

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 Ruttly	27300107
Alice Barkhouse	27486782
Michael Foo	40000225
Pierre-Andre Leger	40004010
Colin Greczkowski	40001600

Contents

1	Introduction	5
2	Architectural Design	5
2.1	Architectural Diagram	6
2.2	Subsystem Interface Specifications	7
2.2.1	Model - View : Observer Pattern	7
2.2.2	Model : IModelView Interface	8
2.2.3	Model : IModelController Interface	8
2.2.4	View : IAccountView	8
2.2.5	View : ITransactionView	9
2.2.6	View : IViewGUI	10
2.2.7	Controller : ActionListener	10
3	Detailed Design	11
3.1	Model	13
3.1.1	Design Diagram	13
3.1.2	Units Description	14
3.2	View	15
3.2.1	Design Diagram	15
3.3	Units Description	16
3.4	Controller	16
3.4.1	Design Diagram	16
3.4.2	Units Description	16
4	Dynamic Design Scenarios	16
4.1	Add an account	17
4.2	Update an account	18
4.3	Delete an account	19
4.4	Import a transaction list	20
4.5	Model implementation details	20
4.5.1	saveAccount()	20
4.5.2	importTransactions()	22

List of Figures

1	High level structure of MVC architecture	6
2	Subsystem specification diagram	7
3	Overview of basic types	11
4	Model subsystem class diagram	13
5	View subsystem class diagram	15
6	Controller subsystem class diagram	16
7	Adding an account	17
8	Updating an account	18
9	Deleting an account	19
10	Import a list of transactions from .csv file	20
11	Model - Saving an account	21
12	Model - Import list of transactions from .csv file	22

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

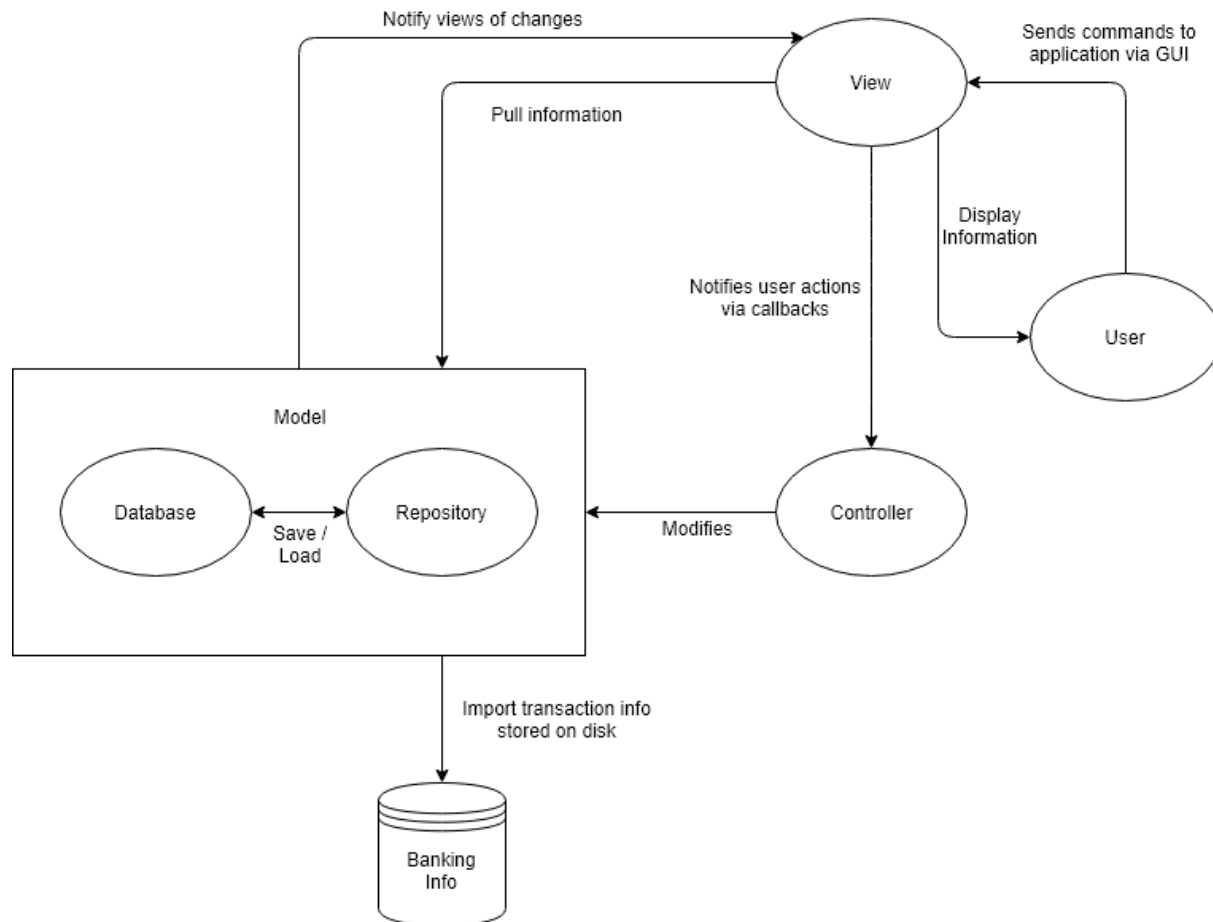


Figure 1: High level structure of MVC architecture

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

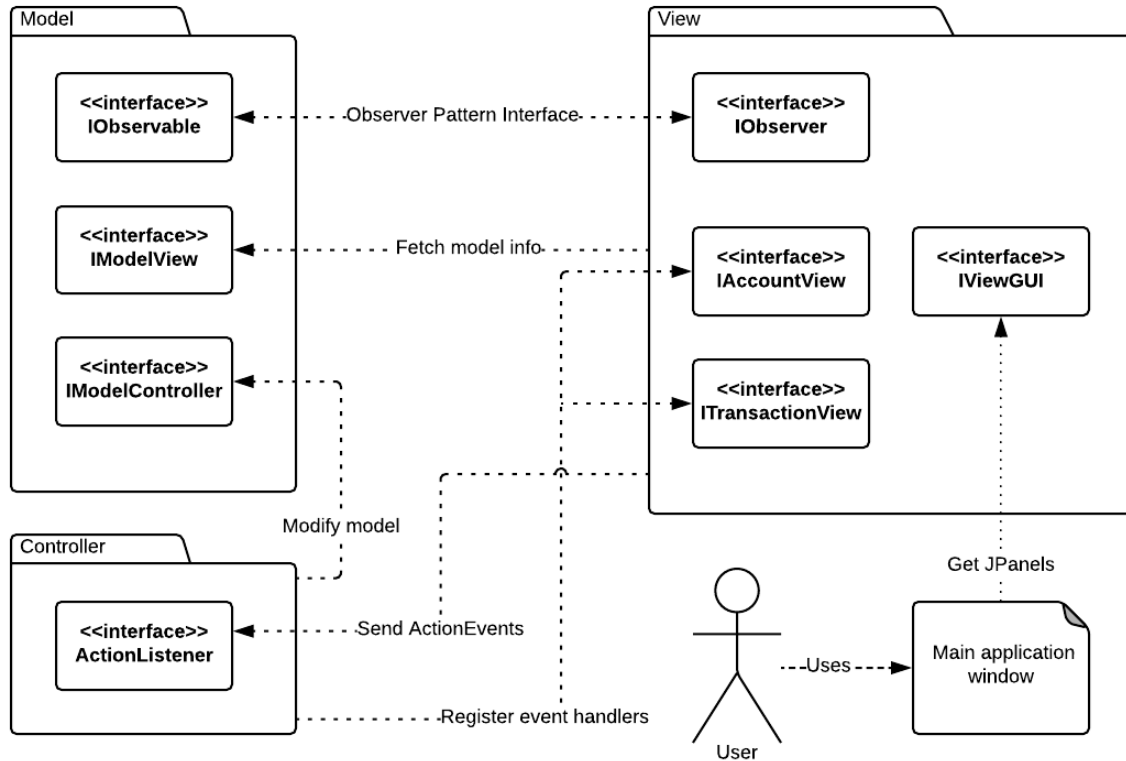


Figure 2: Subsystem specification diagram

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 database 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.

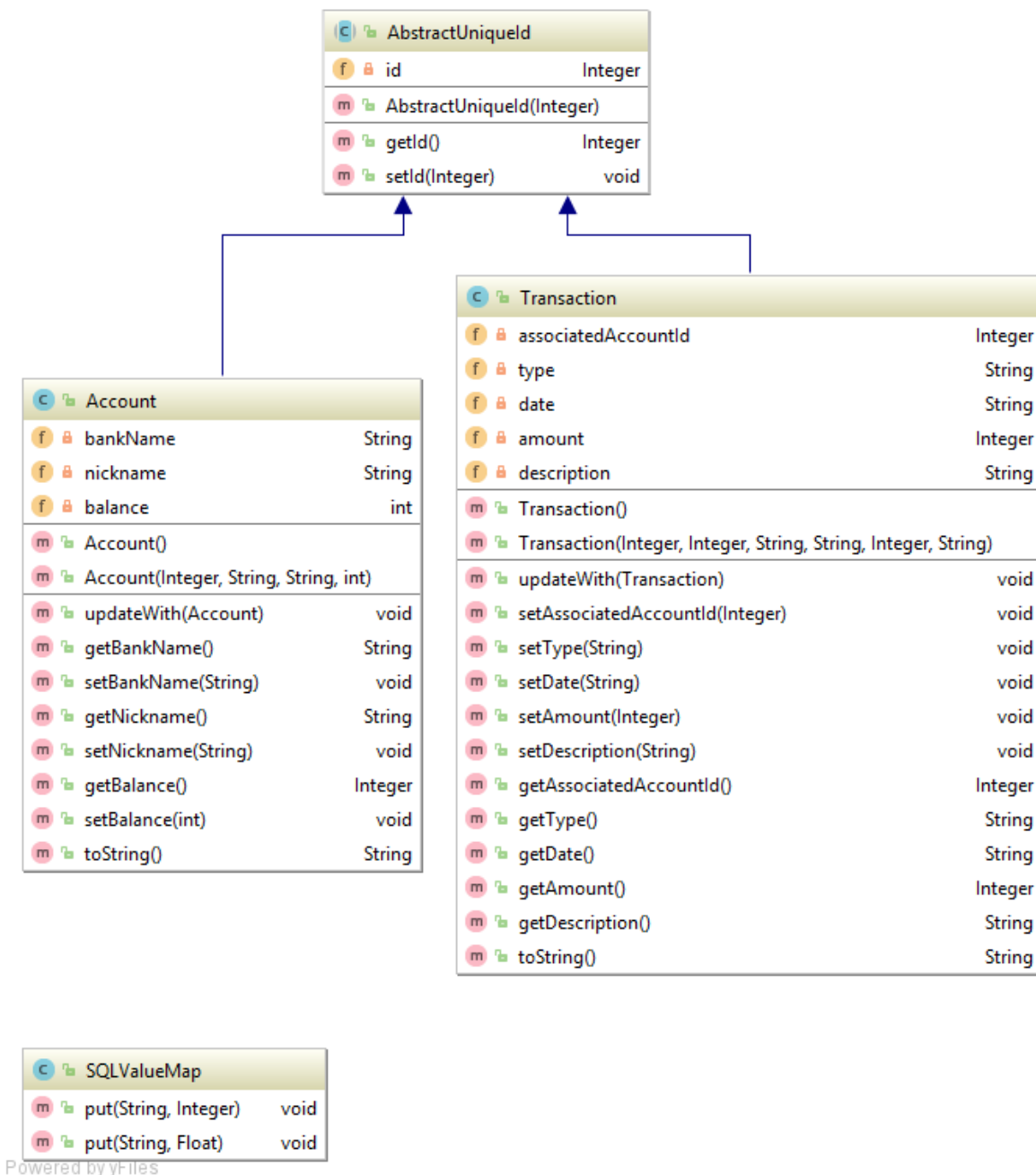


Figure 3: Overview of basic types

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.

3.1 Model

3.1.1 Design Diagram

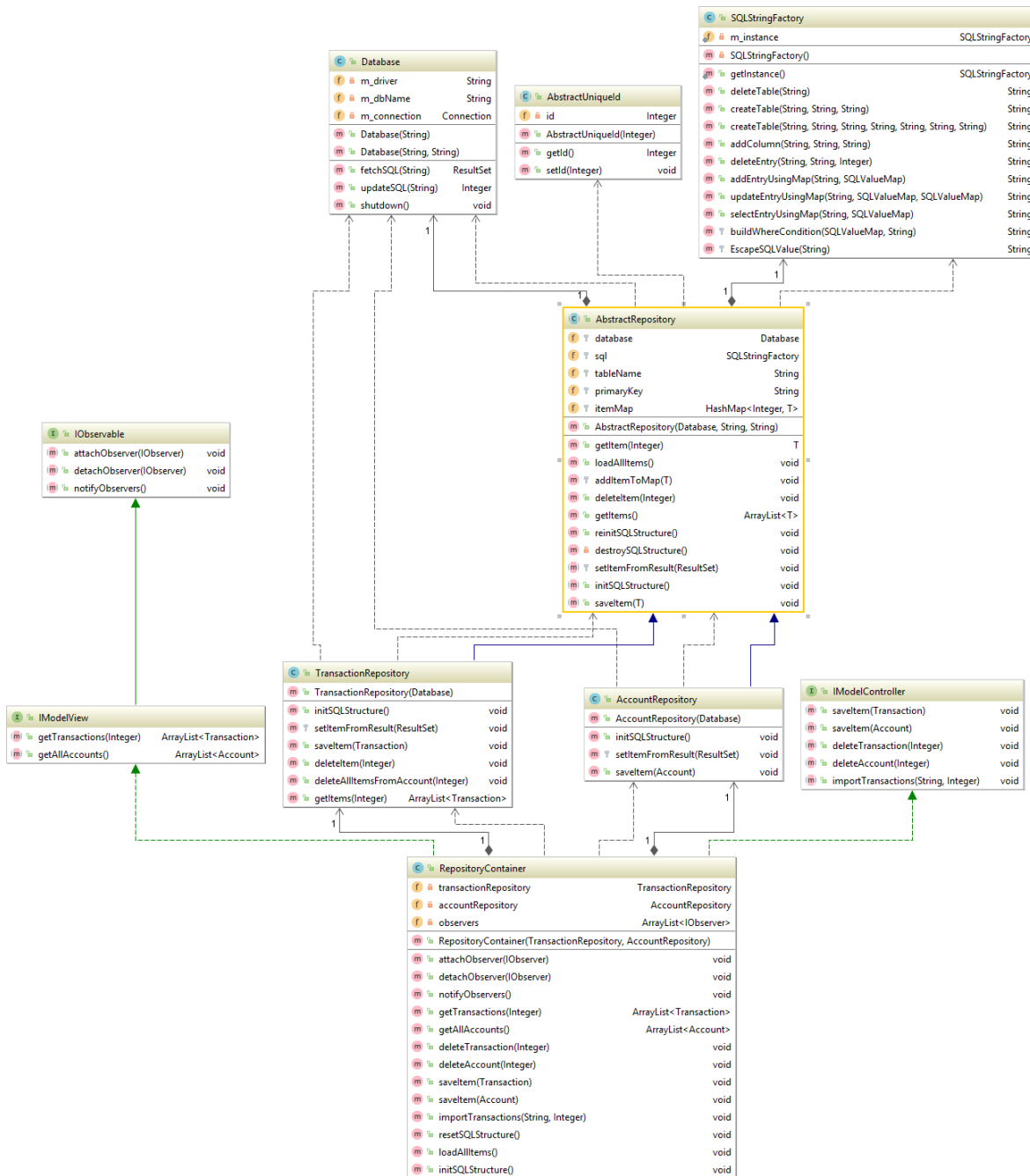


Figure 4: Model subsystem class diagram

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

TODO – Generate from JavaDoc for each class in the subsystem

3.2 View

3.2.1 Design Diagram

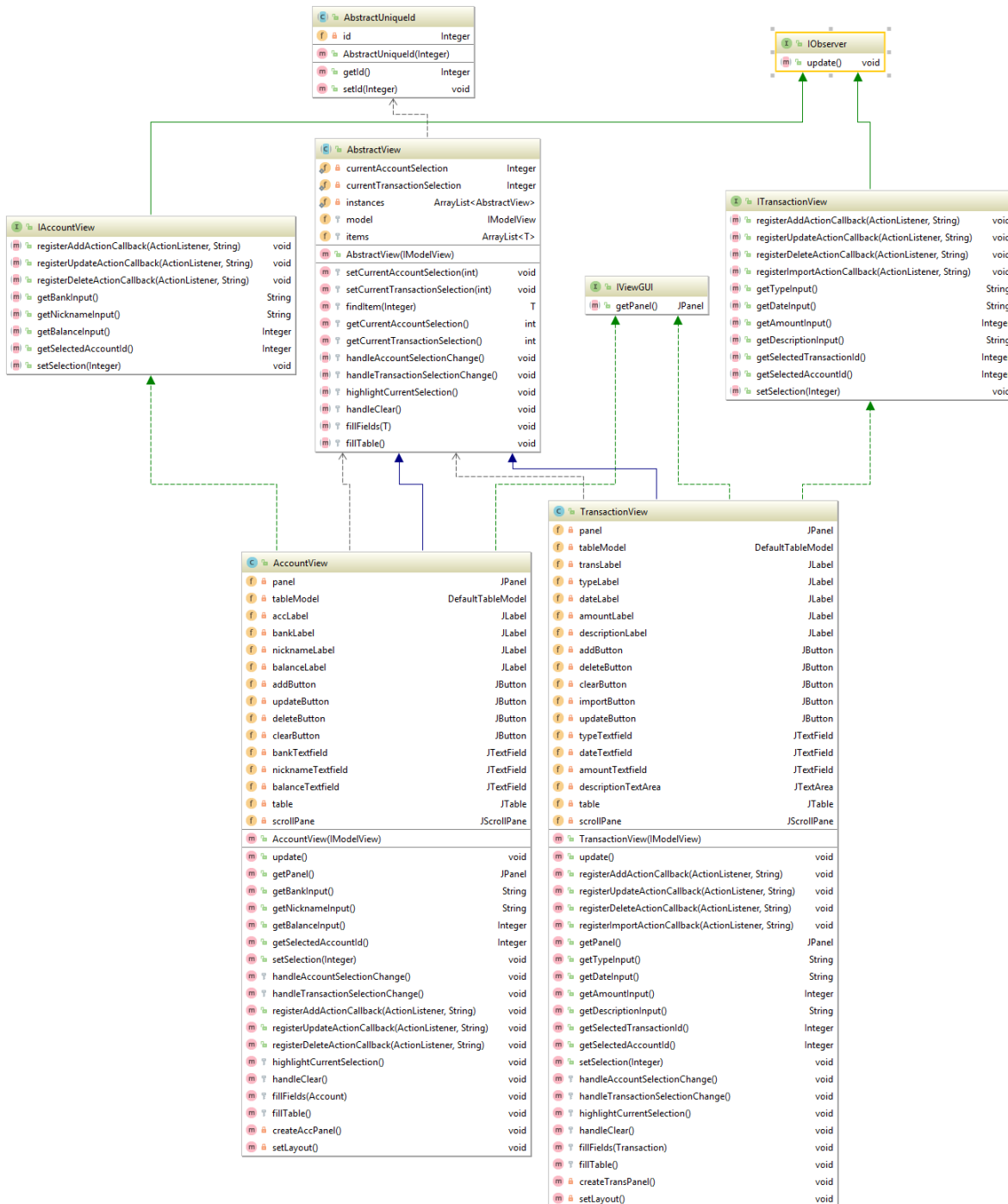


Figure 5: View subsystem class diagram

TODO - Put words here

3.3 Units Description

TODO – Generete from JavaDoc for each class in the subsystem

3.4 Controller

3.4.1 Design Diagram

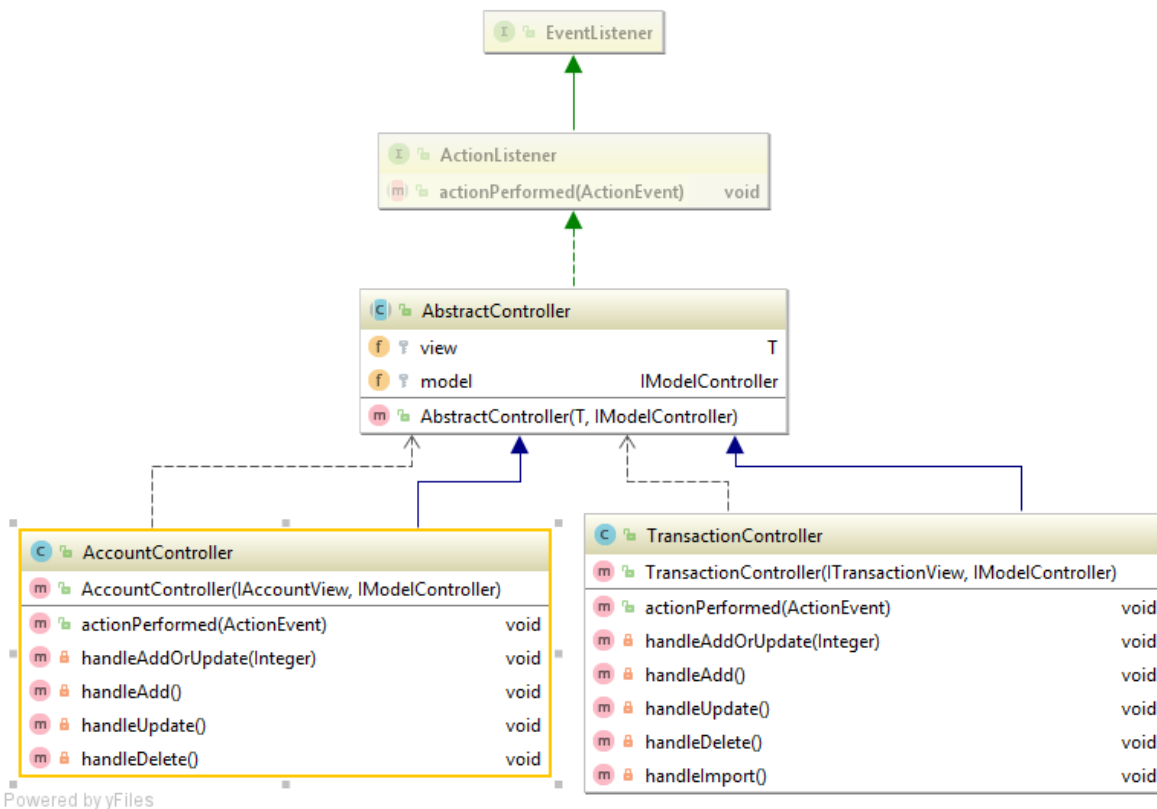


Figure 6: Controller subsystem class diagram

TODO - Put words here

3.4.2 Units Description

TODO – Generete from JavaDoc for each class in the subsystem

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` to 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.

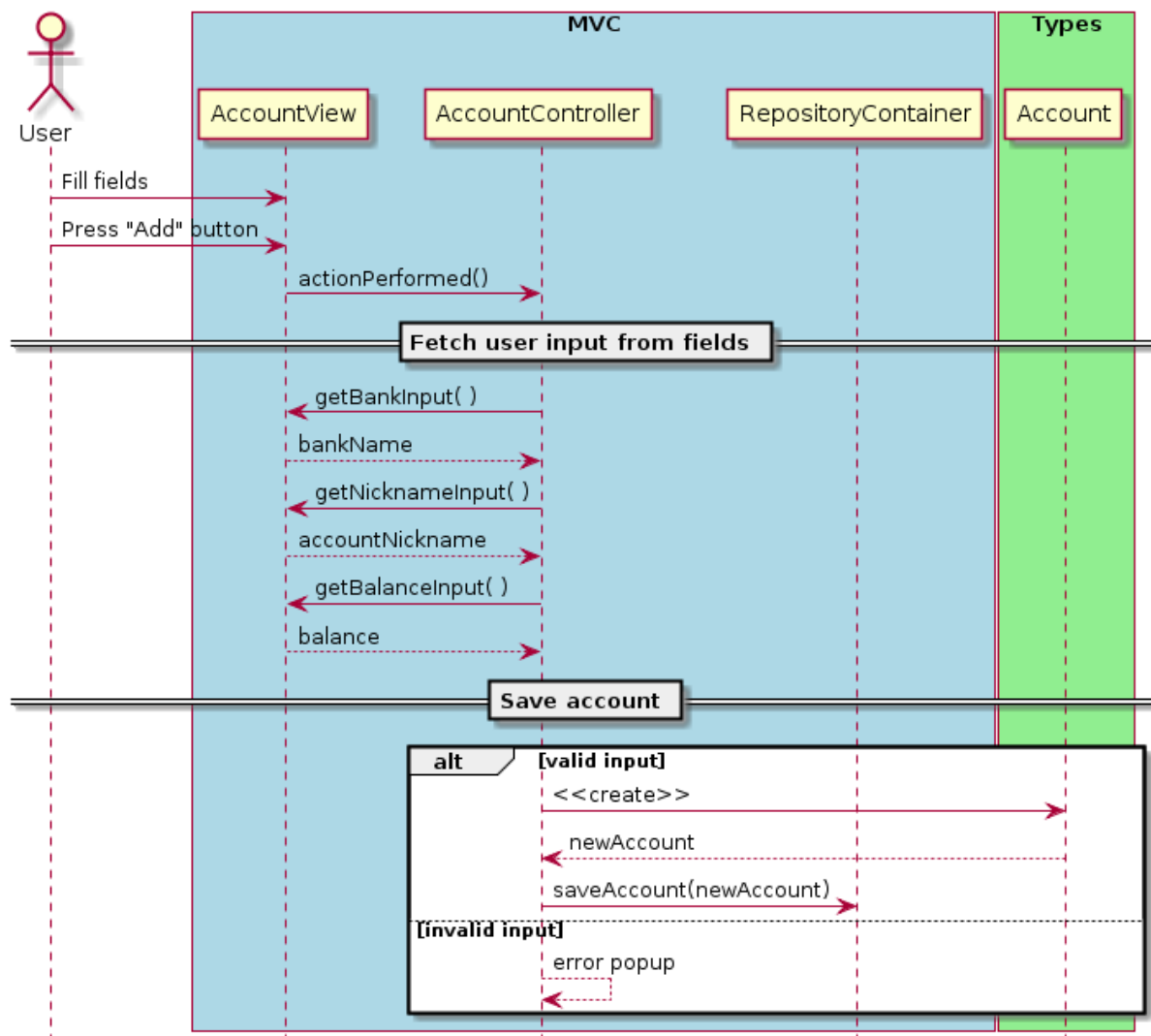


Figure 7: Adding an account

4.2 Update an account

Updating an 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.

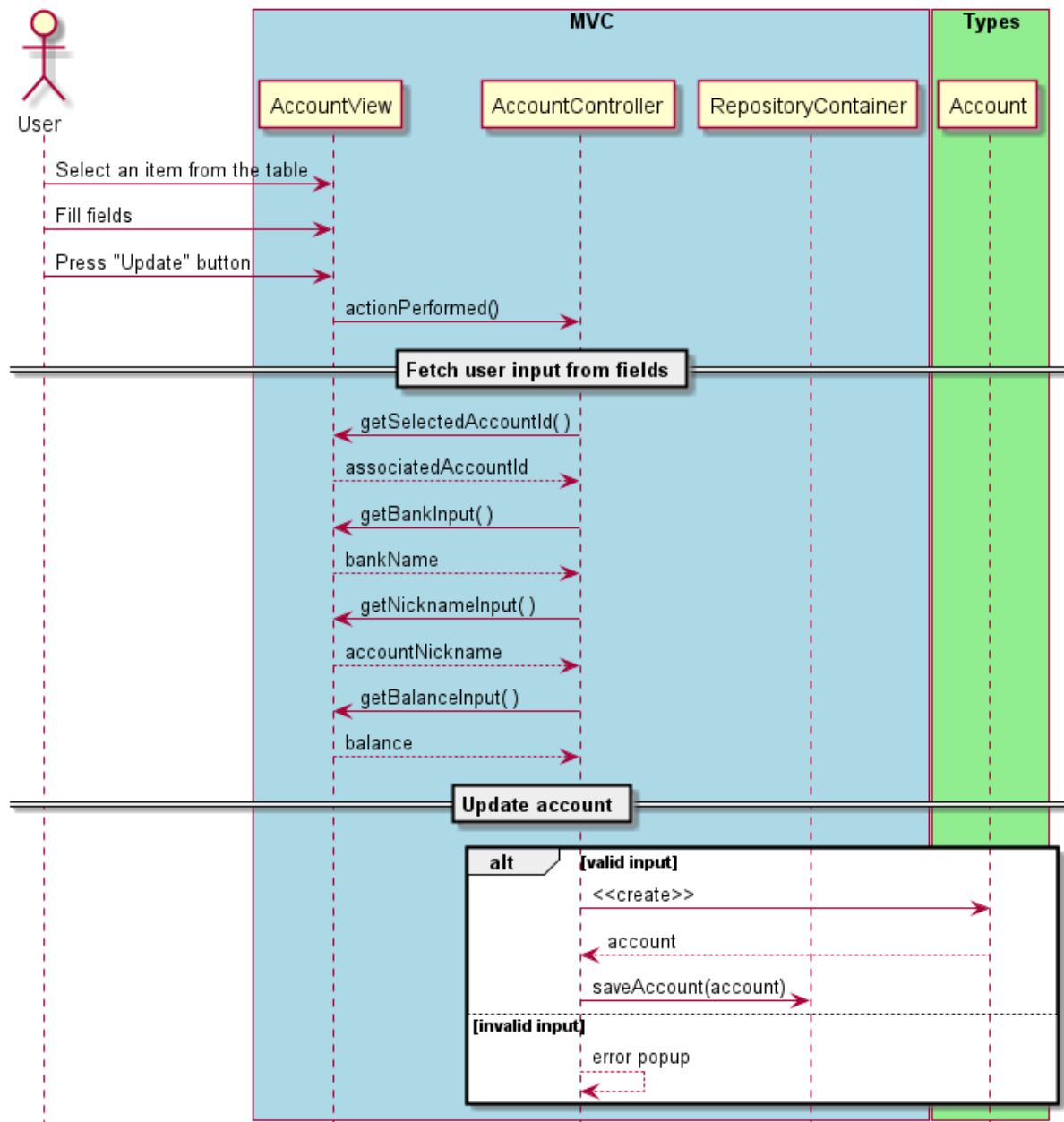


Figure 8: Updating an account

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.

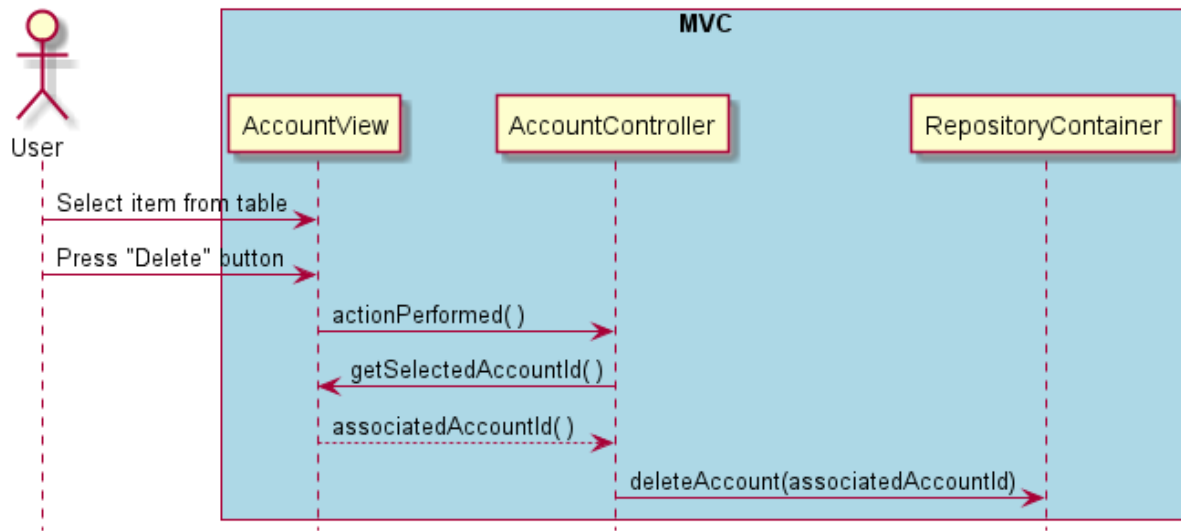


Figure 9: Deleting an account

4.4 Import a transaction list

In this scenario, the user 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.

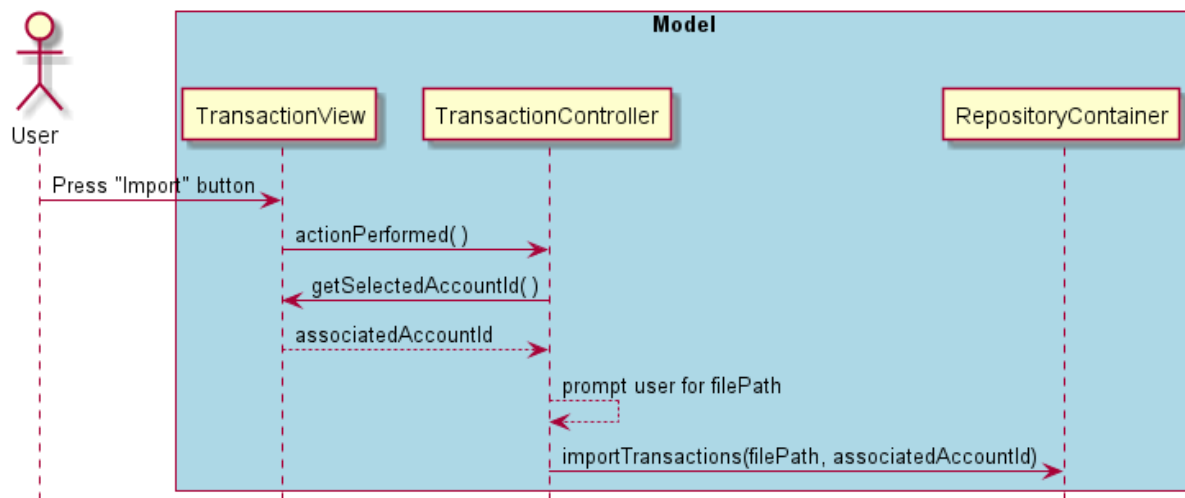


Figure 10: Import a list of transactions from .csv file

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.

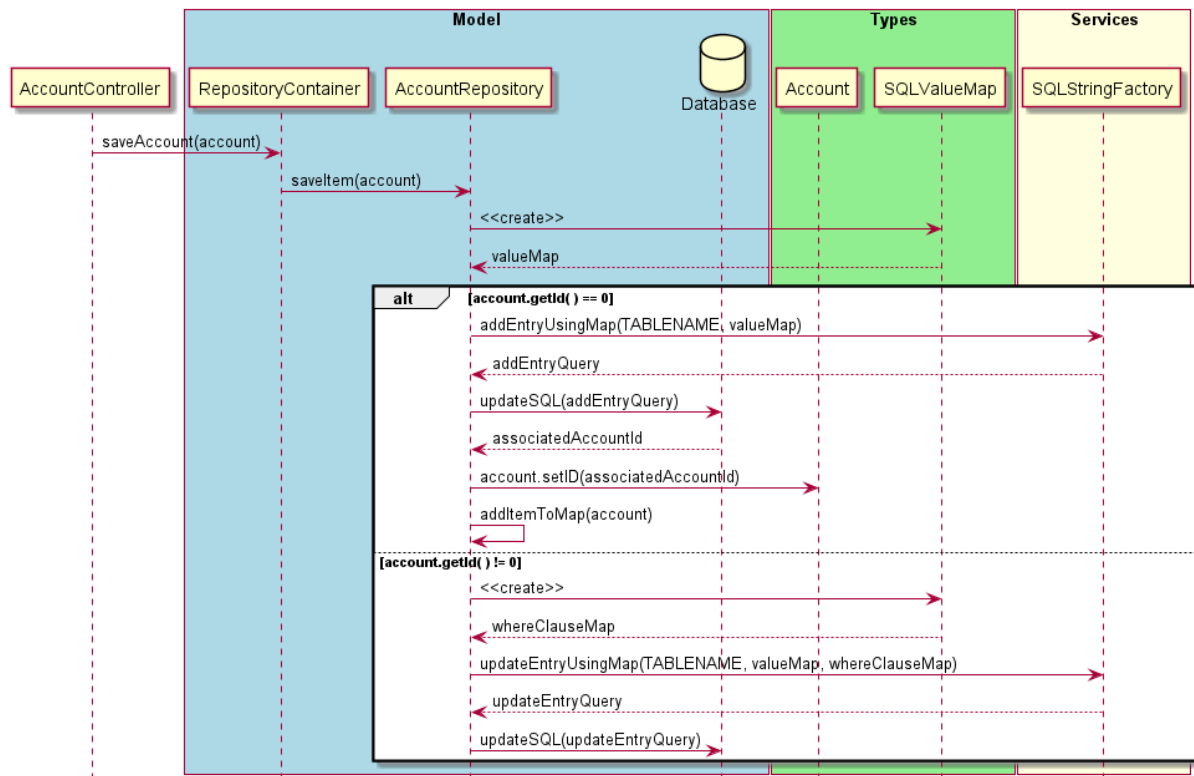


Figure 11: Model - Saving an account

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.

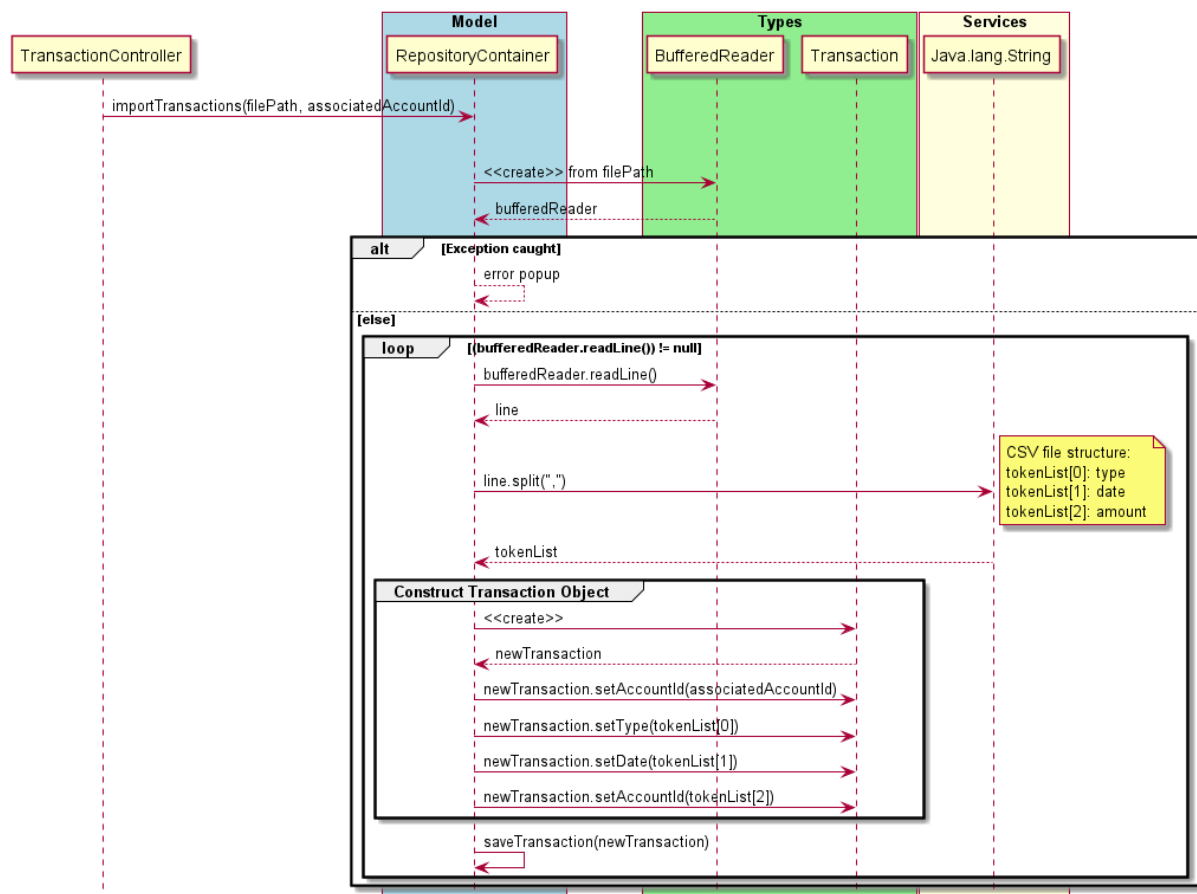


Figure 12: Model - Import list of transactions from .csv file