SYSTEMS PROGRAMMING AND SCRIPTING

FLORIAN BERGMANN

Assessment One: Stock Manager

[October 18, 2010 at 15:44]

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

CSV comma-separated values

GUI graphical user interface

MBA Management of Bank Accounts

MDA Management of Data Access

MSI Management of Stock Items

MVP model-view-presenter

Part I

DEVELOPMENT OF A STOCK MANAGER APPLICATION

1 INTRODUCTION

In chapter an overview over the document, as well as the specified requirements shall be given.

1.1 DOCUMENT OVERVIEW

This report fulfils in major parts the role of a requirements document. As such, it is intended for different audiences: chapter 2 provides an overview over the fulfilled requirements and thus should be of greatest interest for the managerial department, as well as the end users.

chapter 4 is a user guide that showcases the use of the program by showing how to accomplish certain tasks with the application. Naturally part is essential for end users.

The chapters chapter 3 and chapter 5 are intended for engineers and software developers. They provide an overview over the application's high- and low-level design, highlighting certain important aspects that might need to be taken into account to allow further development to proceed at an efficient pace.

chapter 6 provides an overview over the testing that has happened during the development.

chapter 7 will wrap up the development of the application and provide an outlook at possible improvements that might be made.

1.2 REMIT

section shall provide a short recap of the specified requirements. A list of fulfilled requirements will be provided in chapter 2.

The requirements as understood by the contractor are as following [1]:

MSI01: Allow the management of *Stock Items*. Management includes the following operations: *add*, *edit*, *delete*.

MSI02: The operation *add* and *delete* should be possible without the use of an external storage.

1 For further reference the requirements are prefixed with unique numbers: Management of Stock Items (MSI), Management of Bank Accounts (MBA), Management of Data Access (MDA), graphical user interface (GUI)

MSI03: Every stock item should consist of the following attributes: a *Stock Code*, an *Item Name*, a *Supplier Name*, a *Unit's cost*, the *Number Required* and the *Current stock level*.

MSI04: Allow the ordering of stock items via a money transfer.

MBA01: Allow the management of *Bank Accounts*: Management includes the following operations: add, edit, delete.

MBA02: The real transaction of money needs **not** to be implemented.

MBA03: An order should deduct the needed money from the bank account and change the *required* and *current* stock of an item accordingly.

MDA01: Allow the import and export of *Stock Items* from comma-separated values (CSV)-file.

MDA02: The location of the file may be chosen by the user.

MDA03: The ordering of the CSV-file may not be changed.

MDA04: The ordering of the files is as follows:

```
1 StockCode,Name,SupplierName,UnitCost,RequiredStock,CurrentStock
```

MDA04: The file should support blank fields by not entering data between two commas.

GUI01: Interaction between user and program shall happen via a GUI.

GUI02: The GUI shall provide menus, buttons and icons for easier accessibility.

2 REQUIREMENT'S CHECKLIST

From the requirements stated in section 1.2, the following were fulfilled:

MSI01: Implemented in StockItem class with getters and setters.

MSI02: Implemented in a manager-class that allow adding and deleting.

MSI03: Implemented as StockItem class.

MSI04: Implemented in manager-class.

MBA01: Implemented in BankAccount class with getters and setters.

MBA02: Fake-method for ordering: will adjust account balance, but not transfer money.

MBA03: Implemented in manager-class: takes care of Atomicity of request.

MDA01: Implemented in FileHandler-class and StockItem-class.

MDA02: Implemented in FileHandler-class.

MDA03: Implemented in StockItem-class.

MDA04: Implemented in StockItem-class.

MDA04: Implemented in StockItem-class.

GUI01: Implemented via WinForms.

GUI02: Implemented via WinForms.

Apart from fulfilling these requirements the following features were implemented as well to improve the user-experience of the program:

ERROR NOTIFICATION: Upon entering invalid information the user will be informed about the mistakes.

BANKACCOUNT PERSISTENCE: It is possible to import and export bank accounts as well.

ORDER QUANTITY: It is possible to order a certain quantity instead of always ordering the required number of items.

3 DESIGN CONSIDERATIONS

4 USER GUIDE

In this chapter the most common use-cases of the program will be explained. These include:

1. Managing (adding, deleting, editing) a stock item or bank account.
2. Placing an order.
3. Importing and exporting data.

4.1 MANAGE STOCK ITEMS AND BANK ACCOUNTS

Upon starting the application the main window will be displayed. The main window hosts all necessary controls for the first three use cases.

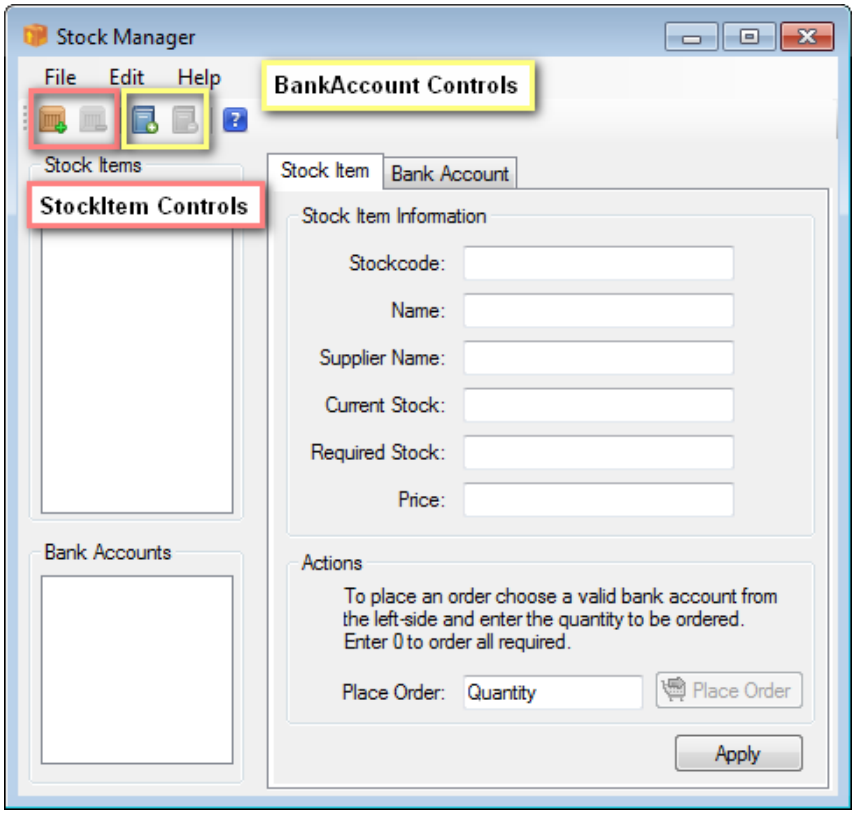

Figure 1: Main Window

To add a stock item or a bank account a click on the appropriate button is necessary:

Figure 2: Add a stock item.

Icon to add a new stock item to the application: the item will be inserted into the stock item list with dummy values.

Icon to add a new bank account to the application: the account will be inserted into the bank account list with dummy values.

Figure 3: Add a bank account.

After inserting a new stock item or bank account, the item can be chosen in the appropriate list (on the left-hand side of the application). By clicking an item, the appropriate panel will be show up, where the values can be edited.

Editing needs to be completed by clicking the *apply*-button. If any incorrect values were entered, the application will inform the user about the occured mistakes.

To manage a bank account there are two more possible commands the user can issue: apart from changing the values, it is possible to deposit or withdraw money from the bank account. Therefore the user simply has to enter a number in the correct field and press the accompanying button.

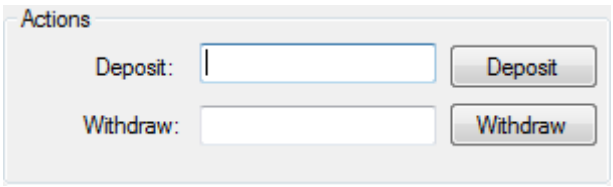

Figure 4: Depositing and withdrawing money

4.2 PLACING AN ORDER

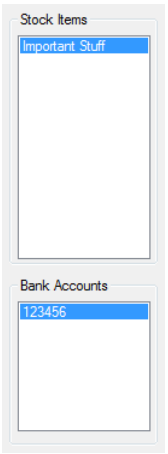

Figure 5: Selection of items.

To place an order the user has to select a bank account and a stock item from the lists (an item needs to be highlighted in both lists).

Then a value can be entered inside the *quantity*-box: either the amount of items to be ordered, or 0. By entering 0 the program will try to order the *required amount*.

If enough funds are available the order will be placed and the stock information will be updated. If not enough funds are available the application will output an error message.

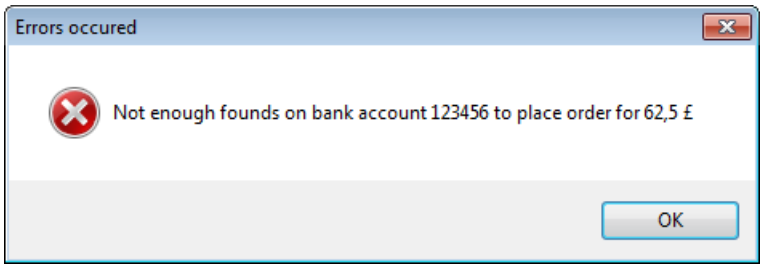

Figure 6: Placing an order without the needed funds

4.3 IMPORTING & EXPORTING DATA

After entering stock items and bank accounts it is possible to save them to a file and open them again for later use.

Therefore the user has to choose the appropriate option from the file menu or set standard-paths and click the menu-bar icon.

4.3.1 *File menu*

To save or load only one of the list the user selects `File` ⇒ `Save (Open)` ⇒ `Save (open) bank accounts / Save (Open) stock items`.

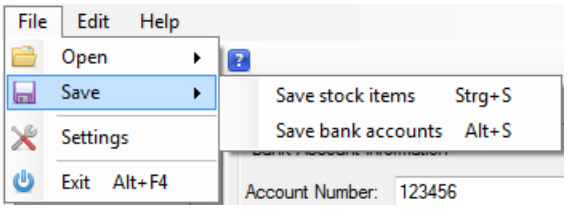

Figure 7: Saving via file menu

4.3.2 *Menu-bar icon*

To save via the menu icon it is necessary to first set default file paths for the files [1]. The paths can be set in the `Settings window` found under `File` ⇒ `Settings`.

1 As soon as these paths are set, the application will also attempt to load items and bank account on start-up.

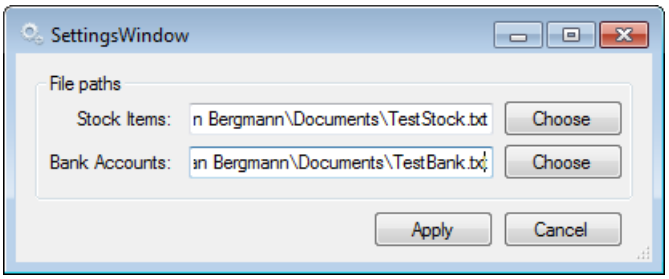

Figure 8: Settings window

After setting these paths both list can be saved with a single click on the menu-bar icon (denoted by two disks).

5 DEVELOPER GUIDE

To allow further development of the application, the design will be described in a top down approach, starting from a high level overview and going into details only where necessary.

5.1 ARCHITECTURAL OVERVIEW

The application was developed taking into account the principle of separating program logic from interface design. To support this approach the model-view-presenter (MVP) design pattern was utilised.

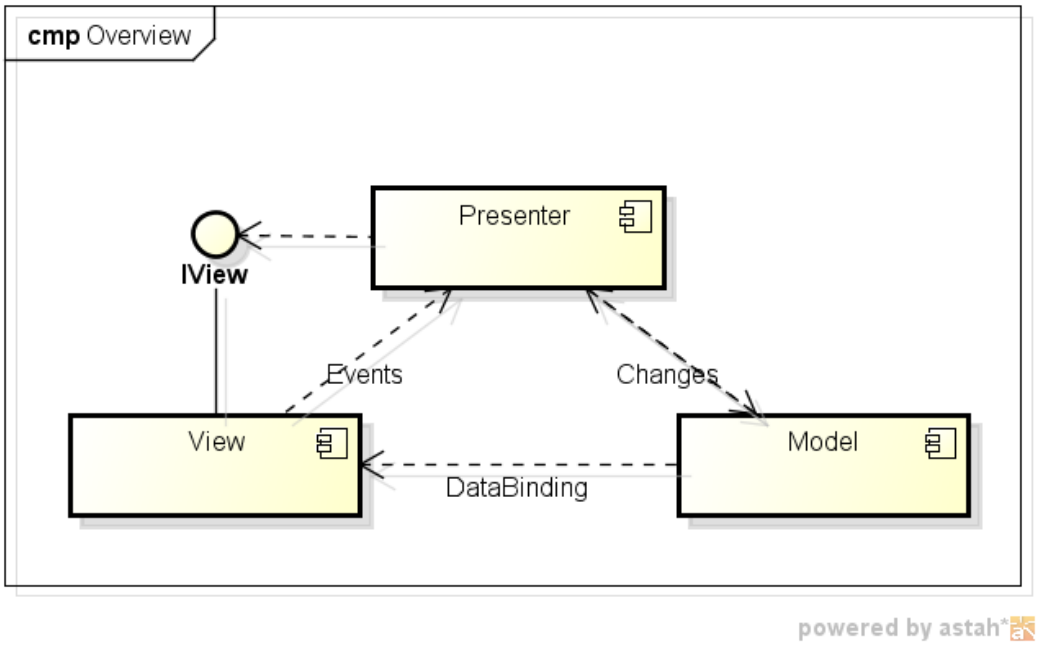

Figure 9: Architecture overview with MVP

This separation allows the view to be decoupled from the presenter and the model, as the presenter only communicates with the view over a well-defined interface (`IView`), whereas the view communicates only with provided methods from the presenter.

When changes occur in the model, the view will be notified via the data-binding mechanism of WinForms.

The implementation of this pattern splits the application into two projects:

USERINTERFACE: Hosts the graphical user interface and all code related to changing the appearance of the application.

APPLICATIONLOGIC: Hosts the `presenter` and the `model` component of the diagram.

These two packages will be described in-depth:

5.2 USERINTERFACE

The user interface package holds the WinForms representation of a possible GUI[1].

The `MainWindow` holds a reference to the `presenter` and the `model`.

The `presenter` handles all events that need more logic than just changing the view's appearance[2]

The `model`-reference is used to set-up the data-binding in the application:

```
1  private void SetUpDataBindings()
2          {
3              stockItemsListBox.DataSource = _Model.StockItems;
4              stockItemsListBox.DisplayMember = "Name";
5
6              /*
7               * The datasourceupdatemode is set to "Never".
8               * This leads to the ability to enforce the use of the presenter
                    to update the values in the model.
9               * This way the validation errors can be handled by the presenter
                    thus leading to better seperation of concerns.
10              */
11             stockCodeTextBox.DataBindings.Add("Text", _Model.StockItems, "
                   StockCode", false, DataSourceUpdateMode.Never);
12             itemNameTextBox.DataBindings.Add("Text", _Model.StockItems, "Name"
                   , false, DataSourceUpdateMode.Never);
13             supplierNameTextBox.DataBindings.Add("Text", _Model.StockItems, "
                   SupplierName", false, DataSourceUpdateMode.Never);
14             currStockTextBox.DataBindings.Add("Text", _Model.StockItems, "
                   CurrentStock", false, DataSourceUpdateMode.Never);
15             reqStockTextBox.DataBindings.Add("Text", _Model.StockItems, "
                   RequiredStock", false, DataSourceUpdateMode.Never);
```

1 It is a *possible* GUI, as another one should - due to the decoupling of view and logic - be easily realisable by implementing the `interfaces` of the `ApplicationLogic` package.

2 E.g. enabling/disabling buttons, changing the color of fields, showing a new window.

```
16              priceTextBox.DataBindings.Add("Text", _Model.StockItems, "UnitCost
                    ", false, DataSourceUpdateMode.Never);
17
18              bankAccountsListBox.DataSource = _Model.BankAccounts;
19              bankAccountsListBox.DisplayMember = "AccountNumber";
20
21              accountNumberTextBox.DataBindings.Add("Text", _Model.BankAccounts,
                     "AccountNumber", false, DataSourceUpdateMode.Never);
22              nameTextBox.DataBindings.Add("Text", _Model.BankAccounts, "Surname
                    ", false, DataSourceUpdateMode.Never);
23              balanceTextBox.DataBindings.Add("Text", _Model.BankAccounts, "
                    Balance", false, DataSourceUpdateMode.Never);
24          }
```

Listing 1: Data Binding of `view` and `model`

Noteworthy is the use of `DataSourceUpdateMode.Never`. This guarantees that changes from the GUI are not propagated to the model via data-binding, but that we can pass them through the presenter that will allow us to handle erroneous input.

Another important architectural aspect of the `view` is that the necessary interfaces for the presenter needs to be implemented: instead of passing all attributes with a method call, the presenter will expect the view to implement a certain interface through which it can access the needed attributes:

```
1  ï»¿using System;
2  namespace ApplicationLogic.Interfaces
3  {
4      /// <summary>
5      /// Utilized by the presenter to get the necessary values from a view.
6      /// </summary>
7      public interface IStockItemView
8      {
9          int CurrentStock { get; }
10         string ItemName { get; }
11         int RequiredStock { get; }
12         string StockCode { get; }
13         string SupplierName { get; }
14         double UnitCost { get; }
15     }
16 }
```

Listing 2: Example interface `IStockItemView`

The `MainWindow` implements three of these presenter-related interfaces: `IStockItemView`, `IBankAccountView`, `ICongregateView`. The first two guarantee the presenter that it

can access all attributes needed to update an item or bank account. The later view provides the following methods and properties:

```
ï»¿using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using ApplicationLogic.Model;

namespace ApplicationLogic.Interfaces
{
    /// <summary>
    /// Utilized by the presenter to get the necessary values from a view.
    /// </summary>
    public interface ICongregateView
    {
        StockItem StockItem { get; }

        BankAccount BankAccount { get; }

        int Quantity { get; }

        double Deposit { get; }

        double Withdraw { get; }

        bool ConfirmDelete();

        bool ConfirmClose();

        void DisplayValidationErrors(ErrorMessageCollection errorCollection);
    }
}
```

Listing 3: Interface `ICongregateView`

It allows to delete items and bank accounts, as well as provide the necessary application logic to order items, deposit and withdraw money as well as method to display possible validation errors.

Should new views be added an interface should be provided that guarantees the separation between the view and the presenter.

A problem arising from the .NET architecture is that only the GUI project provides a settings file. This leads to the fact that the WinForms-GUI has to handle the loading and saving of user-preferences (the file paths to the bank accounts and stock items files).

5.3 APPLICATIONLOGIC

The application logic project hosts the `presenter(s)` as well as the `models`.

An overview of the relation between the classes shall be provided for orientation purposes:

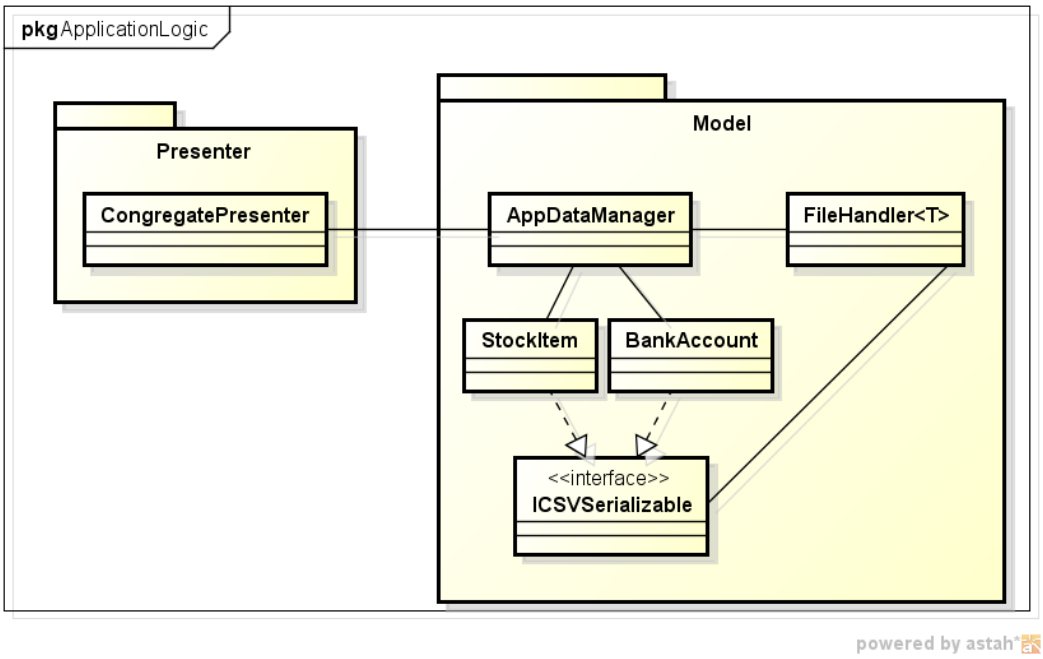

Figure 10: Abstract overview of project `application logic`

It can be seen that the `AppDataManager`-class works as a *facade* for the rest of the model.

Noteworthy implementation in this package are the `FileHandler` and the realisation of error handling.

5.3.1 *FileHandler*

The FileHandler-class utilizes the concept of generics: this way it is possible to reuse this class for multiple classes that need to be persisted.

To allow the serialization and de-serialization-logic to be separated from the file-handling logic, the FileHandler-class requires all classes that need to be persisted to implement the `ICSVSerializable`-interface:

```
1 ï»¿using System;
2 using System.Collections.Generic;
3 using System.Linq;
```

```
4  using System.Text;
5
6  namespace ApplicationLogic.Interfaces
7  {
8      /// <summary>
9      /// Used to ensure all objects will be able to persited via FileHandler
           class.
10     /// </summary>
11     /// <typeparam name="T"></typeparam>
12     public interface ICSVSerializable<T>
13     {
14         String CsvRepresentation();
15
16         T ParseFromString(String stringRepresentation);
17     }
18 }
```

Listing 4: Interface `ICSVSerializable`

Due to this fact - both - the `StockItem` and the `BankAccount`-class implement this interface.

5.3.2 *Error handling*

Error handling is achieved by separate classes called `ErrorMessageCollection` and `ErrorMessage`.

On validation errormessages will be added to an errormessagecollection that can be accessed from the presenter to display the occurred errors.

As an example the code of the StockItem's `Validate()` method, as well as a sequence-diagram of the calling sequence is shown:

```
1  public static bool Validate(String stockCode, String name, String supplierName
       , double unitCost, int required, int currentStock)
2          {
3              if (String.IsNullOrEmpty(stockCode) || !IsValidStockCode(stockCode
                   ))
4              {
5                  ErrorMessages.Add(new ErrorMessage("Need a stockcode that
                       adheres to the stockcode format: 4 numbers."));
6              }
7              if (String.IsNullOrEmpty(name))
8              {
9                  ErrorMessages.Add(new ErrorMessage("Need an item name."));
10             }
11             if (String.IsNullOrEmpty("supplierName"))
12             {
```

```
13                  ErrorMessages.Add(new ErrorMessage("Need a supplier name."));
14              }
15              if (unitCost < 0.0)
16              {
17                  ErrorMessages.Add(new ErrorMessage("Unit costs must be greater
                        or equal 0."));
18              }
19              if (required < 0)
20              {
21                  ErrorMessages.Add(new ErrorMessage("Required must be greater
                        or equal 0."));
22              }
23              if (currentStock < 0)
24              {
25                  ErrorMessages.Add(new ErrorMessage("Current must be greater or
                        equal 0."));
26              }
27              return ErrorMessages.Count == 0;
28          }
```

Listing 5: Validate method of StockItem

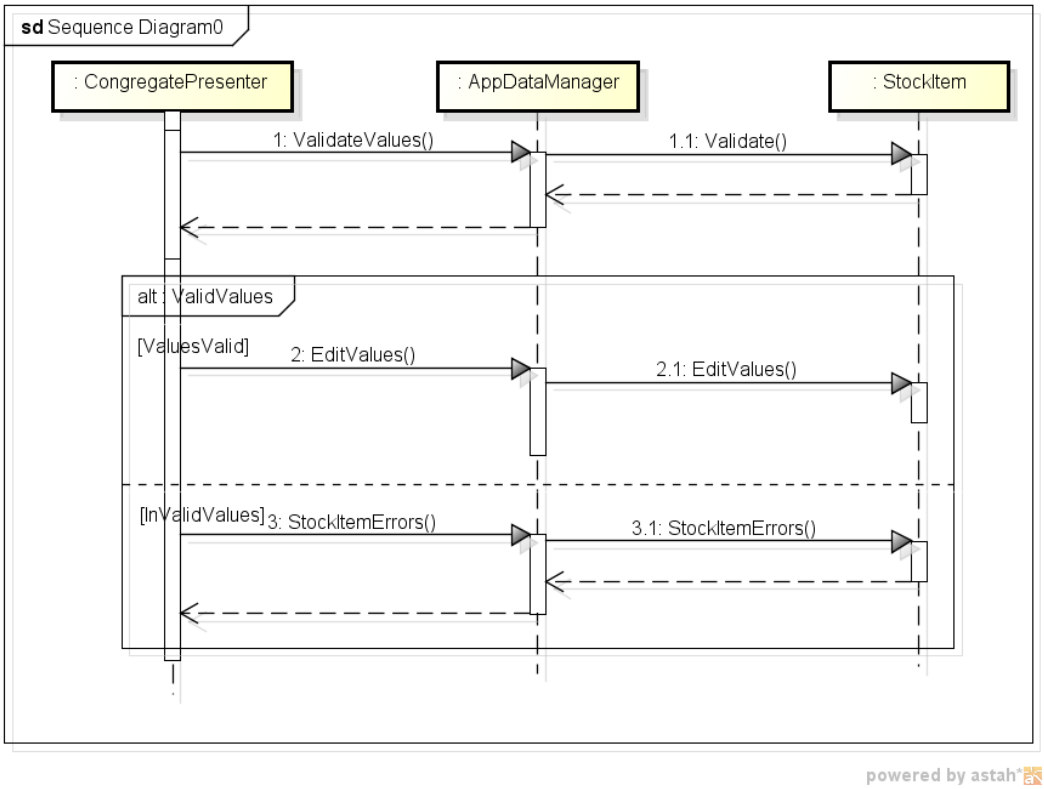

Figure 11: Sequence diagram of input validation

6 TESTING

7 CONCLUSIONS

Part II

APPENDIX

APPENDIX A