SYSTEMS PROGRAMMING AND SCRIPTING

FLORIAN BERGMANN

Assessment One: Stock Manager

CONTENTS

I DEVELOPMENT OF A STOCK MANAGER APPLICATION 1

1 INTRODUCTION 2
- 1.1 Document overview . . . 2
- 1.2 Remit . . . 2

2 REQUIREMENT'S CHECKLIST 4

3 DESIGN CONSIDERATIONS 6
- 3.1 Architectural overview . . . 6
- 3.2 User Interface . . . 7
- 3.3 Application Logic . . . 8

4 USER GUIDE 9
- 4.1 Manage stock items and bank accounts . . . 9
- 4.2 Placing an order . . . 10
- 4.3 Importing & exporting data . . . 11
  - 4.3.1 File menu . . . 11
  - 4.3.2 Menu-bar icon . . . 11

5 DEVELOPER GUIDE 13
- 5.1 UserInterface . . . 13
- 5.2 ApplicationLogic . . . 15
  - 5.2.1 File handler . . . 16
  - 5.2.2 Error handling . . . 16

6 TESTING 19

7 CONCLUSIONS 22

II APPENDIX 23

A APPENDIX: SOURCE CODE 24
- A.1 View . . . 24
- A.2 Application Logic . . . 36
  - A.2.1 Interfaces Package . . . 36
  - A.2.2 Model Package . . . 39
  - A.2.3 Presenter Package . . . 59
- A.3 Tests . . . 66

BIBLIOGRAPHY 75

LIST OF FIGURES

Figure 1 Architecture overview with model-view-presenter (MVP) . . . 6
Figure 2 Abstract overview of project `application logic` 8

Figure 3 Main Window 9
Figure 4 Add a stock item. 10
Figure 5 Add a bank account. 10
Figure 6 Depositing and withdrawing money 10
Figure 7 Selection of items. 10
Figure 8 Placing an order without the needed funds 11
Figure 9 Saving via file menu 11
Figure 10 Settings window 12

Figure 11 Sequence diagram of input validation 18

Figure 12 NUnit test case run 21

LIST OF TABLES

Table 1 Performed tests. 20

LISTINGS

Listing 1 Data Binding of `view` and `model` 13
Listing 2 Example interface `IStockItemView` 14
Listing 3 Interface `ICongregateView` 14
Listing 4 Interface `ICSVSerializable` 16
Listing 5 Validate method of StockItem 16

Listing 6 MainWindow.cs 24

Listing 7 Settings.cs 35
Listing 8 IBankAccountView.cs 36
Listing 9 ICongegrateView.cs 37
Listing 10 ICSVSerializable.cs 38
Listing 11 IStockItemView.cs 38
Listing 12 IViewModel.cs 38
Listing 13 AppDataManager.cs 39
Listing 14 BankAccount.cs 46
Listing 15 ErrorMessage.cs 51
Listing 16 ErrorMessageCollection.cs 51
Listing 17 FileHandler.cs 52
Listing 18 NoFilePathSetException.cs 54
Listing 19 NotEnoughFundsException.cs 54
Listing 20 StockItem.cs 54
Listing 21 CongregatePresenter.cs 59
Listing 22 BankAccountTest.cs 66
Listing 23 FileHandlerTest.cs 69
Listing 24 StockItemTest.cs 69
Listing 25 AppDataManagerTest.cs 72

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
TDD	test-driven development

Part I

DEVELOPMENT OF A STOCK MANAGER APPLICATION

1 INTRODUCTION

In this 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. This part is essential for end users.

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

This 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 follows [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*.

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:

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

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

MDA05: The application should be able to handle at least *100* items.

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 handles adding and deleting in-memory.

MSI03: Implemented in StockItem class.

MSI04: Implemented in AppDataManager-class.

MBA01: Implemented in BankAccount class with getters and setters.

MBA02: Fake-method for ordering: adjusts account balance, without transfer money.

MBA03: Implemented in AppDataManager-class.

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.

MDA05: Verified via testing.

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 by the GUI.

BANK ACCOUNT 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

This chapter should provide a very general overview over the developed system, mainly describing the employed architecture.

3.1 ARCHITECTURAL OVERVIEW

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

In this pattern the presenter separates the GUI from the logical part of the application. The view communicates with the model only through the presenter. However, the model can notify the view directly of data-changes if an observer-pattern or data-binding is employed[1].:

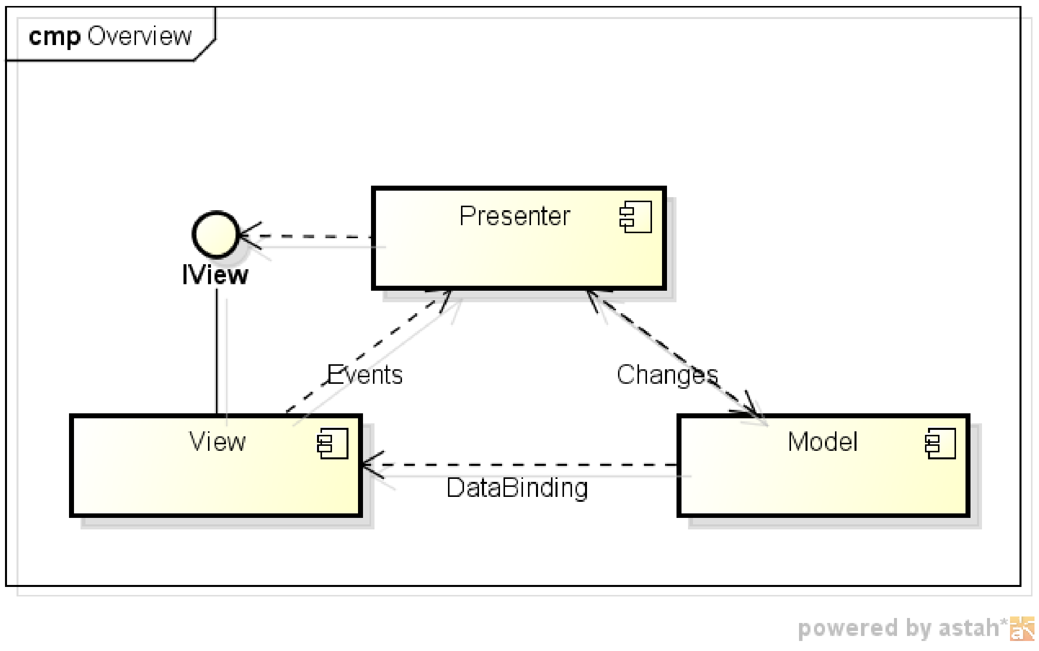


Figure 1: Architecture overview with MVP

1 More information can be found at Boodhoo (2006).

Noteworthy is the `IView`-interface that allows the presenter to collect all needed data from the GUI without knowing what kind of GUI was used. This can help in reusing the presenter for multiple application-front-ends like WinForms or ASP.NET.

In the implementation part changes that occur in the model, will be forwarded to the view via the data-binding mechanisms provided by 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.

Certain decisions made concerning these two packages will be described now, whereas greater detail will be put on implementation detail in Chapter 5.

3.2 USER INTERFACE

The GUI was developed completely in WinForms utilizing only standard controls provided by the .NET framework.

To always display accurate data from the model, data-binding was used to connect the view to the model (further information about the concrete implementation can be found in Section 5.1).

The GUI project itself handles all changes to the GUI-elements (color changes, displaying new windows, etc.), whereas the collection of input-data from the controls is performed in the presenter via interfaces (more information in Section 5.1).

3.3 APPLICATION LOGIC

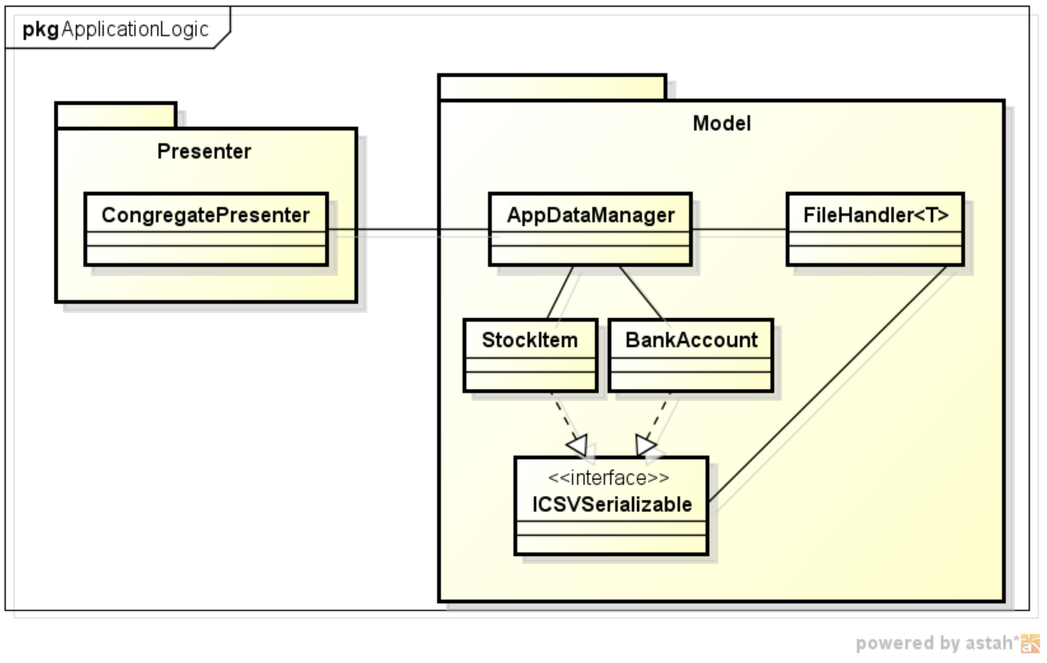


Figure 2: Abstract overview of project `application logic`

The application logic project was implemented in a very straight forward manner:

There is one class that handles all incoming requests - the *AppDataManager*. It coordinates statements as needed: e.g. check if enough money is present on bank account → place order → update stock item. Moreover the lists holding the stock items and bank accounts are managed by this class.

Naturally the classes for handling bank accounts and stock items are implemented in the application-logic-project, as well. Moreover, a generic file handler (see Section 5.2.1 for implementation details) and an error-handling-facility (see Section 5.2.2) were implemented.

Noteworthy is the fact that `BindingLists` were used in the `AppDataManager` to store the bank account and stock item lists, to allow the necessary data-binding with the view to work.

The `CongregatePresenter` seen in the picture is the connection point for the GUI part of the application and mostly forwards the requests to the `AppDataManager`.

4 USER GUIDE

In this chapter ways to achieve 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 two use cases.

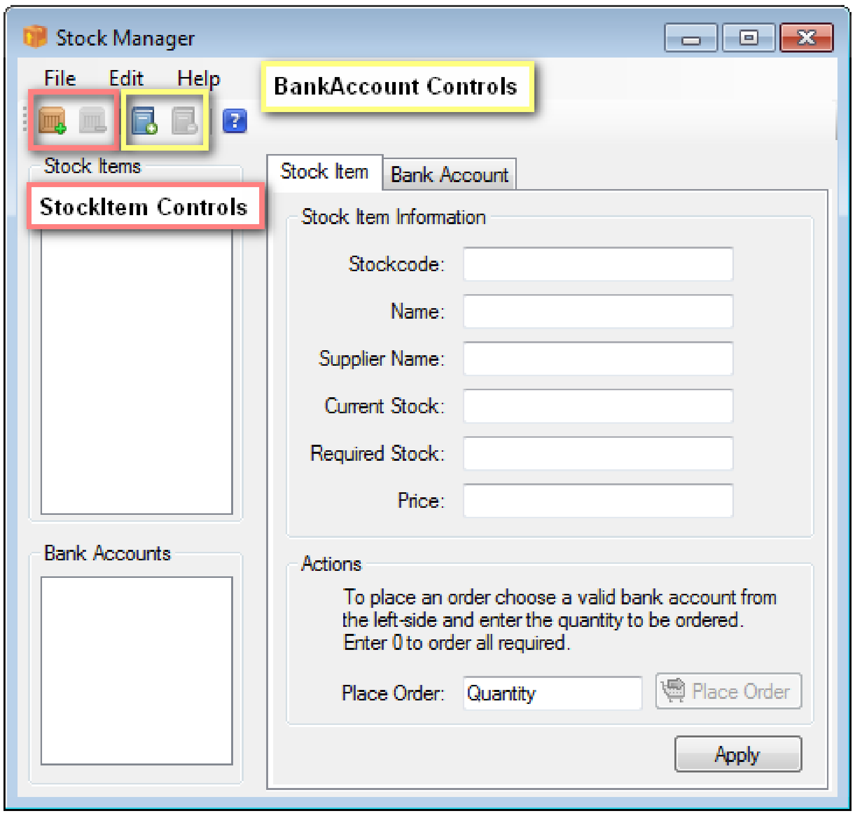


Figure 3: Main Window

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

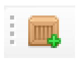


Figure 4: Add a stock item.

By clicking the icon to add a new stock item to the application, an item will be inserted into the stock item list with dummy values.

By clicking the icon to add a new bank account to the application, a account will be inserted into the bank account list with dummy values.

Figure 5: 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.

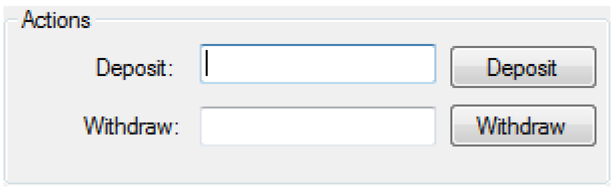


Figure 6: Depositing and withdrawing money

4.2 PLACING AN ORDER

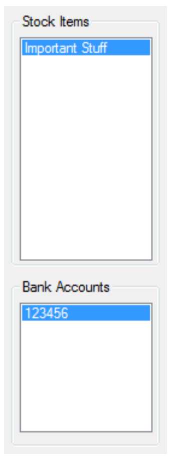
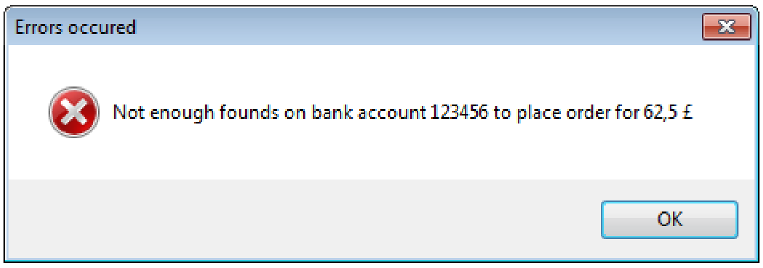


Figure 7: Selection of items.

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.


Errors occured

Not enough founds on bank account 123456 to place order for 62,5 £

Figure 8: 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`.

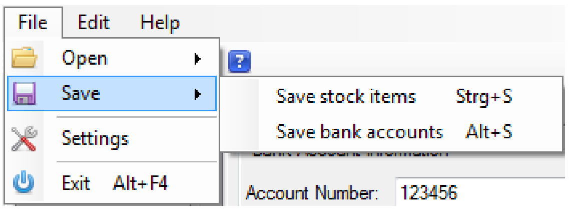
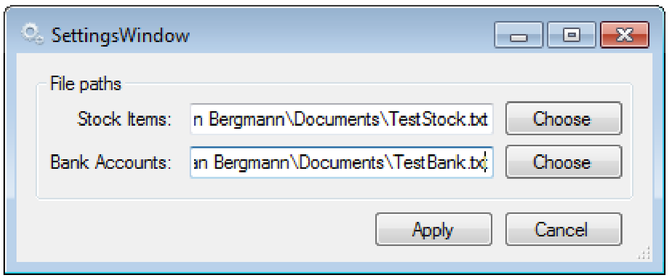


Figure 9: Saving via file menu

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.



Stock Items: n Bergmann\Documents\TestStock.txt

Bank Accounts: an Bergmann\Documents\TestBank.txt

Figure 10: Settings window

After setting these paths both lists 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, certain design decisions from Chapter 3 will be described in greater depth:

5.1 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:

```
private void SetUpDataBindings()
        {
            stockItemsListBox.DataSource = _Model.StockItems;
            stockItemsListBox.DisplayMember = "Name";

            /*
             * The datasourceupdatemode is set to "Never".
             * This leads to the ability to enforce the use of the presenter to update
                 the values in the model.
             * This way the validation errors can be handled by the presenter thus
                leading to better seperation of concerns.
             */
            stockCodeTextBox.DataBindings.Add("Text", _Model.StockItems, "StockCode",
                false, DataSourceUpdateMode.Never);
            itemNameTextBox.DataBindings.Add("Text", _Model.StockItems, "Name", false,
                 DataSourceUpdateMode.Never);
            supplierNameTextBox.DataBindings.Add("Text", _Model.StockItems, "
                SupplierName", false, DataSourceUpdateMode.Never);
            currStockTextBox.DataBindings.Add("Text", _Model.StockItems, "CurrentStock
                ", false, DataSourceUpdateMode.Never);
            reqStockTextBox.DataBindings.Add("Text", _Model.StockItems, "RequiredStock
                ", false, DataSourceUpdateMode.Never);
            priceTextBox.DataBindings.Add("Text", _Model.StockItems, "UnitCost", false
                , DataSourceUpdateMode.Never);

            bankAccountsListBox.DataSource = _Model.BankAccounts;
            bankAccountsListBox.DisplayMember = "AccountNumber";
```

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.

```

            accountNumberTextBox.DataBindings.Add("Text", _Model.BankAccounts, "
                AccountNumber", false, DataSourceUpdateMode.Never);
            nameTextBox.DataBindings.Add("Text", _Model.BankAccounts, "Surname", false
                , DataSourceUpdateMode.Never);
            balanceTextBox.DataBindings.Add("Text", _Model.BankAccounts, "Balance",
                false, DataSourceUpdateMode.Never);
        }
```

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 and keep the separation between view and model intact.

Another important architectural aspect of the `view` is the implementation of necessary interfaces for the presenter: 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:

```
ï»¿using System;
namespace ApplicationLogic.Interfaces
{
    /// <summary>
    /// Utilized by the presenter to get the necessary values from a view.
    /// </summary>
    public interface IStockItemView
    {
        int CurrentStock { get; }
        string ItemName { get; }
        int RequiredStock { get; }
        string StockCode { get; }
        string SupplierName { get; }
        double UnitCost { get; }
    }
}
```

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 view and presenter, thus allowing the possible reuse of the presenter across multiple GUIs.

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).

5.2 APPLICATIONLOGIC

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

As has been pointed auto in Chapter 3, the `AppDataManager`-class works as a *facade* for the rest of the model.

Noteworthy implementations in this package are the `file handler` and the realisation of `error handling`.

5.2.1 *File handler*

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:

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

namespace ApplicationLogic.Interfaces
{
    /// <summary>
    /// Used to ensure all objects will be able to persited via FileHandler class.
    /// </summary>
    /// <typeparam name="T"></typeparam>
    public interface ICSVSerializable<T>
    {
        String CsvRepresentation();

        T ParseFromString(String stringRepresentation);
    }
}
```

Listing 4: Interface `ICSVSerializable`

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

5.2.2 *Error handling*

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

On validation error-messages will be added to an error-message-collection that can be accessed from the presenter to display the occurred errors.

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

```
public static bool Validate(String stockCode, String name, String supplierName, double
     unitCost, int required, int currentStock)
        {
            if (String.IsNullOrEmpty(stockCode) || !IsValidStockCode(stockCode))
            {
                ErrorMessages.Add(new ErrorMessage("Need a stockcode that adheres to
                    the stockcode format: 4 numbers."));
            }
            if (String.IsNullOrEmpty(name))
            {
                ErrorMessages.Add(new ErrorMessage("Need an item name."));
            }
            if (String.IsNullOrEmpty("supplierName"))
            {
                ErrorMessages.Add(new ErrorMessage("Need a supplier name."));
            }
            if (unitCost < 0.0)
            {
                ErrorMessages.Add(new ErrorMessage("Unit costs must be greater or equal 0."));
            }
            if (required < 0)
            {
                ErrorMessages.Add(new ErrorMessage("Required must be greater or equal 0."));
            }
            if (currentStock < 0)
            {
                ErrorMessages.Add(new ErrorMessage("Current must be greater or equal 0."));
            }
            return ErrorMessages.Count == 0;
        }
```

Listing 5: Validate method of StockItem

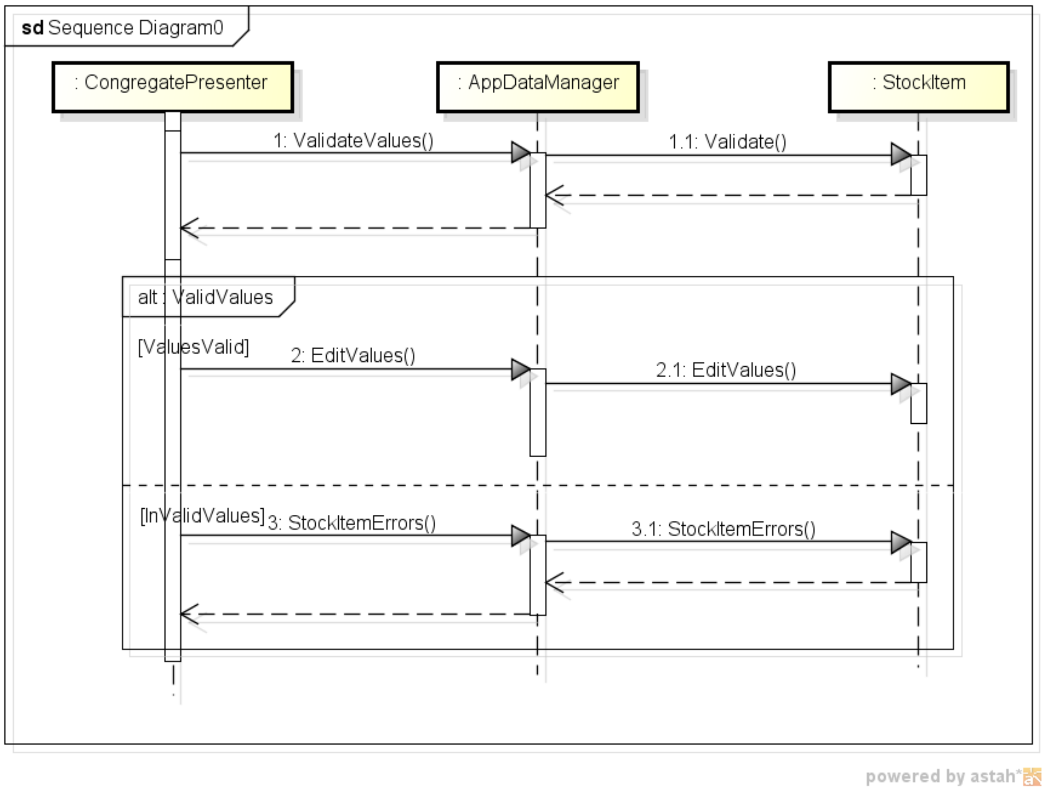


Figure 11: Sequence diagram of input validation

6 TESTING

Testing was performed in two stages:

1. Unit-Testing
2. System-Integration-Testing

The part of sub-system-integration-testing as lined out in Sommerville (2006, p. 520) was skipped, as there was no significant number of sub-modules that needed testing. Most of the integration consisted of passing on parameters through the three layers of the application (view, presenter, model).

These tests were incorporated into the main system-integration-testing.

As the model was the part that hosted most of the application logic, test-cases were created for its four main-classes:

1. `StockItem`
2. `BankAccount`
3. `FileHandler`
4. `AppDataManager`

The concrete error-cases that were tested are:

Under Test	Input	Output
Stock Item	Invalid StockCode	ArgumentException
Stock Item	Invalid Current Stock	ArgumentException
Stock Item	Invalid Required Stock	ArgumentException
Stock Item	Invalid Parsing	FormatException
Stock Item	Invalid Cost	ArgumentException
Bank Account	Invalid Balance	ArgumentException
Bank Account	Invalid Withdraw (Value too small)	ArgumentException
Bank Account	Invalid Withdraw (Value too high)	NotEnoughFundsException

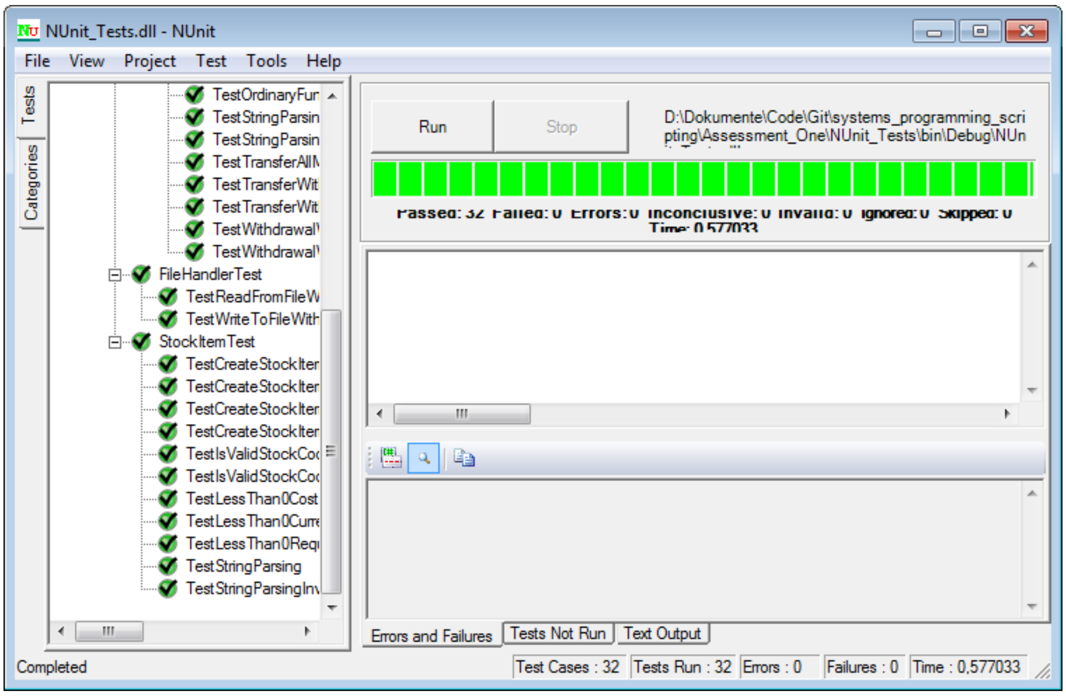
Bank Account	Invalid Deposit (Value too small)	ArgumentException
Bank Account	Invalid Transfer (Value too small)	ArgumentException
Bank Account	Invalid Transfer (Value too high)	NotEnoughFundsException
Bank Account	Too little funds for transfer	NotEnoughFundsException
Bank Account	Invalid Parsing	FormatException
AppDataManager	Remove invalid item	ArgumentException
AppDataManager	Perform Order	Correct sequence of statements
AppDataManager	Order with too little funds	NotEnoughFundsException
File Handler	Save without file path	NoFilePathSetException
File Handler	Load without file path	NoFilePathSetException

Table 1: Performed tests.

Moreover test cases were written to ensure correct behaviour for correct input. A complete listing of all test cases can be found under Section A.3[1].

Upon delivery a comprehensive suite of passing test cases is provided:

1 Running the test cases will require the NUnit (`http://www.nunit.org/`)- and Moq (`http://code.google.com/p/moq/`)-libraries.



```
TestOrdinaryFur
TestStringParsin
TestStringParsin
TestTransferAllN
TestTransferWit
TestTransferWit
TestWithdrawal\
TestWithdrawal\
FileHandlerTest
    TestReadFromFileW
    TestWriteToFileWith
StockItemTest
    TestCreateStockIter
    TestCreateStockIter
    TestCreateStockIter
    TestCreateStockIter
    TestIsValidStockCoc
    TestIsValidStockCoc
    TestLessThan0Cost
    TestLessThan0Curre
    TestLessThan0Requ
    TestStringParsing
    TestStringParsingInv
```

```
D:\Dokumente\Code\Git\systems_programming_scri
pting\Assessment_One\NUnit_Tests\bin\Debug\NUn

Passed: 32  Failed: 0  Errors: 0  Inconclusive: 0  Invalid: 0  Ignored: 0  Skipped: 0
Time: 0.577033
```

```
Completed          Test Cases : 32   Tests Run : 32   Errors : 0   Failures : 0   Time : 0,577033
```

Figure 12: NUnit test case run

7 CONCLUSIONS

After developing the application and looking at the resulting design, there are a few things that might have been accomplished differently and produced a better and cleaner design.

The data-binding part - due to the removal of the back-write mechanism (so the view does not automatically update the model) was mostly unnecessary and restricted the choices of collections in the model-part to `BindingLists`, whereas the model should actually be independent of the views.

Another problem arising from the `BindingLists` is that they can not be returned as read-only and this way can provide write access to the model. However, the lists were still implemented as public properties to allow unit-testing.

Another fact that became clear while testing was that the auto-creation of stock items and bank accounts in the model was unfortunate to test the correct behaviour of adding and deleting items. One could not easily add a specific item and delete the same one, but let the model create one, acquire a reference to it, delete it and test if the list was changed accordingly. A thorough test-driven development (TDD)-approach might have circumvented these shortcomings.

However, even with these shortcomings the application should fulfil the requirements. Moreover, the implementation of the generic file handler should be reusable to persist all kinds of objects - even in other projects.

And with the provided test-cases many cases should be covered that should allow a thorough refactoring of the application if the need arises.

Part II

APPENDIX

A APPENDIX: SOURCE CODE

A.1 VIEW

```
ï»¿using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Configuration;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using ApplicationLogic.Interfaces;
using ApplicationLogic.Presenter;
using ApplicationLogic.Model;
using System.Collections.Specialized;
using Assessment_One.Properties;

namespace Assessment_One
{
    public partial class MainWindow : Form, ICongregateView, IStockItemView,
        IBankAccountView
    {
        private CongregatePresenter _Presenter;
        private IViewModel _Model;
        private List<Control> backgroundColorChanged;

        private const int STOCKITEMTAB = 0;
        private const int BANKACCOUNTTAB = 1;

        /// <summary>
        /// Initializes a new instance of the <see cref="MainWindow"/> class.
        /// </summary>
        public MainWindow()
        {
            InitializeComponent();
            this._Model = new AppDataManager();
            _Presenter = new CongregatePresenter(this, this, this, this._Model);
            this.backgroundColorChanged = new List<Control>();
            LoadFilePathSettings();
            SetUpDataBindings();
        }



        #region EventHandler

        private void bankAccountsListBox_SelectedIndexChanged(object sender, EventArgs
             e)
        {
            if (this.bankAccountsListBox.SelectedItem != null)
            {
                this.tabControl1.SelectTab(BANKACCOUNTTAB);
                this.SwitchBankAccountControls(true);
            }
            else
            {
                this.SwitchBankAccountControls(false);
            }
        }

        private void stockItemsListBox_SelectedValueChanged(object sender, EventArgs e
            )
        {
            if (this.stockItemsListBox.SelectedItem != null)
            {
                this.tabControl1.SelectTab(STOCKITEMTAB);
                this.SwitchStockItemControls(true);
            }
            else
            {
                this.SwitchStockItemControls(false);
            }
        }

        private void addStockItemToolStripMenuItem_Click(object sender, EventArgs e)
        {
            this._Presenter.CreateNewStockItem();
        }

        private void deleteStockItemToolStripMenuItem_Click(object sender, EventArgs e
            )
        {
            _Presenter.DeleteStockItem();
        }

        private void addBankAccountToolStripMenuItem_Click(object sender, EventArgs e)
        {
            this._Presenter.CreateNewBankAccount();
        }

        private void deleteBankAccountToolStripMenuItem_Click(object sender, EventArgs
             e)
        {
            this._Presenter.DeleteBankAccount();
        }

        private void openStockItemToolStripMenuItem_Click(object sender, EventArgs e)
        {

            if (this.openFileDialog.ShowDialog() == DialogResult.OK)
            {
                OpenFileDialog file = this.openFileDialog;
                this._Presenter.LoadStockItemsFromFile(file.FileName);
            }
        }

        private void openBankAccountsToolStripMenuItem_Click(object sender, EventArgs
            e)
        {
            if (this.openFileDialog.ShowDialog() == DialogResult.OK)
            {
                OpenFileDialog file = this.openFileDialog;
                this._Presenter.LoadBankAccountsFromFile(file.FileName);
            }
        }

        private void saveStockItemsToolStripMenuItem_Click(object sender, EventArgs e)
        {
            if (this.saveFileDialog.ShowDialog() == DialogResult.OK)
            {
                SaveFileDialog file = this.saveFileDialog;
                this._Presenter.SaveStockItemsToFile(file.FileName);
            }
        }

        private void saveBankAccountsToolStripMenuItem_Click(object sender, EventArgs
            e)
        {
            if (this.saveFileDialog.ShowDialog() == DialogResult.OK)
            {
                SaveFileDialog file = this.saveFileDialog;
                this._Presenter.SaveBankAccountsToFile(file.FileName);
            }
        }

        private void applyButton_Click(object sender, EventArgs e)
        {
            this.ResetColoring();
            this._Presenter.EditStockItem();
        }

        private void ResetColoring()
        {
            foreach (Control control in this.backgroundColorChanged)
            {
                control.BackColor = Color.White;
            }
        }

        private void applyBankAccountButton_Click(object sender, EventArgs e)
        {
            this.ResetColoring();
            this._Presenter.EditBankAccount();
        }

        void PlaceOrderButton_Click(object sender, EventArgs e)
        {
            this._Presenter.OrderItem();
        }

        private void depositButton_Click(object sender, EventArgs e)
        {
            this._Presenter.Deposit();
        }

        private void withdrawButton_Click(object sender, EventArgs e)
        {
            this._Presenter.Withdraw();
        }

        private void exitToolStripMenuItem_Click(object sender, EventArgs e)
        {
            this._Presenter.CloseApplication();
        }

        private void settingsToolStripMenuItem_Click(object sender, EventArgs e)
        {
            SettingsWindow sw = new SettingsWindow();
            sw.ShowDialog();
            this.LoadFilePathSettings();
        }

        private void saveStripButton_Click(object sender, EventArgs e)
        {
            this._Presenter.SaveBankAccountsToFile();
            this._Presenter.SaveStockItemsToFile();
        }

        #endregion

        #region IStockItemView

        /// <summary>
        /// Gets the current stock.
        /// </summary>
        /// <value>The current stock.</value>
        public int CurrentStock
        {
            get
            {
                int currentStock;
                try
                {
                    currentStock = int.Parse(currStockTextBox.Text);
                    return currentStock;
                }
                catch (FormatException e)
                {
                    this.DisplayError(currStockTextBox);
                    throw;
                }
            }
        }

        /// <summary>
        /// Gets the required stock.
        /// </summary>
        /// <value>The required stock.</value>
        public int RequiredStock
        {
            get
            {
                int requiredStock;
                try
                {
                    requiredStock = int.Parse(reqStockTextBox.Text);
                    return requiredStock;
                }
                catch (FormatException e)
                {
                    this.DisplayError(reqStockTextBox);
                    throw;
                }
            }
        }

        /// <summary>
        /// Gets the stock code.
        /// </summary>
        /// <value>The stock code.</value>
        public string StockCode
        {
            get { return stockCodeTextBox.Text; }
        }

        /// <summary>
        /// Gets the name of the supplier.
        /// </summary>
        /// <value>The name of the supplier.</value>
        public string SupplierName
        {
            get { return supplierNameTextBox.Text; }
        }

        /// <summary>
        /// Gets the unit cost.
        /// </summary>
        /// <value>The unit cost.</value>
        public double UnitCost
        {
            get
            {
                double unitCost;
                try
                {
                    unitCost = double.Parse(priceTextBox.Text);
                    return unitCost;
                }
                catch (FormatException e)
                {
                    this.DisplayError(priceTextBox);
                    throw;
                }
            }
        }

        /// <summary>
        /// Gets the name of the item.
        /// </summary>
        /// <value>The name of the item.</value>
        public string ItemName
        {
            get { return itemNameTextBox.Text; }
        }

        #endregion

        #region IBankAccountView

        /// <summary>
        /// Gets the account number.
        /// </summary>
        /// <value>The account number.</value>
        public int AccountNumber
        {
            get
            {
                int accountNumber;
                try
                {
                    accountNumber = int.Parse(accountNumberTextBox.Text);
                    return accountNumber;
                }
                catch (FormatException e)
                {
                    this.DisplayError(accountNumberTextBox);
                    throw;
                }
            }
        }

        /// <summary>
        /// Gets the surname.
        /// </summary>
        /// <value>The surname.</value>
        public string Surname
        {
            get { return nameTextBox.Text; }
        }

        /// <summary>
        /// Gets the balance.
        /// </summary>
        /// <value>The balance.</value>
        public double Balance
        {
            get
            {
                double balance;
                try
                {
                    balance = double.Parse(balanceTextBox.Text);
                    return balance;
                }
                catch (FormatException e)
                {
                    this.DisplayError(balanceTextBox);
                    throw;
                }
            }
        }

        #endregion

        #region ICongregateView

        /// <summary>
        /// Gets the stock item.
        /// </summary>
        /// <value>The stock item.</value>
        public StockItem StockItem
        {
            get { return (StockItem)this.stockItemsListBox.SelectedItem; }
        }

        /// <summary>
        /// Gets the bank account.
        /// </summary>
        /// <value>The bank account.</value>
        public BankAccount BankAccount
        {
            get { return (BankAccount)this.bankAccountsListBox.SelectedItem; }
        }

        /// <summary>
        /// Gets the quantity value.
        /// </summary>
        /// <value>The quantity.</value>
        public int Quantity
        {
            get
            {
                int quan = 0;
                try
                {
                    quan = int.Parse(quantityTextBox.Text);
                    return quan;
                }
                catch (FormatException e)
                {
                    this.DisplayError(quantityTextBox);
                    throw;
                }
            }
        }

        /// <summary>
        /// Gets the deposit value.
        /// </summary>
        /// <value>The deposit.</value>
        public double Deposit
        {
            get
            {
                double deposit;
                try
                {
                    deposit = double.Parse(depositQuantityTextBox.Text);
                    return deposit;
                }
                catch (FormatException e)
                {
                    this.DisplayError(depositQuantityTextBox);
                    throw;
                }
            }
        }

        /// <summary>
        /// Gets the withdraw value.
        /// </summary>
        /// <value>The withdraw.</value>
        public double Withdraw
        {
            get
            {
                double withdraw;
                try
                {
                    withdraw = double.Parse(withdrawQuantityTextBox.Text);
                    return withdraw;
                }
                catch (FormatException e)
                {
                    this.DisplayError(withdrawQuantityTextBox);
                    throw;
                }
            }
        }

        /// <summary>
        /// Displays the validation errors.
        /// </summary>
        /// <param name="errorCollection">The error collection.</param>
        public void DisplayValidationErrors(ErrorMessageCollection errorCollection)
        {
            MessageBox.Show(errorCollection.ToString(), "Errors occured", MessageBoxButtons.OK, MessageBoxIcon.Error);
        }

        /// <summary>
        /// Asks fir confirmation of a deletion.
        /// </summary>
        /// <returns></returns>
        public bool ConfirmDelete()
        {
            DialogResult result = MessageBox.Show("Are you sure you want to delete this item?", "Confirm delete", MessageBoxButtons.YesNo, MessageBoxIcon.Question);
            return result == DialogResult.Yes;
        }

        /// <summary>
        /// Asks fir confirmation of closing the application.
        /// </summary>
        /// <returns></returns>
        public bool ConfirmClose()
        {
            DialogResult result = MessageBox.Show("Are you sure you want to close the
                application?", "Confirm close", MessageBoxButtons.YesNo,
                MessageBoxIcon.Question);
            return result == DialogResult.Yes;
        }

        #endregion

        #region Private methods

        /// <summary>
        /// Switches the BankAccount Controls depending on the selection.
        /// </summary>
        /// <param name="enabled">True if controls shall be enabled, false otherwise
            .</param>
        private void SwitchBankAccountControls(bool enabled)
        {
            this.deleteBankAccountToolStripMenuItem.Enabled = enabled;
            this.deleteBankAccountToolStripButton.Enabled = enabled;
            this.accountNumberTextBox.Enabled = enabled;
            this.nameTextBox.Enabled = enabled;
            this.depositQuantityTextBox.Enabled = enabled;
            this.withdrawQuantityTextBox.Enabled = enabled;
        }

        /// <summary>
        /// Switches the StockItem Controls depending on the selection.
        /// </summary>
        /// <param name="enabled">True if controls shall be enabled, false otherwise
            .</param>
        private void SwitchStockItemControls(bool enabled)
        {
            this.deleteStockItemToolStripButton.Enabled = enabled;
            this.deleteStockItemToolStripMenuItem.Enabled = enabled;
            this.stockCodeTextBox.Enabled = enabled;
            this.itemNameTextBox.Enabled = enabled;
            this.supplierNameTextBox.Enabled = enabled;
            this.reqStockTextBox.Enabled = enabled;
            this.currStockTextBox.Enabled = enabled;
            this.priceTextBox.Enabled = enabled;
        }

        private void DisplayError(Control form)
        {
            backgroundColorChanged.Add(form);
            form.BackColor = Color.MistyRose;
        }

        private void LoadFilePathSettings()
        {
            Settings settings = Settings.Default;

            String filePathStockItems = settings.StockItemFilePath;
            String filePathBankAccounts = settings.BankAccountFilePath;
            if (!String.IsNullOrEmpty(filePathStockItems))
            {
                this._Presenter.SetUpStockItemFilePath(filePathStockItems);
            }
            if (!String.IsNullOrEmpty(filePathBankAccounts))
            {
                this._Presenter.SetUpBankAccountsFilePath(filePathBankAccounts);
            }
        }

        private void SetUpDataBindings()
        {
            stockItemsListBox.DataSource = _Model.StockItems;
            stockItemsListBox.DisplayMember = "Name";

            /*
             * The datasourceupdatemode is set to "Never".
             * This leads to the ability to enforce the use of the presenter to update the values in the model.
             * This way the validation errors can be handled by the presenter thus leading to better seperation of concerns.
             */
            stockCodeTextBox.DataBindings.Add("Text", _Model.StockItems, "StockCode", false, DataSourceUpdateMode.Never);
            itemNameTextBox.DataBindings.Add("Text", _Model.StockItems, "Name", false, DataSourceUpdateMode.Never);
            supplierNameTextBox.DataBindings.Add("Text", _Model.StockItems, "SupplierName", false, DataSourceUpdateMode.Never);
            currStockTextBox.DataBindings.Add("Text", _Model.StockItems, "CurrentStock", false, DataSourceUpdateMode.Never);
            reqStockTextBox.DataBindings.Add("Text", _Model.StockItems, "RequiredStock", false, DataSourceUpdateMode.Never);
            priceTextBox.DataBindings.Add("Text", _Model.StockItems, "UnitCost", false, DataSourceUpdateMode.Never);

            bankAccountsListBox.DataSource = _Model.BankAccounts;
            bankAccountsListBox.DisplayMember = "AccountNumber";

            accountNumberTextBox.DataBindings.Add("Text", _Model.BankAccounts, "AccountNumber", false, DataSourceUpdateMode.Never);
            nameTextBox.DataBindings.Add("Text", _Model.BankAccounts, "Surname", false
                , DataSourceUpdateMode.Never);
            balanceTextBox.DataBindings.Add("Text", _Model.BankAccounts, "Balance",
                false, DataSourceUpdateMode.Never);
        }

        private void quantityTextBox_TextChanged(object sender, EventArgs e)
        {
            String newText = quantityTextBox.Text;
            int parseInt = 0;
            if (int.TryParse(newText, out parseInt))
            {
                placeOrderButton.Enabled = true;
            }
            else
            {
                placeOrderButton.Enabled = false;
            }
        }

        #endregion
    }
}
```

Listing 6: MainWindow.cs

```
ï»¿using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.Collections.Specialized;
using System.Configuration;
using Assessment_One.Properties;

namespace Assessment_One
{
    public partial class SettingsWindow : Form
    {
        public SettingsWindow()
        {
            InitializeComponent();
            SetUpTextBoxes();
        }

        private void SetUpTextBoxes()
        {
            Settings settings = Settings.Default;
            stockItemsFilePathTextBox.Text = settings.StockItemFilePath;
            bankAccountFilePathTextBox.Text = settings.BankAccountFilePath;
        }

        private void chooseStockItemsFilePath_Click(object sender, EventArgs e)
        {
            this.openFileDialog.ShowDialog();
            OpenFileDialog file = this.openFileDialog;
            this.stockItemsFilePathTextBox.Text = file.FileName;
        }

        private void chooseBankAccountsFilePath_Click(object sender, EventArgs e)
        {
            this.openFileDialog.ShowDialog();
            OpenFileDialog file = this.openFileDialog;
            this.bankAccountFilePathTextBox.Text = file.FileName;
        }

        private void cancelButton_Click(object sender, EventArgs e)
        {
            this.Dispose();
        }

        private void applyButton_Click(object sender, EventArgs e)
        {
            String stockItemsFilePath = stockItemsFilePathTextBox.Text;
            String bankAccountsFilePath = bankAccountFilePathTextBox.Text;
            this.SaveApplicationSettings(stockItemsFilePath, bankAccountsFilePath);
            this.Dispose();
        }

        private void SaveApplicationSettings(string stockItemsFilePath, string
             bankAccountsFilePath)
        {
            Settings settings = Settings.Default;
            settings.StockItemFilePath = stockItemsFilePath;
            settings.BankAccountFilePath = bankAccountsFilePath;
            settings.Save();
        }
    }
}
```

Listing 7: Settings.cs

A.2 APPLICATION LOGIC

A.2.1 *Interfaces Package*

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

namespace ApplicationLogic.Interfaces
{
    /// <summary>
    /// Utilized by the presenter to get the necessary values from a view.
    /// </summary>
    public interface IBankAccountView
    {
        int AccountNumber { get; }
        String Surname { get; }
        double Balance { get; }
    }
}
```

Listing 8: IBankAccountView.cs

```
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 9: ICongegrateView.cs

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

namespace ApplicationLogic.Interfaces
{
    /// <summary>
    /// Used to ensure all objects will be able to persited via FileHandler class.
    /// </summary>
    /// <typeparam name="T"></typeparam>
    public interface ICSVSerializable<T>
    {
        String CsvRepresentation();

        T ParseFromString(String stringRepresentation);
    }
}
```

Listing 10: ICSVSerializable.cs

```
ï»¿using System;
namespace ApplicationLogic.Interfaces
{
    /// <summary>
    /// Utilized by the presenter to get the necessary values from a view.
    /// </summary>
    public interface IStockItemView
    {
        int CurrentStock { get; }
        string ItemName { get; }
        int RequiredStock { get; }
        string StockCode { get; }
        string SupplierName { get; }
        double UnitCost { get; }
    }
}
```

Listing 11: IStockItemView.cs

```
using System;
using System.ComponentModel;
using ApplicationLogic.Model;

namespace ApplicationLogic.Interfaces
{
    /// <summary>
    /// Utilized by the View to set up the data binding to the lists in the model
    /// </summary>
        public interface IViewModel
        {
                BindingList<StockItem> StockItems { get; }
                BindingList<BankAccount> BankAccounts { get; }
        }
}
```

Listing 12: IViewModel.cs

A.2.2 *Model Package*

```
using System;
using System.ComponentModel;
using System.Security.Cryptography;
using ApplicationLogic.Interfaces;
using System.Collections;
using System.Collections.Generic;

namespace ApplicationLogic.Model
{
    /// <summary>
    /// Handles persistence issues and ensure the correct sequence of method calls.
    /// </summary>
    public class AppDataManager : IViewModel
    {
        private BindingList<StockItem> _StockItems;
        /// <summary>
        /// Gets or sets the stock items.
        /// </summary>
        /// <value>The stock items.</value>
        public BindingList<StockItem> StockItems
        {
            get { return _StockItems; }
            set { _StockItems = value; }
        }

        private BindingList<BankAccount> _BankAccounts;
        /// <summary>
        /// Gets or sets the bank accounts.
        /// </summary>
        /// <value>The bank accounts.</value>
        public BindingList<BankAccount> BankAccounts
        {
            get { return _BankAccounts; }
            set { _BankAccounts = value; }
        }

        /// <summary>
        /// Gets or sets the stock item handler.
        /// </summary>
        /// <value>The stock item handler.</value>
        public FileHandler<StockItem> StockItemHandler { get; private set; }
        /// <summary>
        /// Gets or sets the bank account handler.
        /// </summary>
        /// <value>The bank account handler.</value>
        public FileHandler<BankAccount> BankAccountHandler { get; private set; }

        public AppDataManager()
        {
            this.StockItems = new BindingList<StockItem>();
            this.BankAccounts = new BindingList<BankAccount>();
            this.StockItemHandler = new FileHandler<StockItem>();
            this.BankAccountHandler = new FileHandler<BankAccount>();
        }

        /// <summary>
        /// Creates a new StockItem and initializes it with dummy values.
        /// Adds the item to the StockItem collection.
        /// </summary>
        public void CreateNewStockItem()
        {
            StockItem si = new StockItem("0000", "Dummy Item", "None", 0.0, 0, 0);
            this.StockItems.Add(si);
        }

        /// <summary>
        /// Deletes a StockItem from the StockItem collection.
        /// Throws an ArgumentException if the item can not be found.
        /// </summary>
        public void DeleteStockItem(StockItem si)
        {
            if (this.StockItems.Contains(si))
            {
                this.StockItems.Remove(si);
            }
            else
            {
                throw new ArgumentException("Item to delete not present.");
            }
        }

        /// <summary>
        /// Creates a new BankAccount and initializes it with dummy values.
        /// Adds the item to the BankAccount collection.
        /// </summary>
        public void CreateNewBankAccount()
        {
            BankAccount ba = new BankAccount(0, "Dummy Account", 0.0);
            this.BankAccounts.Add(ba);
        }

        /// <summary>
        /// Deletes a BankAccount from the BankAccount collection.
        /// Throws an ArgumentException if the account can not be found.
        /// </summary>
        public void DeleteBankAccount(BankAccount ba)
        {
            if (this.BankAccounts.Contains(ba))
            {

                this.BankAccounts.Remove(ba);
            }
            else
            {
                throw new ArgumentException("Item to delete not present.");
            }
        }

        /// <summary>
        /// Attempts to edit specified StockItem in the StockItem collection.
        /// Throws an ArgumentException if the item can not be found.
        /// </summary>
        /// <param name="si">StockItem to be edited.</param>
        /// <param name="stockCode">New StockCode</param>
        /// <param name="supplier">New SupplierName</param>
        /// <param name="name">New Name</param>
        /// <param name="currentStock">New CurrentStock</param>
        /// <param name="reqStock">New RequiredStock</param>
        /// <param name="price">New Price</param>
        internal void EditStockItem(StockItem si, string stockCode, string supplier,
             string name, int currentStock, int reqStock, double price)
        {
            if (si != null)
            {
                si.EditStockItem(stockCode, name, supplier, price, reqStock,
                     currentStock);
            }
            else
            {
                throw new ArgumentNullException("Stock item to edit not present.");
            }
        }

        /// <summary>
        /// Attempts to edit specified BankAccount in the BankAccount collection.
        /// Throws an ArgumentException if the account can not be found.
        /// </summary>
        /// <param name="ba">BankAccount to be edited</param>
        /// <param name="surname">New surname</param>
        /// <param name="accountNumber">New accoutnumber</param>
        internal void EditBankAccount(BankAccount ba, string surname, int
            accountNumber)
        {
            if (ba != null)
            {
                ba.EditBankAccount(surname, accountNumber);
            }
            else
            {
                throw new ArgumentNullException("Bank account to edit not present.");
            }

        }

        /// <summary>
        /// Validates the changes that may be made to a stock item.
        /// </summary>
        /// <param name="accountNumber">New account number</param>
        /// <param name="surname">New surname</param>
        /// <returns>True if new values are vald. False otherwise.</returns>
        internal bool ValidateStockItem(string stockCode, string name, string supplier
            , double price, int reqStock, int currentStock)
        {
            bool areValuesValid = StockItem.Validate(stockCode, name, supplier, price,
                 reqStock, currentStock);
            return areValuesValid;
        }

        /// <summary>
        /// Stores the errors that occured in the last validation of stock item data.
        /// </summary>
        /// <returns>The errors that occured.</returns>
        internal ErrorMessageCollection StockItemErrors()
        {
            return StockItem.ErrorMessages; ;
        }


        /// <summary>
        /// Clears the errors of the last stock item validation.
        /// </summary>
        internal void ClearStockItemErrors()
        {
            StockItem.ErrorMessages.Clear();
        }

        /// <summary>
        /// Clears the errors of the last bank account validation.
        /// </summary>
        internal void ClearBankAccountErrors()
        {
            BankAccount.ErrorMessages.Clear();
        }

        /// <summary>
        /// Stores the errors that occured in the last validation of bank account data
            .
        /// </summary>
        /// <returns>The errors that occured.</returns>
        internal ErrorMessageCollection BankAccountErrors()
        {
            return BankAccount.ErrorMessages;
        }

        /// <summary>
        /// Validates the changes that may be made to a bank account.
        /// </summary>
        /// <param name="accountNumber">New account number</param>
        /// <param name="surname">New surname</param>
        /// <returns>True if new values are vald. False otherwise.</returns>
        internal bool ValidateBankAccount(int accountNumber, string surname)
        {
            bool areValidValues = BankAccount.Validate(accountNumber, surname);
            return areValidValues;
        }

        /// <summary>
        /// Attempts to order an item. If no quantity was provided the required
            quantity will be ordered.
        /// Otherwise the provided quantity will be ordered.
        /// </summary>
        /// <param name="indexStockItem">Index of the stock item in the stock items
            list.</param>
        /// <param name="indexBankAccount">Index of the bank account in the bank
            account list.</param>
        /// <param name="quantity">The quantity to be ordered. 0 orders the required
            quantity.</param>
        public void OrderItem(StockItem si, BankAccount ba, int quantity)
        {
            if (ba != null && si != null)
            {
                bool buyExcessStock = false;
                if (quantity == 0)
                {
                    quantity = si.RequiredStock;
                }
                else
                {
                    /*
```

```
                     * It is possible to order more than the needed quantity.
                     */
                    buyExcessStock = true;
                }
                double priceOfOrder = quantity * si.UnitCost;
                if (priceOfOrder <= ba.Balance)
                {
                    ba.Transfer(1, priceOfOrder);
                    si.CurrentStock += quantity;
                    /*
                     * Allow the user to buy more than needed.
                     */
                    if (buyExcessStock)
                    {
                        if (quantity > si.RequiredStock)
                        {
                            si.RequiredStock = 0;
                        }
                    }
                }
                else
                {
                    throw new NotEnoughFundsException(String.Format("Not enough founds on bank account {0} to place an order for {1} £", ba.AccountNumber, priceOfOrder));
                }
            }
            else
            {
                throw new ArgumentNullException("Stock item or bank account provided do not exist.");
            }
        }

        /// <summary>
        /// Attempts to deposit the requested amount from the specified bank account.
        /// </summary>
        /// <param name="indexBankAccount"></param>
        /// <param name="amount"></param>
        internal void Deposit(BankAccount ba, double amount)
        {
            if (ba != null)
            {
                ba.Deposit(amount);
            }
            else
            {
                throw new ArgumentNullException("Provided bank account does not exist.");
            }
        }

        /// <summary>
        /// Attempts to withdraw the specified amount from the specified bank account.
        /// </summary>
        /// <param name="indexBankAccount"></param>
        /// <param name="amount"></param>
        internal void Withdraw(BankAccount ba, double amount)
        {
            if (ba != null)
            {
                ba.Withdraw(amount);
            }
            else
            {
                throw new ArgumentNullException("Bank account provided does not exist.
                    ");
            }
        }

        /// <summary>
        /// Will reload the StockItem collection from the specified file.
        /// Will overwrite the currently existing StockItem collection.
        /// </summary>
        /// <param name="filePath"></param>
        internal void LoadStockItemsFromFile(string filePath)
        {
            this.StockItemHandler.ReadFilePath = filePath;
            IList<StockItem> stockItems = StockItemHandler.LoadFromFile(new StockItem
                ());
            this.StockItems.Clear();
            foreach (StockItem item in stockItems)
            {
                this.StockItems.Add(item);
            }
        }

        /// <summary>
        /// Will reload the BankAccount collection from the specified file.
        /// Will overwrite the currently existing BankAccount collection.
        /// </summary>
        /// <param name="filePath">The path to the file</param>
        internal void LoadBankAccountsFromFile(string filePath)
        {
            this.BankAccountHandler.ReadFilePath = filePath;
            IList<BankAccount> bankAccounts = BankAccountHandler.LoadFromFile(new
                BankAccount());
            this.BankAccounts.Clear();
            foreach (BankAccount item in bankAccounts)
            {
                this.BankAccounts.Add(item);
            }
        }

        /// <summary>
        /// Will save the current StockItem collection to the specified file.
        /// </summary>
        /// <param name="filePath">The path to the file</param>
        internal void SaveStockItemsToFile(string filePath)
        {
            this.StockItemHandler.WriteFilePath = filePath;
            this.StockItemHandler.SaveToFile(this.StockItems);
        }

        /// <summary>
        /// Will save the current BankAccount collection to the specified file.
        /// </summary>
        /// <param name="filePath">The path to the file</param>
        internal void SaveBankAccountsToFile(string filePath)
        {
            this.BankAccountHandler.WriteFilePath = filePath;
            this.BankAccountHandler.SaveToFile(this.BankAccounts);
        }

        /// <summary>
        /// Will save the current BankAccount collection to the file stored in the
            FileHandler.
        /// Throws NoFilePathSetException if no path is set.
        /// </summary>
        internal void SaveBankAccountsToFile()
        {
            this.BankAccountHandler.SaveToFile(this.BankAccounts);
        }

        /// <summary>
        /// Will save the current BankAccount collection to the file stored in the
            FileHandler.
        /// Throws NoFilePathSetException if no path is set.
        /// </summary>
        internal void SaveStockItemsToFile()
        {
            this.StockItemHandler.SaveToFile(this.StockItems);
        }
    }
}
```

Listing 13: AppDataManager.cs

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

namespace ApplicationLogic.Model
{
        /// <summary>
        /// Pseudo bank account to allow the placement of an order.
        /// </summary>
        public class BankAccount : INotifyPropertyChanged, ICSVSerializable<
            BankAccount>
        {
                private int _AccountNumber;
                /// <summary>
                /// Gets or sets the account number.
                /// </summary>
                /// <value>The account number.</value>
                public virtual int AccountNumber {
                        get { return _AccountNumber; }
                        set {
                                _AccountNumber = value;
                                this.NotifyPropertyChanged ("AccountNumber");
                        }
                }

                private String _Surname;
                /// <summary>
                /// Gets or sets the surname.
                /// </summary>
                /// <value>The surname.</value>
                public virtual String Surname {
                        get { return _Surname; }
                        set {
                                _Surname = value;
                                this.NotifyPropertyChanged ("Surname");
                        }
                }

                private double _Balance;
                /// <summary>
                /// Gets or sets the balance.
                /// </summary>
                /// <value>The balance.</value>
                public virtual double Balance {
                        get { return _Balance; }
                        private set {
                                if (value < 0.0) {
                                        throw new ArgumentException ("This class does
                                            not allow a balance smaller than 0.");
                                } else {
                                        _Balance = value;
                                        this.NotifyPropertyChanged ("Balance");
                                }

                        }
                }

                public static ErrorMessageCollection ErrorMessages = new
                    ErrorMessageCollection ();

                /// <summary>
                /// Initializes a new instance of the <see cref="BankAccount"/> class.
                /// </summary>
                public BankAccount ()
                {
                }

                /// <summary>
                /// Initializes a new instance of the <see cref="BankAccount"/> class.
                /// </summary>
                /// <param name="acc">The acc.</param>
                /// <param name="name">The name.</param>
                /// <param name="balance">The balance.</param>
                public BankAccount (int acc, string name, double balance)
                {
                        this.AccountNumber = acc;
                        this.Surname = name;
                        this.Balance = balance;
                }

                /// <summary>
                /// Allows the withdrawal of money from this account.
                /// Credit is not granted.
                /// </summary>
                /// <param name="amount">Amount to be withdrawn - must be greater than
                     0.</param>
                public void Withdraw (double amount)
                {
                        if (amount > 0.0) {
                                if (Balance >= amount) {
                                        this.Balance -= amount;
                                } else {
                                        throw new NotEnoughFundsException ("Not enough
                                             founds on bank account to withdraw.");
                                }
                        } else {
                throw new ArgumentException("To deposit money please use the
                    appropriate function.");
                        }
                }

                /// <summary>
                /// Allows the deposit of money to this account.
                /// </summary>
```

```
		/// <param name="amount">Amount to be deposited - must be greater than 0.</param>
		public void Deposit (double amount)
		{
			if (amount <= 0.0) {
				throw new ArgumentException ("To withdraw money please use the appropriate function.");
			} else {
				this.Balance += amount;
			}
		}

		/// <summary>
		/// Amount to be transfered to another account.
		/// NOTE: This method is a fake to simulate "real" banking. The money will not be transfered to any account.
		/// </summary>
		/// <param name="amount">Amount to be transfered - must be greater than 0.</param>
		/// <param name="accountNumber">Account number to transfer the money to.</param>
		public void Transfer (int accountNumber, double amount)
		{
			if (amount >= 0.0) {
				if (this.Balance >= amount) {
					this.Balance -= amount;
					// TODO: In reality: fancy logic to transfer money.
				} else {
					throw new NotEnoughFundsException ("There are not enough funds present to fulfill the required action.");
				}
			} else {
				throw new ArgumentException ("It is not possible to transfer funds from another account to yours.");
			}
		}

		/// <summary>
		/// Validates a set of possible changes to a BankAccount.
		/// </summary>
		/// <param name="accountNumber">AccountNumber to be verified</param>
		/// <param name="surname">Surename to be verified</param>
		/// <returns>True if the values would be valid, false otherwise.</returns>
		static internal bool Validate (int accountNumber, String surname)
		{
			if (String.IsNullOrEmpty (surname)) {
				ErrorMessages.Add (new ErrorMessage ("Need the name of the account owner."));
```

```
        }
        if (accountNumber <= 0) {
                ErrorMessages.Add (new ErrorMessage ("Need a valid
                    account number: greater 0."));
        }
        return ErrorMessages.Count == 0;
}

public event PropertyChangedEventHandler PropertyChanged;

private void NotifyPropertyChanged (String info)
{
        if (PropertyChanged != null) {
                PropertyChanged (this, new PropertyChangedEventArgs (
                    info));
        }
}

internal void EditBankAccount (string surname, int accountNumber)
{
        this.Surname = surname;
        this.AccountNumber = accountNumber;
}

/// <summary>
/// Returns the current BankAccount object as a CSV-String.
/// </summary>
/// <returns>Representation of the current object as CSV-String.</
    returns>
public string CsvRepresentation ()
{
        return String.Format ("{0},{1},{2}", this.AccountNumber, this.
            Surname, this.Balance);
}

/// <summary>
/// Attempts to create a BankAccount object from a string.
/// </summary>
/// <param name="stringRepresentation">The String to be parsed to bank
     account.</param>
/// <returns>BankAccount object.</returns>
public BankAccount ParseFromString (string stringRepresentation)
{
        string[] split = stringRepresentation.Split (',');
        String accountNumber = split[0];
        String surname = split[1];
        String balance = split[2];
        int accNumber = 0;
        double bal = 0;
        if (!String.IsNullOrEmpty (accountNumber)) {
                accNumber = int.Parse (accountNumber);
```

```
                        }
                        if (!String.IsNullOrEmpty (balance)) {
                                bal = double.Parse (balance);
                        }
                        return new BankAccount (accNumber, surname, bal);
                }
        }
}
```

Listing 14: BankAccount.cs

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

namespace ApplicationLogic.Model
{
    public class ErrorMessage
    {
        public String Message { get; set; }
        public String Source { get; set; }

        public ErrorMessage(string p)
        {
            this.Message = p;
        }

        public override string ToString()
        {
            return Message;
        }
    }
}
```

Listing 15: ErrorMessage.cs

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

namespace ApplicationLogic.Model
{
    /// <summary>
    /// Used to store meaningful errormessages for further use.
    /// </summary>
    public class ErrorMessageCollection : List<ErrorMessage>
    {
        public override string ToString()
        {
            StringBuilder sb = new StringBuilder();

            foreach (ErrorMessage item in this)
            {
                if (sb.Length > 0)
                {
                    sb.Append(Environment.NewLine);
                }

                sb.Append(item.ToString());
            }

            return sb.ToString();
        }
    }
}
```

Listing 16: ErrorMessageCollection.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using ApplicationLogic.Interfaces;
using System.IO;
using ApplicationLogic.Model;

namespace ApplicationLogic
{
    /// <summary>
    /// Handles reading and writing from files.
    /// </summary>
    public class FileHandler<T> where T : ICSVSerializable<T>
    {

        public String ReadFilePath;
        public String WriteFilePath;

        public FileHandler()
        {
        }

        public FileHandler(String readFilePath, String writeFilePath)
        {
            this.ReadFilePath = readFilePath;
            this.WriteFilePath = writeFilePath;
        }

        /// <summary>
        /// Attempts to write the specified collection to the file specified in the
             Handler.
        /// Throws NoFilePathSetException if no file has yet been set.
        /// </summary>
        /// <param name="elements">Collection to be saved to file.</param>
        public void SaveToFile(IList<T> elements)
        {
            if (!String.IsNullOrEmpty(this.WriteFilePath))
            {
                FileStream writeFile = File.Open(WriteFilePath, FileMode.Create);
                using (StreamWriter sw = new StreamWriter(writeFile))
                {
                    foreach (T item in elements)
                    {
                        sw.Write(item.CsvRepresentation());
                        sw.Write("\n");
                    }
                }
            }
            else
            {
                throw new NoFilePathSetException("No file path to write to set.");
            }

        }

        /// <summary>
        /// Attempts to read a collection of items from the specified file.
        /// Throws NoFilePathSetException if no file has yet been set.
        /// </summary>
        /// <param name="item">Parameter needed to construct the objects.</param>
        /// <returns>Collection of items.</returns>
        public IList<T> LoadFromFile(T item)
        {
            if (!String.IsNullOrEmpty(this.ReadFilePath))
            {
                FileStream readFile = File.Open(ReadFilePath, FileMode.Open);
                List<T> returnList = new List<T>();
                using (StreamReader sr = new StreamReader(readFile))
                {
                    String readString = "";
                    while ((readString = sr.ReadLine()) != null)
                    {
                                            try {
                                                    T t = item.ParseFromString(
                                                        readString);
                                returnList.Add(t);
                                            } catch (FormatException ex) {
                                                    Console.WriteLine(ex.
                                                        StackTrace);
                                            }
                    }
                }
                return returnList;
            }
            else
            {
                throw new NoFilePathSetException("No file path to write to set.");
            }
        }
    }
}
```

Listing 17: FileHandler.cs

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

namespace ApplicationLogic.Model
{
    public class NoFilePathSetException : Exception
    {

        public NoFilePathSetException(String msg)
            : base(msg)
        {
        }
    }
}
```

Listing 18: NoFilePathSetException.cs

```
ï»¿using System;

namespace ApplicationLogic.Model
{
        /// <summary>
        /// Description of NotEnoughFundsException.
        /// </summary>
        public class NotEnoughFundsException: Exception
        {
                public NotEnoughFundsException(String message): base(message)
                {
                }
        }
}
```

Listing 19: NotEnoughFundsException.cs

```
ï»¿using System;
```

```
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Text.RegularExpressions;
using System.ComponentModel;
using ApplicationLogic.Interfaces;

namespace ApplicationLogic.Model
{
        /// <summary>
        /// Stores all necessary data for a StockItem.
        /// </summary>
        public class StockItem : INotifyPropertyChanged, ICSVSerializable<StockItem>
        {
        private const string REGEX = "^[0−9]{4}$";
                private String _StockCode;
                public virtual String StockCode {
                        get { return _StockCode; }
                        set {
                                if (!IsValidStockCode (value)) {
                                        throw new ArgumentException ("Provided
                                            stockcode did not match designated format.
                                            ");
                                } else {
                                        this._StockCode = value;
                                        this.NotifyPropertyChanged ("StockCode");
                                }
                        }
                }

                private String _Name;
                public virtual String Name {
                        get { return _Name; }
                        set {
                                _Name = value;
                                this.NotifyPropertyChanged ("Name");
                        }
                }

                private String _SupplierName;
                public virtual String SupplierName {
                        get { return _SupplierName; }
                        set {
                                _SupplierName = value;
                                this.NotifyPropertyChanged ("SupplierName");
                        }
                }

                private double _UnitCost;
                public virtual double UnitCost {
                        get { return _UnitCost; }
```

```
            private set {
                if (value < 0.0) {
                    throw new ArgumentException ("Price can not be lower than 0.");
                } else {
                    _UnitCost = value;
                    this.NotifyPropertyChanged ("UnitCost");
                }

            }
        }

        private int _RequiredStock;
        public virtual int RequiredStock {
            get { return _RequiredStock; }
            set {
                if (value < 0) {
                    throw new ArgumentException ("Can not require less than 0 items.");
                } else {
                    _RequiredStock = value;
                    this.NotifyPropertyChanged ("RequiredStock");
                }

            }
        }

        private int _CurrentStock;
        public virtual int CurrentStock {
            get { return _CurrentStock; }
            set {
                if (value < 0) {
                    throw new ArgumentException ("Current stock can not be less than 0 items.");
                } else {
                    _CurrentStock = value;
                    this.NotifyPropertyChanged ("CurrentStock");
                }

            }
        }

        public static ErrorMessageCollection ErrorMessages = new ErrorMessageCollection ();

        public StockItem ()
        {
        }

        public StockItem (String stockCode, String name, String supplierName, double unitCost, int required, int currentStock)
        {
                this.StockCode = stockCode;
                this.Name = name;
                this.SupplierName = supplierName;
                this.UnitCost = unitCost;
                this.RequiredStock = required;
                this.CurrentStock = currentStock;
        }

        /// <summary>
        /// Checks if the stock code conforms to a certain format.
        /// Current format is exactly four numbers, with leading 0 allowed.
        /// </summary>
        /// <param name="value">The string that must be checked against the schema.</param>
        /// <returns>True if the string conforms to the schema, false otherwise.</returns>
        public static bool IsValidStockCode (string value)
        {
                if (String.IsNullOrEmpty (value)) {
                        throw new ArgumentNullException ("Provided stockcode was null or empty.");
                } else {
                        // TODO: Do not use magic numbers in code
                        Regex regexp = new Regex (REGEX);
                        return regexp.IsMatch (value);
                }

        }

        public void EditStockItem (String stockCode, String name, String supplierName, double unitCost, int required, int currentStock)
        {
                this.StockCode = stockCode;
                this.Name = name;
                this.SupplierName = supplierName;
                this.UnitCost = unitCost;
                this.RequiredStock = required;
                this.CurrentStock = currentStock;
        }

        /// <summary>
        /// Validates a set of possible changes to a StockItem.
        /// </summary>
        /// <param name="stockCode">StockCodew to be verified</param>
        /// <param name="name">Name to be verified</param>
        /// <param name="supplierName">SupplierName to be verified</param>
        /// <param name="unitCost">UnitCost to be verified</param>
        /// <param name="required">RequiredStock to be verified</param>
        /// <param name="currentStock">CurrentStock to be verified</param>
        /// <returns>True if the values would be valid, false otherwise.</returns>
        public static bool Validate (String stockCode, String name, String supplierName, double unitCost, int required, int currentStock)
        {
            if (String.IsNullOrEmpty (stockCode) || !IsValidStockCode (stockCode)) {
                ErrorMessages.Add (new ErrorMessage ("Need a stockcode that adheres to the stockcode format: 4 numbers."));
            }
            if (String.IsNullOrEmpty (name)) {
                ErrorMessages.Add (new ErrorMessage ("Need an item name."));
            }
            if (String.IsNullOrEmpty (supplierName)) {
                ErrorMessages.Add (new ErrorMessage ("Need a supplier name."));
            }
            if (unitCost < 0.0) {
                ErrorMessages.Add (new ErrorMessage ("Unit costs must be greater or equal 0."));
            }
            if (required < 0) {
                ErrorMessages.Add (new ErrorMessage ("Required stock must be greater or equal 0."));
            }
            if (currentStock < 0) {
                ErrorMessages.Add (new ErrorMessage ("Current stock must be greater or equal 0."));
            }
            return ErrorMessages.Count == 0;
        }

        public event PropertyChangedEventHandler PropertyChanged;

        private void NotifyPropertyChanged (String info)
        {
            if (PropertyChanged != null) {
                PropertyChanged (this, new PropertyChangedEventArgs (info));
            }
        }

        /// <summary>
        /// Returns the current StockItem object as a CSV-String.
        /// </summary>
        /// <returns>Representation of the current object as CSV-String.</returns>
        public String CsvRepresentation ()
        {
```

```
                        return String.Format ("{0},{1},{2},{3},{4},{5}", this.
                            StockCode, this.Name, this.SupplierName, this.UnitCost,
                            this.RequiredStock, this.CurrentStock);
                }

                /// <summary>
                /// Attempts to create a StockItem object from a string.
                /// </summary>
                /// <param name="stringRepresentation">The String to be parsed to
                    StockItem.</param>
                /// <returns>StockItem object.</returns>
                public StockItem ParseFromString (string stringRepresentation)
                {
                        string[] split = stringRepresentation.Split (',');
                        String stockCode = split[0];
                        String name = split[1];
                        String supplierName = split[2];
                        String unitCost = split[3];
                        String requiredStock = split[4];
                        String currentStock = split[5];
                        double cost = 0;
                        int reqStock = 0;
                        int currStock = 0;
                        if (!String.IsNullOrEmpty (unitCost)) {
                                cost = double.Parse (unitCost);
                        }
                        if (!String.IsNullOrEmpty (requiredStock)) {
                                reqStock = int.Parse (requiredStock);
                        }
                        if (!String.IsNullOrEmpty (currentStock)) {
                                currStock = int.Parse (currentStock);
                        }
                        return new StockItem (stockCode, name, supplierName, cost,
                            reqStock, currStock);
                }
        }
}
```

Listing 20: StockItem.cs

A.2.3 *Presenter Package*

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

```

namespace ApplicationLogic.Presenter
{
    /// <summary>
    /// Presenter for the MainWindow: handles events and GUI-related part of the
        application logic.
    /// </summary>
    public class CongregatePresenter
    {

        public ICongregateView _View;
        public IStockItemView _StockItemView;
        public IBankAccountView _BankAccountView;
        public AppDataManager _Model;

        /// <summary>
        /// Initializes a new instance of the <see cref="CongregatePresenter"/> class.
        /// </summary>
        /// <param name="view">The view.</param>
        /// <param name="stockItemView">The stock item view.</param>
        /// <param name="bankAccountView">The bank account view.</param>
        /// <param name="model">The model.</param>
        public CongregatePresenter(ICongregateView view, IStockItemView stockItemView,
             IBankAccountView bankAccountView, IViewModel model)
        {
            this._View = view;
            this._StockItemView = stockItemView;
            this._BankAccountView = bankAccountView;
            this._Model = model as AppDataManager;
        }

        /// <summary>
        ///     <see cref="ApplicationLogic.Model.AppDataManagerClass"/>
        /// </summary>
        public void CreateNewStockItem()
        {
            this._Model.CreateNewStockItem();
        }

        /// <summary>
        /// Deletes the stock item.
        /// </summary>
        public void DeleteStockItem()
        {
            if (this._View.ConfirmDelete())
            {
                StockItem si = this._View.StockItem;
                this._Model.DeleteStockItem(si);
            }
        }

        /// <summary>
        /// Creates the new bank account.
        /// </summary>
        public void CreateNewBankAccount()
        {
            this._Model.CreateNewBankAccount();
        }

        /// <summary>
        /// Deletes the bank account.
        /// </summary>
        public void DeleteBankAccount()
        {
            if (this._View.ConfirmDelete())
            {
                BankAccount ba = this._View.BankAccount;
                this._Model.DeleteBankAccount(ba);
            }
        }

        /// <summary>
        /// Edits the stock item.
        /// </summary>
        public void EditStockItem()
        {
            try
            {
                StockItem si = this._View.StockItem;
                String stockCode = this._StockItemView.StockCode;
                String supplier = this._StockItemView.SupplierName;
                String name = this._StockItemView.ItemName;
                int currentStock = this._StockItemView.CurrentStock;
                int reqStock = this._StockItemView.RequiredStock;
                double price = this._StockItemView.UnitCost;
                bool areValuesValid = this._Model.ValidateStockItem(stockCode, name,
                    supplier, price, reqStock, currentStock);
                if (areValuesValid)
                {
                    this._Model.EditStockItem(si, stockCode, supplier, name,
                        currentStock, reqStock, price);
                }
                else
                {
                    this._View.DisplayValidationErrors(this._Model.StockItemErrors());
                    this._Model.ClearStockItemErrors();
                }
            }
            catch (FormatException e)
            {
                DisplayError(e);
            }

        }

        /// <summary>
        /// Edits the bank account.
        /// </summary>
        public void EditBankAccount()
        {
            try
            {
                BankAccount ba = this._View.BankAccount;
                String surname = this._BankAccountView.Surname;
                int accountNumber = this._BankAccountView.AccountNumber;
                bool areValuesValid = this._Model.ValidateBankAccount(accountNumber,
                    surname);
                if (areValuesValid)
                {
                    this._Model.EditBankAccount(ba, surname, accountNumber);
                }
                else
                {
                    this._View.DisplayValidationErrors(this._Model.BankAccountErrors()
                        );
                    this._Model.ClearBankAccountErrors();
                }
            }
            catch (FormatException e)
            {
                DisplayError(e);
            }

        }

        /// <summary>
        /// Orders the item.
        /// </summary>
        public void OrderItem()
        {
            try
            {
                BankAccount ba = this._View.BankAccount;
                StockItem si = this._View.StockItem;
                this.EditStockItem();
                int quantity = this._View.Quantity;
                this._Model.OrderItem(si, ba, quantity);
            }
            catch (FormatException e)
            {
                DisplayError(e);
            }
            catch (NotEnoughFundsException e)
            {
                DisplayError(e);
            }
        }

        /// <summary>
        /// Deposits this instance.
        /// </summary>
        public void Deposit()
        {
            try
            {
                BankAccount ba = this._View.BankAccount;
                double amount = this._View.Deposit;
                this._Model.Deposit(ba, amount);
            }
            catch (ArgumentNullException e)
            {
                DisplayError(e);
            }
            catch (ArgumentException e)
            {
                DisplayError(e);
            }
            catch (FormatException e)
            {
                DisplayError(e);
            }
            catch (NotEnoughFundsException e)
            {
                DisplayError(e);
            }
        }

        /// <summary>
        /// Withdraws this instance.
        /// </summary>
        public void Withdraw()
        {
            try
            {
                BankAccount ba = this._View.BankAccount;
                double amount = this._View.Withdraw;
                this._Model.Withdraw(ba, amount);
            }
            catch (ArgumentNullException e)
            {
                DisplayError(e);
            }
            catch (ArgumentException e)
            {
                DisplayError(e);
            }
            catch (FormatException e)
            {
                DisplayError(e);
            }
            catch (NotEnoughFundsException e)
            {
                DisplayError(e);
            }
        }

        /// <summary>
        /// Displays the error.
        /// </summary>
        /// <param name="e">The e.</param>
        private void DisplayError(Exception e)
        {
            ErrorMessageCollection col = new ErrorMessageCollection();
            col.Add(new ErrorMessage(e.Message));
            this._View.DisplayValidationErrors(col);
        }

        /// <summary>
        /// Closes the application.
        /// </summary>
        public void CloseApplication()
        {
            if (this._View.ConfirmClose())
                Environment.Exit(1);
        }

        /// <summary>
        /// Loads the stock items from file.
        /// </summary>
        /// <param name="filePath">The file path.</param>
        public void LoadStockItemsFromFile(String filePath)
        {
            this._Model.LoadStockItemsFromFile(filePath);
        }

        /// <summary>
        /// Loads the bank accounts from file.
        /// </summary>
        /// <param name="filePath">The file path.</param>
        public void LoadBankAccountsFromFile(String filePath)
        {
            this._Model.LoadBankAccountsFromFile(filePath);
        }

        /// <summary>
        /// Saves the stock items to file.
        /// </summary>
        /// <param name="filePath">The file path.</param>
        public void SaveStockItemsToFile(String filePath)
        {
            this._Model.SaveStockItemsToFile(filePath);
        }

        /// <summary>
        /// Saves the bank accounts to file.
        /// </summary>
        /// <param name="filePath">The file path.</param>
        public void SaveBankAccountsToFile(String filePath)
        {
            this._Model.SaveBankAccountsToFile(filePath);
        }

        /// <summary>
        /// Sets up stock item file path.
        /// </summary>
        /// <param name="filePathStockItems">The file path stock items.</param>
        public void SetUpStockItemFilePath(string filePathStockItems)
        {
            // TODO: Check if good.
            this._Model.StockItemHandler.ReadFilePath = filePathStockItems;
            this._Model.StockItemHandler.WriteFilePath = filePathStockItems;
            this._Model.LoadStockItemsFromFile(filePathStockItems);
        }

        /// <summary>
        /// Sets up bank accounts file path.
        /// </summary>
        /// <param name="filePathBankAccounts">The file path bank accounts.</param>
        public void SetUpBankAccountsFilePath(string filePathBankAccounts)
        {
            // TODO: Check if good.
            this._Model.BankAccountHandler.ReadFilePath = filePathBankAccounts;
            this._Model.BankAccountHandler.WriteFilePath = filePathBankAccounts;
            this._Model.LoadBankAccountsFromFile(filePathBankAccounts);
        }

        /// <summary>
        /// Saves the bank accounts to file.
        /// </summary>
        public void SaveBankAccountsToFile()
        {
            try
            {
                this._Model.SaveBankAccountsToFile();
            }
            catch (NoFilePathSetException e)
            {
                DisplayError(e);
            }
        }

        /// <summary>
        /// Saves the stock items to file.
        /// </summary>
        public void SaveStockItemsToFile()
        {
            try
            {
                this._Model.SaveStockItemsToFile();
            }
            catch (NoFilePathSetException e)
            {
                DisplayError(e);
            }
        }
    }
}
```

Listing 21: CongregatePresenter.cs

A.3 TESTS

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

namespace NUnit_Tests.ApplicationLogic
{
    [TestFixture]
    public class BankAccountTest
    {
        private BankAccount ba;

        [SetUp]
        public void SetUp()
        {
            ba = new BankAccount(123, "Test", 0.0);
        }

        [Test]
        [ExpectedException(typeof(ArgumentException))]
        public void TestBalanceUnder0()
        {
            ba = new BankAccount(123, "Test", -1.0);
        }

        [Test]
        [ExpectedException(typeof(NotEnoughFundsException))]
        public void TestWithdrawalWithTooHighValues()
        {
            double amountToWithdrawTooHigh = 50.0;
            ba.Withdraw(amountToWithdrawTooHigh);
        }

        [Test]
        [ExpectedException(typeof(ArgumentException))]
        public void TestWithdrawalWithTooSmallValues()
        {
            double amountToWithdrawTooSmall = -10.0;
            ba.Withdraw(amountToWithdrawTooSmall);
        }

        [Test]
        [ExpectedException(typeof(ArgumentException))]
        public void TestDepositWithTooSmallValue()
        {
            double amountToDepositTooSmall = -10;
            ba.Deposit(amountToDepositTooSmall);
        }

        [Test]
        [ExpectedException(typeof(NotEnoughFundsException))]
        public void TestTransferWithTooHighValue()
        {
            double amountToTransferTooHigh = 10;
            ba.Transfer(123, amountToTransferTooHigh);
        }

        [Test]
        [ExpectedException(typeof(ArgumentException))]
        public void TestTransferWithTooSmallValue()
        {
            double amountToTransferTooSmall = -10;
            ba.Transfer(123, amountToTransferTooSmall);
        }

        [Test]
        public void TestOrdinaryFunctions()
        {
            double currentValue = ba.Balance;
            double deposit = 50.0;
            currentValue += deposit;
            ba.Deposit(deposit);
            Assert.AreEqual(currentValue, ba.Balance);

            double withdraw = 25.0;
            currentValue -= withdraw;
            ba.Withdraw(withdraw);
            Assert.AreEqual(currentValue, ba.Balance);

            double transfer = 10.50;
            currentValue -= transfer;
            ba.Transfer(123, transfer);
            Assert.AreEqual(currentValue, ba.Balance);
        }

        [Test]
        public void TestDepositWithdrawSameAmount()
        {
            double amount = 50;
            double value = 0;
            ba.Deposit(amount);
            ba.Withdraw(amount);
            Assert.AreEqual(value, ba.Balance);
        }

        [Test]
        public void TestTransferAllMoney()
        {
            double amount = 50;
            double value = 0;
            ba.Deposit(amount);
            ba.Transfer(0, amount);
            Assert.AreEqual(value, ba.Balance);
        }

                [Test]
                public void TestStringParsing()
                {
                        BankAccount parseAccount = new BankAccount();

                        String parseOne = "123456,Rambo,500.50";
                        BankAccount ba1 = parseAccount.ParseFromString(parseOne);

                        String parseTwo = "000000,,";
                        BankAccount ba2 = parseAccount.ParseFromString(parseTwo);
                }

                [Test]
                [ExpectedException(typeof(FormatException))]
                public void TestStringParsingInvalidValues()
                {
                        BankAccount parseAccount = new BankAccount();

                        String parseOne = "abcd,Rambo,50.50";
                        BankAccount ba1 = parseAccount.ParseFromString(parseOne);
                }
        }
}
```

Listing 22: BankAccountTest.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using ApplicationLogic;
using ApplicationLogic.Model;
using NUnit.Framework;
namespace NUnit_Tests
{
        [TestFixture()]
        public class FileHandlerTest
        {
                [Test]
                [ExpectedException(typeof(NoFilePathSetException))]
                public void TestWriteToFileWithoutPath ()
                {
                        FileHandler<StockItem> fh = new FileHandler<StockItem>();
                        fh.SaveToFile(new List<StockItem>());
                }

                [Test]
                [ExpectedException(typeof(NoFilePathSetException))]
                public void TestReadFromFileWithoutPath()
                {
                        FileHandler<StockItem> fh = new FileHandler<StockItem>();
                        fh.LoadFromFile(new StockItem());
                }
        }
}
```

Listing 23: FileHandlerTest.cs

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

namespace NUnit_Tests
{
    [TestFixture]
    public class StockItemTest
    {
        private StockItem si;

        [SetUp]
        public void SetUp()
        {
            si = new StockItem("1234", "Test", "Test", 10.0, 5, 5);
        }

        [Test]
        [ExpectedException(typeof(ArgumentException))]
        public void TestLessThan0Cost()
        {
            si = new StockItem("1234", "Test", "Test", -1.0, 5, 5);
        }

        [Test]
        [ExpectedException(typeof(ArgumentException))]
        public void TestLessThan0CurrentStock()
        {
            si = new StockItem("1234", "Test", "Test", 1.0, -1, 5);
        }

        [Test]
        [ExpectedException(typeof(ArgumentException))]
        public void TestLessThan0RequiredStock()
        {
            si = new StockItem("1234", "Test", "Test", 1.0, 5, -1);
        }

        [Test]
        public void TestIsValidStockCode()
        {
            bool validSC = StockItem.IsValidStockCode("1234");
            Assert.IsTrue(validSC);
            bool tooLongSC = StockItem.IsValidStockCode("123456");
            Assert.IsFalse(tooLongSC);
            bool stringSC = StockItem.IsValidStockCode("test");
            Assert.IsFalse(stringSC);
        }

        [Test]
        [ExpectedException(typeof(ArgumentNullException))]
        public void TestIsValidStockCodeRaiseException()
        {
            StockItem.IsValidStockCode(null);
        }

        [Test]
        [ExpectedException(typeof(ArgumentException))]
        public void TestCreateStockItemInvalidStockCode()
        {
            String invalidStockCode = "00001";
            StockItem si = new StockItem(invalidStockCode, "", "", 0.0, 0, 0);
        }

        [Test]
        [ExpectedException(typeof(ArgumentException))]
        public void TestCreateStockItemInvalidCost()
        {
            double invalidCost = -1.0;
            StockItem si = new StockItem("0001", "", "", invalidCost, 0, 0);
        }

        [Test]
        [ExpectedException(typeof(ArgumentException))]
        public void TestCreateStockItemInvalidRequiredStock()
        {
            int invalidStock = -1;
            StockItem si = new StockItem("0001", "", "", 0.0, invalidStock, 0);
        }

        [Test]
        [ExpectedException(typeof(ArgumentException))]
        public void TestCreateStockItemInvalidCurrentStock()
        {
            int invalidStock = -1;
            StockItem si = new StockItem("0001", "", "", 0.0,  0, invalidStock);
        }

                [Test]
                public void TestStringParsing()
                {
                        StockItem parseItem = new StockItem();

                        String parseOne = "0001,Pencil Holder,John Rambo,5.50,10,15";

                        StockItem si = parseItem.ParseFromString(parseOne);
                        Assert.IsNotNull(si);

                        String parseTwo = "0001,,John Rambo,5.50,10,15";

                        StockItem si2 = parseItem.ParseFromString(parseTwo);
                        Assert.IsNotNull(si2);

                        String parseThree = "0001,,,,,";

                        StockItem si3 = parseItem.ParseFromString(parseThree);
                        Assert.IsNotNull(si3);
                }

                [Test]
                [ExpectedException(typeof(FormatException))]
                public void TestStringParsingInvalidValues()
                {
                        StockItem parseItem = new StockItem();

                        String parseOne = "0001,Pencil Holder,John Rambo,abc,10,15";
                        StockItem si = parseItem.ParseFromString(parseOne);
                }
    }
}
```

Listing 24: StockItemTest.cs

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

namespace NUnit_Tests.ApplicationLogic
{
    [TestFixture]
    public class AppDataManagerTest
    {
        private AppDataManager _Manager;
        private StockItem _Stock;
        private BankAccount _Account;

        [SetUp]
        public void SetUp()
        {
            this._Manager = new AppDataManager();
            this._Stock = new StockItem();
            this._Account = new BankAccount();
        }

        [Test]
        public void TestAddStockItem()
        {
            this._Manager.CreateNewStockItem();
            Assert.AreEqual(1, this._Manager.StockItems.Count);
        }

        [Test]
        public void TestAddBankAccount()
        {
            this._Manager.CreateNewBankAccount();
            Assert.AreEqual(1, this._Manager.BankAccounts.Count);
        }

        [Test]
        public void TestRemoveStockItem()
        {
            this._Manager.CreateNewStockItem();
            Assert.AreEqual(1, this._Manager.StockItems.Count);
            StockItem remove = this._Manager.StockItems.ElementAt(0);
            this._Manager.DeleteStockItem(remove);
            Assert.AreEqual(0, this._Manager.StockItems.Count);
        }

        [Test]
        public void TestRemoveBankAccount()
        {
            this._Manager.CreateNewBankAccount();
            Assert.AreEqual(1, this._Manager.BankAccounts.Count);
            BankAccount remove = this._Manager.BankAccounts.ElementAt(0);
            this._Manager.DeleteBankAccount(remove);
            Assert.AreEqual(0, this._Manager.BankAccounts.Count);
        }

        [Test]
        [ExpectedException(typeof(ArgumentException))]
        public void TestRemoveStockItemNotPresent()
        {
            StockItem si = new StockItem();
            this._Manager.DeleteStockItem(si);
        }

        [Test]
        [ExpectedException(typeof(ArgumentException))]
        public void TestRemoveBankAccountNotPresent()
        {
            BankAccount ba = new BankAccount();
            this._Manager.DeleteBankAccount(ba);
        }

        [Test]
        public void TestCorrectSequenceOfOrdering()
        {
            var mockBa = new Mock<BankAccount>();
            var mockSi = new Mock<StockItem>();

            mockBa.Setup(ba => ba.Balance).Returns(50.0);

            this._Manager.BankAccounts.Add(mockBa.Object);
            this._Manager.StockItems.Add(mockSi.Object);
            this._Manager.OrderItem(mockSi.Object, mockBa.Object, 0);

            mockBa.VerifyGet(ba => ba.Balance);
            mockBa.Verify(ba => ba.Transfer(0, 10.0), Times.AtMostOnce());
        }

        [Test]
        [ExpectedException(typeof(NotEnoughFundsException))]
        public void TestCorrectSequenceOnInvalidFunds()
        {
            var mockBa = new Mock<BankAccount>();
            var mockSi = new Mock<StockItem>();

            mockBa.Setup(ba => ba.Balance).Returns(0.0);
            mockSi.Setup(si => si.UnitCost).Returns(10.0);

            this._Manager.BankAccounts.Add(mockBa.Object);
            this._Manager.StockItems.Add(mockSi.Object);
            this._Manager.OrderItem(mockSi.Object, mockBa.Object, 1);

            mockBa.VerifyGet(ba => ba.Balance);
            mockBa.Verify(ba => ba.Transfer(0, 10.0), Times.Never());
        }
    }
}
```

Listing 25: AppDataManagerTest.cs

BIBLIOGRAPHY

Balzert, Helmut (2009). *Lehrbuch der Software-Technik: Basiskonzepte und Requirements Engineering*. 3. Heidelberg: Spektrum. ISBN: 9783827417053.

Boodhoo, Jean-Paul (2006). *Design Patterns: Model View Presenter*. English. Microsoft. URL: http://msdn.microsoft.com/en-us/magazine/cc188690.aspx (visited on 19/10/2010).

Dorman, Scott (2010). *Sams Teach Yourself Visual CSharp®2010 in 24 Hours: Complete Starter Kit*. Sams Publishing. ISBN: 978-0-672-33101-5.

Freeman, Eric and Elisabeth Freeman (2004). *Head First - Design Pattern*. Ed. by Mike Loukides. O'Reilly. ISBN: 0-596-00712-4.

Noyes, Brian (2006). *Data Binding With Windows Forms 2.0 - Programming Smart Client Data Applications With .NET*. Addison Wesley Professional. ISBN: 978-0-321-26892-1.

Sommerville, Ian (2006). *Software Engineering*. 8th ed. Addison Wesley. ISBN: 9780321210265.

Stellman, Andrew and Jennifer Greene (2010). *Head First CSharp*. O'Reilly. ISBN: 978-1-449-38034-2.