent)` : Event handler for `ActionEvent` events created by the view. The controller registers its handlers by using the `IAccountView` or `ITransactionView` interfaces provided by the view(s).

3 Detailed Design

This section will cover the implementation details for each subsystem. Before moving on, a quick overview of the basic types used by the system is in order.

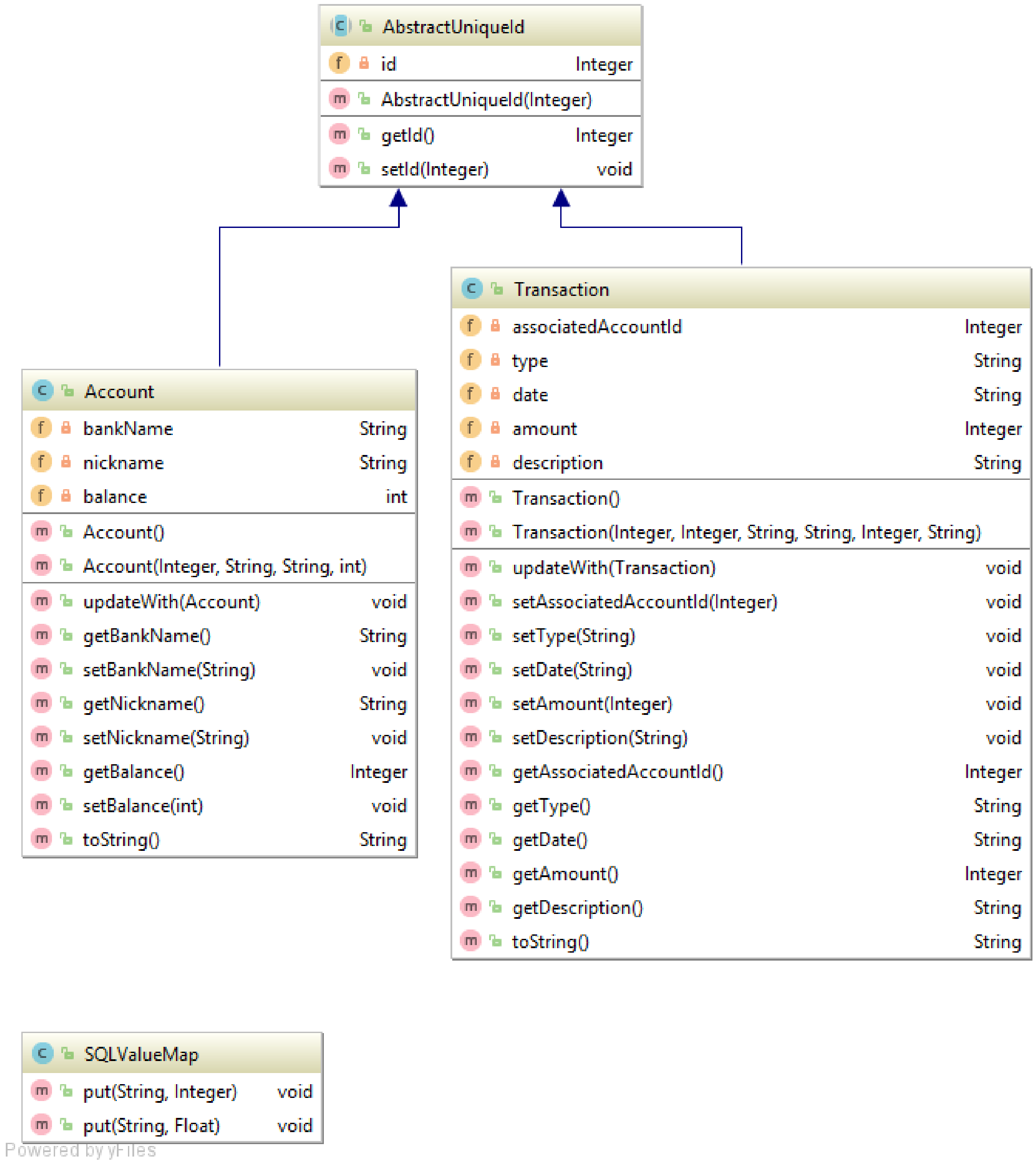


Figure 3: Overview of basic types

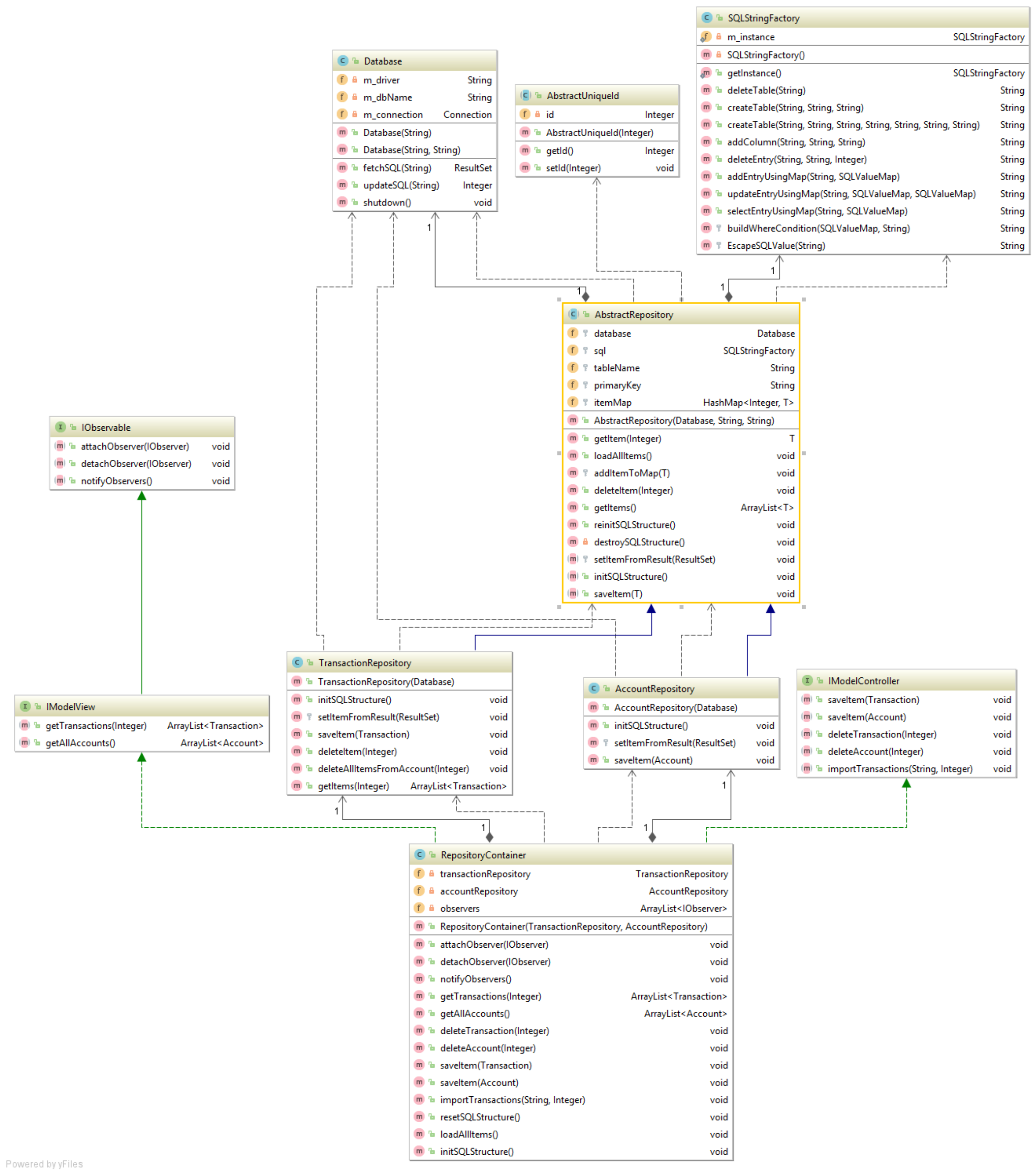
The `Transaction` and `Account` are nothing more than plain old data structures with public read and write access. They both inherit from the `AbstractUniqueId` class which holds the id that will be used as primary key for the database. The `SQLValueMap` is a helper class that derives from Java's `HashMap` and overrides the `put()` method to quickly convert the `value` of the (key,value) pair into a string. `SQLValueMaps` instances are extensively used when building strings that are sent to the SQL database.

In order to simplify the following class diagrams, dependencies on these types are to be assumed and have been omitted.

Links to javadoc generated documentation:
AbstractUniqueId abstract class
Account class
Transaction class
SQLValueMap class

3.1 Model

3.1.1 Design Diagram



SQLStringFactory
m_instance SQLStringFactory
SQLStringFactory()
getInstance() SQLStringFactory
deleteTable(String) String
createTable(String, String, String) String
createTable(String, String, String, String, String, String, String) String
addColumn(String, String, String) String
deleteEntry(String, String, Integer) String
addEntryUsingMap(String, SQLValueMap) String
updateEntryUsingMap(String, SQLValueMap, SQLValueMap) String
selectEntryUsingMap(SQLValueMap, String) String
buildWhereCondition(SQLValueMap, String) String
EscapeSQLValue(String) String
Database
m_driver String
m_dbName String
m_connection Connection
Database(String)
Database(String, String)
fetchSQL(String) ResultSet
updateSQL(String) Integer
shutdown() void
AbstractUniqueId
id Integer
AbstractUniqueId(Integer)
getId() Integer
setId(Integer) void
AbstractRepository
database Database
sql SQLStringFactory
tableName String
primaryKey String
itemMap HashMap<Integer, T>
AbstractRepository(Database, String, String)
getItem(Integer) T
loadAllItems() void
addItemToMap(T) void
deleteItem(Integer) void
getItems() ArrayList<T>
reinitSQLStructure() void
destroySQLStructure() void
setItemFromResult(ResultSet) void
initSQLStructure() void
saveItem(T) void
IObservable
attachObserver(IObserver) void
detachObserver(IObserver) void
notifyObservers() void
TransactionRepository
TransactionRepository(Database)
initSQLStructure() void
setItemFromResult(ResultSet) void
saveItem(Transaction) void
deleteItem(Integer) void
deleteAllItemsFromAccount(Integer) void
getItems(Integer) ArrayList<Transaction>
AccountRepository
AccountRepository(Database)
initSQLStructure() void
setItemFromResult(ResultSet) void
saveItem(Account) void
IModelController
saveItem(Transaction) void
saveItem(Account) void
deleteTransaction(Integer) void
deleteAccount(Integer) void
importTransactions(String, Integer) void
IModelView
getTransactions(Integer) ArrayList<Transaction>
getAllAccounts() ArrayList<Account>
RepositoryContainer
transactionRepository TransactionRepository
accountRepository AccountRepository
observers ArrayList<IObserver>
RepositoryContainer(TransactionRepository, AccountRepository)
attachObserver(IObserver) void
detachObserver(IObserver) void
notifyObservers() void
getTransactions(Integer) ArrayList<Transaction>
getAllAccounts() ArrayList<Account>
deleteTransaction(Integer) void
deleteAccount(Integer) void
saveItem(Transaction) void
saveItem(Account) void
importTransactions(String, Integer) void
resetSQLStructure() void
loadAllItems() void
initSQLStructure() void
1
Powered by yFiles

Figure 4: Model subsystem class diagram

The model subsystem consists of two main parts. The first is the `RepositoryContainer` that interfaces with the rest of the application. It owns the `TransactionRepository` and `AccountRepository` instances and passes read or write commands to each as they are received from the controllers. It also notifies the views of any CRUD operations using the observer pattern.

The second part consists of the SQL database. This is where all information is serialized to allow access between user sessions. It is continuously updated by the repositories as modifications are made. SQL queries to the database can easily be created by using the singleton `SQLStringFactory`.

3.1.2 Units Description

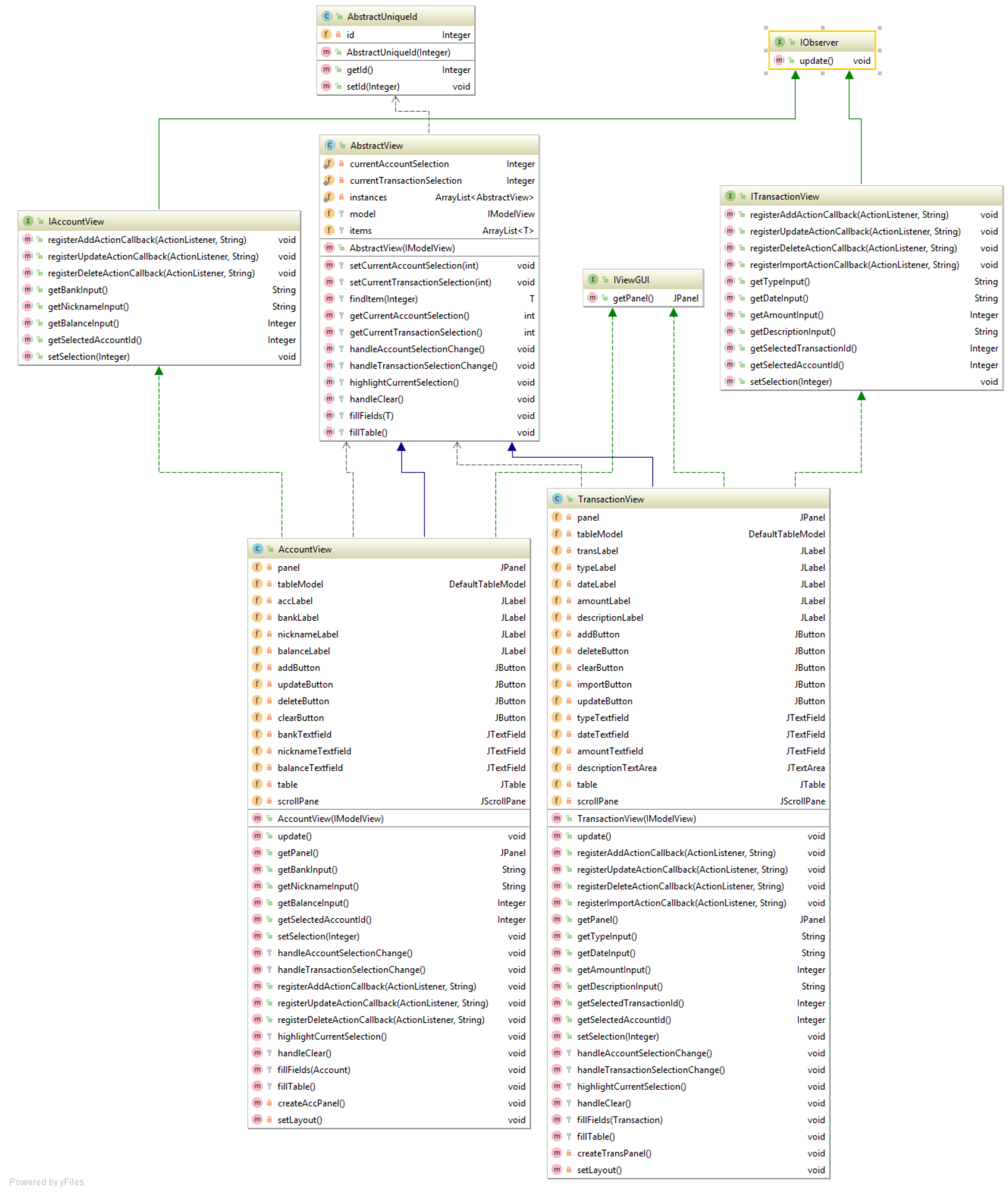
Links to javadoc generated documentation:
IObservable interface
IModelView interface
IModelController interface

AbstractRepository abstract class
AccountRepository class
TransactionRepository class
RepositoryContainer class

Database class
SQLStringFactory class

3.2 View

3.2.1 Design Diagram



AbstractUniqueId
id Integer
AbstractUniqueId(Integer)
getId() Integer
setId(Integer) void
IObserver
update() void
AbstractView
currentAccountSelection Integer
currentTransactionSelection Integer
instances ArrayList<AbstractView>
model IModelView
items ArrayList<T>
AbstractView(IModelView)
setCurrentAccountSelection(int) void
setCurrentTransactionSelection(int) void
findItem(Integer) T
getCurrentAccountSelection() int
getCurrentTransactionSelection() int
handleAccountSelectionChange() void
handleTransactionSelectionChange() void
highlightCurrentSelection() void
handleClear() void
fillFields(T) void
fillTable() void
IAccountView
registerAddActionCallback(ActionListener, String) void
registerUpdateActionCallback(ActionListener, String) void
registerDeleteActionCallback(ActionListener, String) void
getBankInput() String
getNicknameInput() String
getBalanceInput() Integer
getSelectedAccountId() Integer
setSelection(Integer) void
IViewGUI
getPanel() JPanel
ITransactionView
registerAddActionCallback(ActionListener, String) void
registerUpdateActionCallback(ActionListener, String) void
registerDeleteActionCallback(ActionListener, String) void
registerImportActionCallback(ActionListener, String) void
getTypeInput() String
getDateInput() String
getAmountInput() Integer
getDescriptionInput() String
getSelectedTransactionId() Integer
getSelectedAccountId() Integer
setSelection(Integer) void
AccountView
panel JPanel
tableModel DefaultTableModel
accLabel JLabel
bankLabel JLabel
nicknameLabel JLabel
balanceLabel JLabel
addButton JButton
updateButton JButton
deleteButton JButton
clearButton JButton
bankTextfield JTextField
nicknameTextfield JTextField
balanceTextfield JTextField
table JTable
scrollPane JScrollPane
AccountView(IModelView)
update() void
getPanel() JPanel
getBankInput() String
getNicknameInput() String
getBalanceInput() Integer
getSelectedAccountId() Integer
setSelection(Integer) void
handleAccountSelectionChange() void
handleTransactionSelectionChange() void
registerAddActionCallback(ActionListener, String) void
registerUpdateActionCallback(ActionListener, String) void
registerDeleteActionCallback(ActionListener, String) void
highlightCurrentSelection() void
handleClear() void
fillFields(Account) void
fillTable() void
createAccPanel() void
setLayout() void
TransactionView
panel JPanel
tableModel DefaultTableModel
transLabel JLabel
typeLabel JLabel
dateLabel JLabel
amountLabel JLabel
descriptionLabel JLabel
addButton JButton
deleteButton JButton
clearButton JButton
importButton JButton
updateButton JButton
typeTextfield JTextField
dateTextfield JTextField
amountTextfield JTextField
descriptionTextArea JTextArea
table JTable
scrollPane JScrollPane
TransactionView(IModelView)
update() void
registerAddActionCallback(ActionListener, String) void
registerUpdateActionCallback(ActionListener, String) void
registerDeleteActionCallback(ActionListener, String) void
registerImportActionCallback(ActionListener, String) void
getPanel() JPanel
getTypeInput() String
getDateInput() String
getAmountInput() Integer
getDescriptionInput() String
getSelectedTransactionId() Integer
getSelectedAccountId() Integer
setSelection(Integer) void
handleAccountSelectionChange() void
handleTransactionSelectionChange() void
highlightCurrentSelection() void
handleClear() void
fillFields(Transaction) void
fillTable() void
createTransPanel() void
setLayout() void
Powered by yFiles

Figure 5: View subsystem class diagram

The view subsystems consists of two concrete implementations, one for transactions and the other for accounts. They both implement the required interfaces for the other subsystems. Their roles are to update themselves when the model changes via their observer pattern implementation as well as sending events to the controller's registered callbacks methods whenever an action is required from the user input.

3.3 Units Description

Links to javadoc generated documentation:
IObserver interface
IAccountView interface
IViewGUI interface
ITransactionView interface

AbstractView abstract class
AccountView class
TransactionView class

3.4 Controller

3.4.1 Design Diagram

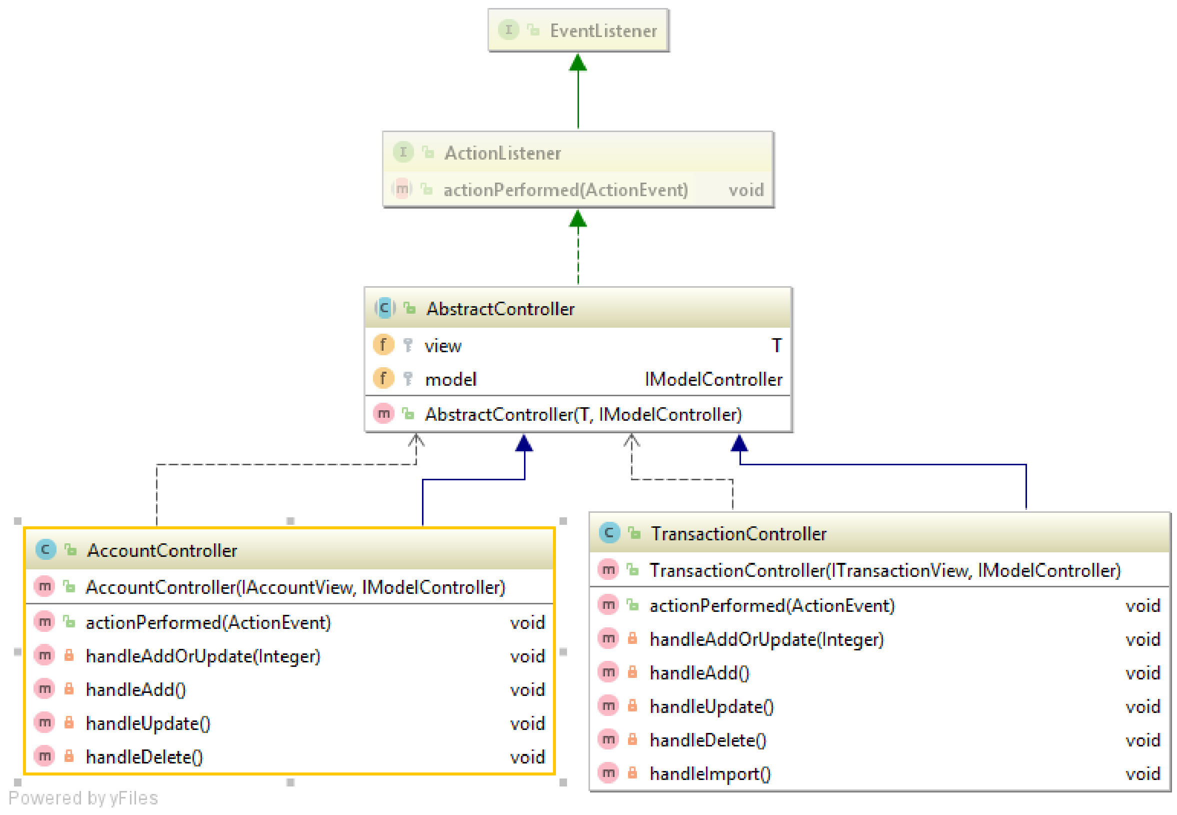

EventListener
ActionListener
actionPerformed(ActionEvent) void
AbstractController
view T
model IModelController
AbstractController(T, IModelController)
AccountController
AccountController(IAccountView, IModelController)
actionPerformed(ActionEvent) void
handleAddOrUpdate(Integer) void
handleAdd() void
handleUpdate() void
handleDelete() void
TransactionController
TransactionController(ITransactionView, IModelController)
actionPerformed(ActionEvent) void
handleAddOrUpdate(Integer) void
handleAdd() void
handleUpdate() void
handleDelete() void
handleImport() void
Powered by yFiles

Figure 6: Controller subsystem class diagram

The controller subsystem is very slim. It consists only of two concrete implementations, one for accounts and the other for transactions. It registers event handlers to the views on creation and calls the appropriate methods on the model through the `IModelController` interface when action is required by the user.

3.4.2 Units Description

AbstractController abstract class
AccountController class
TransactionController class

4 Dynamic Design Scenarios

To illustrate the interactions between the difference classes of our system, we have drawn sequence diagrams for the main features of our program. For simplicity, the model interface is the lowest layer of abstraction in these diagrams. The final section illustrates the internal structure of the model to complement the higher level diagrams.

4.1 Add an account

The first scenario illustrates the addition of an account to the system. The user completes the required text fields and presses the "Add" button, which sends an `ActionEvent` the `AccountController`. The controller then gathers and validates the inputs. If they are valid, the controller constructs an `Account` object and initializes it using the info gathered from the view. Finally, the object is saved by calling the model's `saveAccount()` method. See 4.5.1 for the sequence diagram from the model's perspective once the call to `saveAccount()` is made. The sequence of calls for adding a `Transaction` is fundamentally the same.

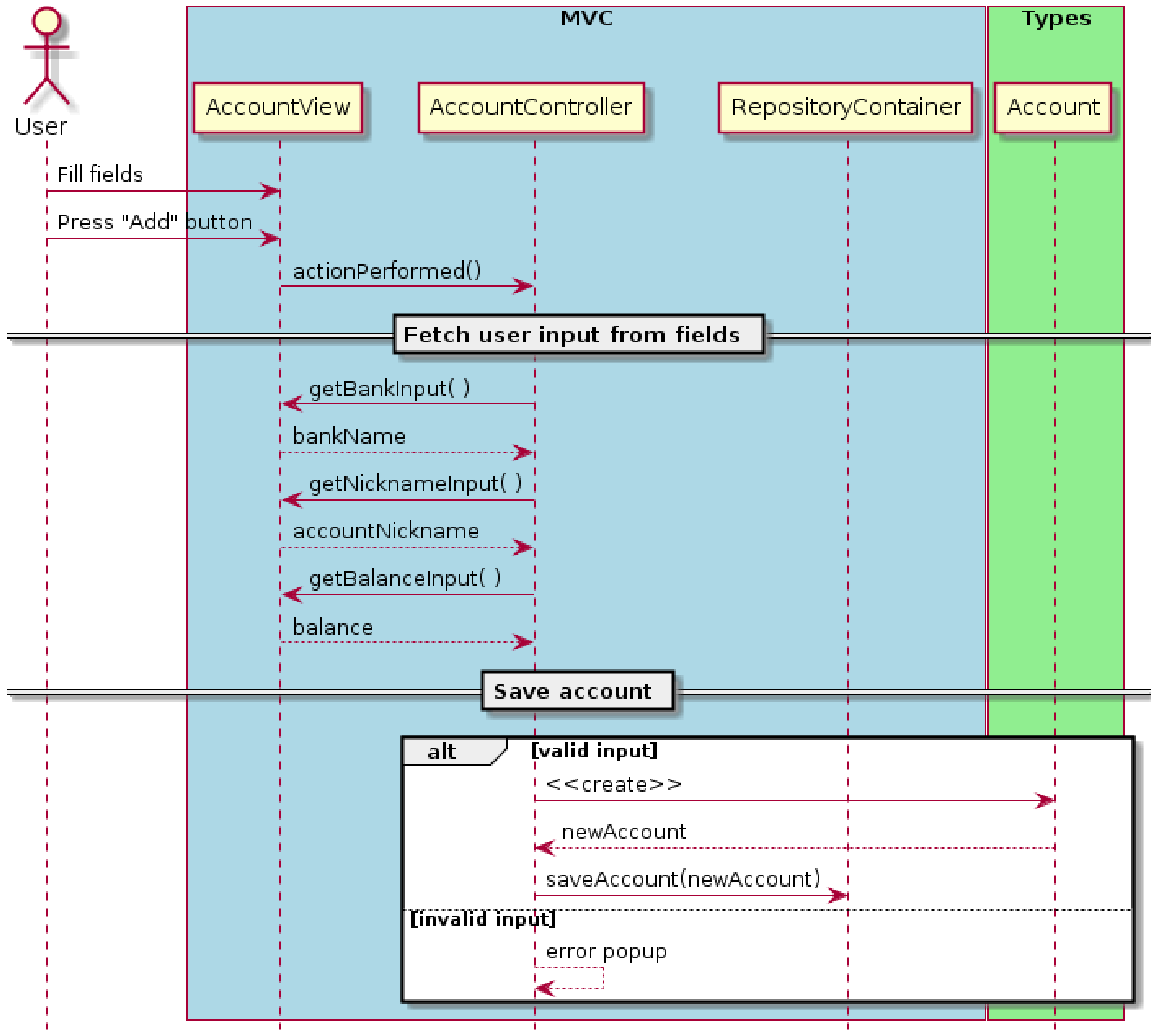


Figure 7: Adding an account

4.2 Update an account

Updating and existing `Account` is very similar to creating a new one. The one difference is that the user must first select an entry from the view. The fields will then update to show the selected `Account`'s information. The user can then modify the fields as desired and press the "Update" button when ready. The flow is then identical to 4.1, with the exception that the `AccountController` will set the created `Account` object's id from the one selected by the user. Again, the sequence of calls for updating a `Transaction` is fundamentally the same.

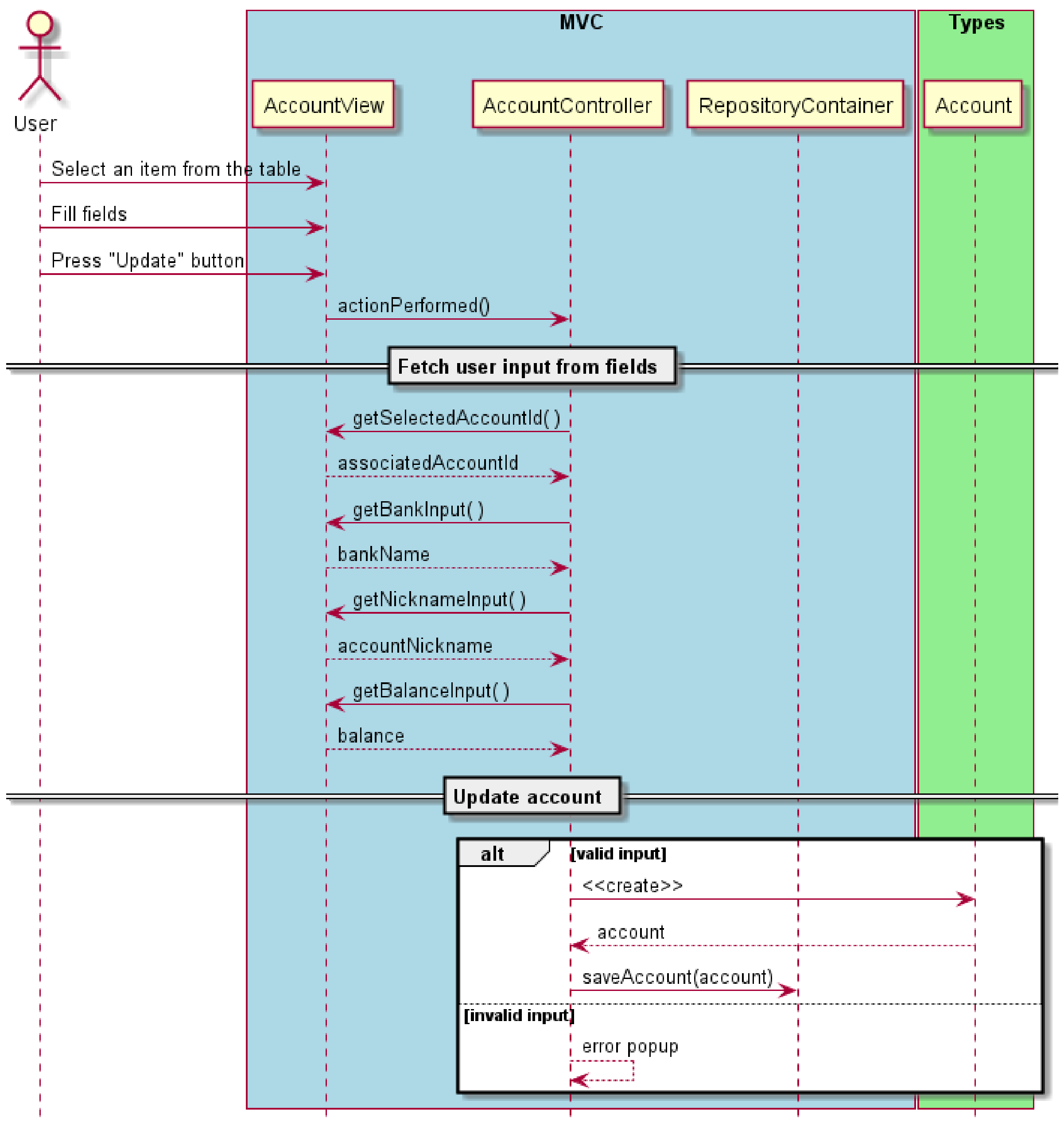


Figure 8: Updating an account

4.3 Delete an account

To delete an account, the user selects an entry in the `AccountView`'s table and clicks the "Delete" button. This sends an `ActionEvent` to the `AccountController`. The registered listener for this event then calls the `deleteAccount()` on the model. `Transaction` deletion is handled similarly.

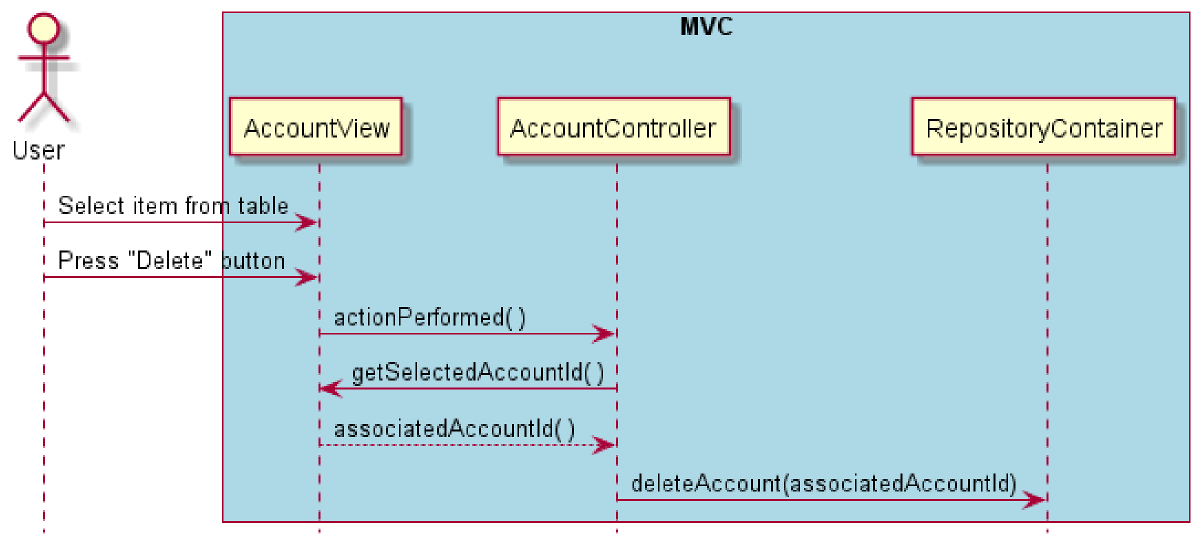


Figure 9: Deleting an account

4.4 Import a transaction list

In this scenario, the users imports a list of transactions from a .csv file and adds them to the model. When the user clicks the "Import" button, the `TransactionController` creates a window with a dialog box. The user then inputs the file path of the transaction csv file. If the file path is valid, both the file path and the currently selected accountId is passed to the model with a call to `importTransactions()`. For details on how the model then handles the import, see 4.5.2.

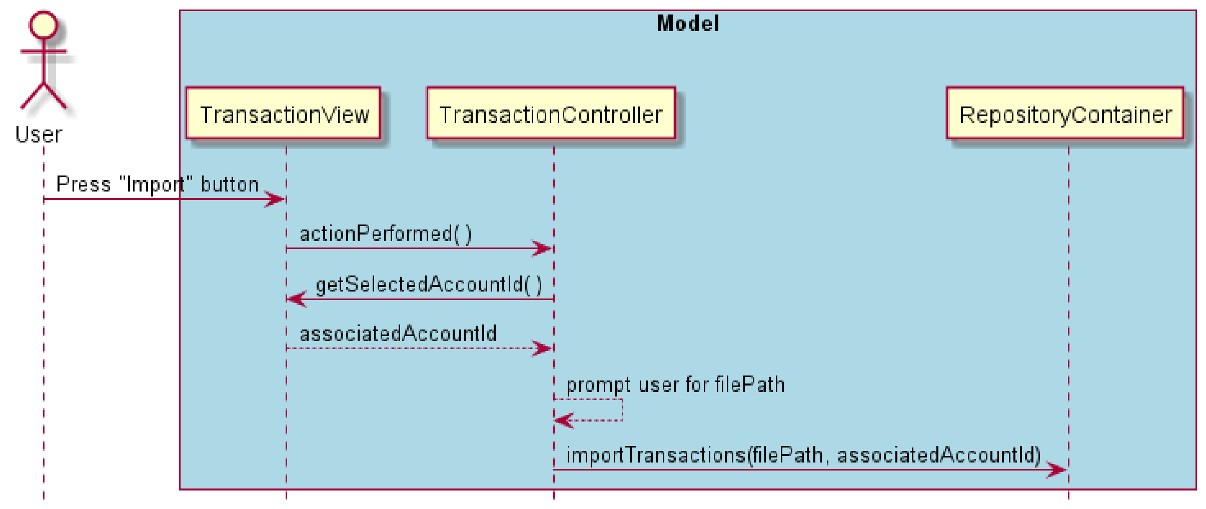


Figure 10: Import a list of transactions from .csv file

4.5 Model implementation details

The next subsections offer a glimpse into the internal logic of the model and how it implements the methods of its interface.

4.5.1 saveAccount()

The `saveAccount(Account)` method provides a simple interface for adding new `Account` objects or updating existing ones. When the specified `Account` object's `accountId` member variable is 0, it is assumed that this is a new entry. Otherwise the call is treated as an update.

After the `saveItem()` method is called, the `AccountRepository` creates an `SQLValueMap` object (linked HashMap with keys and values of type String) to store the column-value mapping.

If the account is new, then the repository uses the `SQLStringFactory` class to build a `String` insert query using the (key,value) pairs in the `SQLValueMap`. It executes the query by calling `updateSQL` from the Database class. The repository then updates the account's ID using the accountId value returned from updateSQL and proceeds to add the new account to the repository's `AccountMap`.

If the account already exists, then the repository will construct a `SQLValueMap` for the where clause of the query using the account's ID. `SQLStringFactory` is then used to generate an update query, which will be executed by calling the `Database` class's `updateSQL` method. The returned accountId is unused.

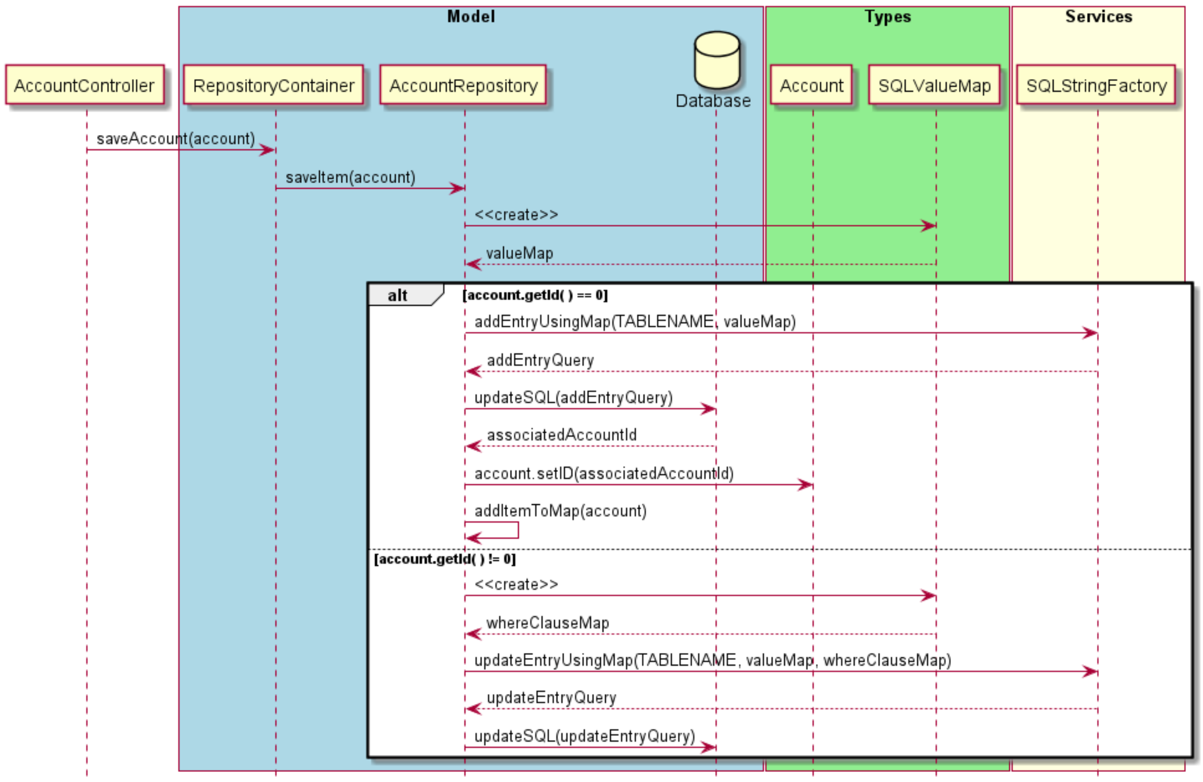
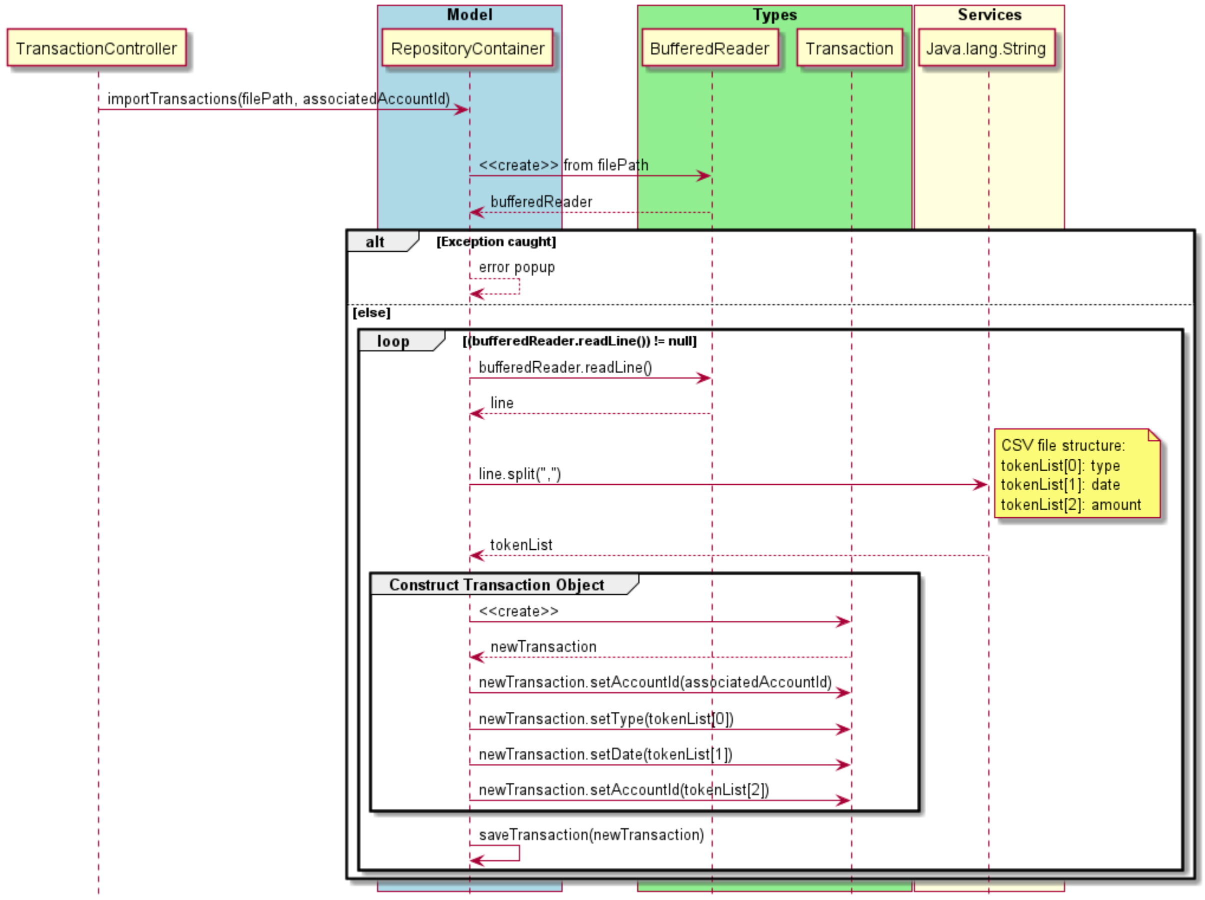


Figure 11: Model - Saving an account

4.5.2 importTransactions()

The `importTransaction()` method first creates a `BufferedReader` using the file path. Then, it iterates over the file, line by line, using the `BufferedReader`'s `readLine()` method.

The line is split into tokens using the `split()` method. Using the returned array of tokens, a `Transaction` object is constructed. The model then saves the new `Transaction` object by calling its own `saveTransaction()` method.



Model
Types
Services
TransactionController
RepositoryContainer
BufferedReader
Transaction
Java.lang.String
importTransactions(filePath, associatedAccountId)
<<create>> from filePath
bufferedReader
alt
[Exception caught]
error popup
[else]
loop
[(bufferedReader.readLine()) != null]
bufferedReader.readLine()
line
line.split(",")
CSV file structure:
tokenList[0]: type
tokenList[1]: date
tokenList[2]: amount
tokenList
Construct Transaction Object
<<create>>
newTransaction
newTransaction.setAccountId(associatedAccountId)
newTransaction.setType(tokenList[0])
newTransaction.setDate(tokenList[1])
newTransaction.setAccountId(tokenList[2])
saveTransaction(newTransaction)

Figure 12: Model - Import list of transactions from .csv file