|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Submission Coversheet | | | |  |  |  |  |
|  |  |  |  |  |  |  |  |
| **Student ID Number** | S00041694 | | | | | | |
| **Programme Title** | Computing Technologies Degree | | | | | | |
| **Module Title** | Object Oriented Programming | | | | | | |
| **Module Code** *(listed on Moodle and in LTAFP)* | QAC020C152A | | | | | | |
| **Module Convenor** | Tendai Mhlanga | | | | | | |
| **Coursework Title** | Coursework1: Ten Thousands Steps Snack Machines | | | | | | |
| **Academic Declaration:**  *Students are reminded that the electronic copy of their essay may be checked, at any point during their degree, with Turnitin or other plagiarism detection software for plagiarised material.* | | | | | | | |
| **Word Count** | 7458 | | | **Date Submitted** | | **26/10/2018** | |
|  |  | | |  | |  | |

Table of Contents

[Project Details 4](#_Toc528346824)

[Project Name 4](#_Toc528346825)

[Description 4](#_Toc528346826)

[Requirements Details 4](#_Toc528346827)

[Specifications 4](#_Toc528346828)

[Requested Features 4](#_Toc528346829)

[Application Design 5](#_Toc528346830)

[Architecture 5](#_Toc528346831)

[Design Patterns 5](#_Toc528346832)

[Singleton Design Pattern 5](#_Toc528346833)

[Factory Method Design Pattern 6](#_Toc528346834)

[Class Diagram 7](#_Toc528346835)

[User Interface 8](#_Toc528346836)

[Initial State of Main Menu 8](#_Toc528346837)

[Selecting Crisps Snacks 9](#_Toc528346838)

[Selecting Mars Bar Snacks 10](#_Toc528346839)

[Selecting Coca Cola Snacks 10](#_Toc528346840)

[Selecting Eugenia Snacks 11](#_Toc528346841)

[Selecting Water Snacks 11](#_Toc528346842)

[Selecting Snack Quantity 12](#_Toc528346843)

[Selecting Coins 12](#_Toc528346844)

[Insert Coin 13](#_Toc528346845)

[Pay For Snacks 13](#_Toc528346846)

[Cash In Change 14](#_Toc528346847)

[Login Page 14](#_Toc528346848)

[Report Page 15](#_Toc528346849)

[Popup Error Message for Selected Snack Quantity 15](#_Toc528346850)

[Popup Error Message for Out Of Stock Failure 15](#_Toc528346851)

[Popup Error Message for Not Enough Coins Inserted 16](#_Toc528346852)

[Popup Error Message for Maximum Coins Allowed 16](#_Toc528346853)

[Popup Error Message for Invalid Login 16](#_Toc528346854)

[Popup Error Message for No Change Available 16](#_Toc528346855)

[Implementation 17](#_Toc528346856)

[Packages 17](#_Toc528346857)

[Main Packages 17](#_Toc528346858)

[Secondary Packages 17](#_Toc528346859)

[Classes, Abstract Classes, Interfaces and Enums 17](#_Toc528346860)

[Main class 17](#_Toc528346861)

[MainJFrame class 17](#_Toc528346862)

[AdminJDialog class 20](#_Toc528346863)

[CrispsSelectionAction class 21](#_Toc528346864)

[MarsSelectionAction class 21](#_Toc528346865)

[CokeSelectionAction class 21](#_Toc528346866)

[EugeniaSelectionAction class 22](#_Toc528346867)

[WaterSelectionAction class 22](#_Toc528346868)

[InsertCoinAction class 23](#_Toc528346869)

[PayForSnackAction class 23](#_Toc528346870)

[PayForSnackWorker class 24](#_Toc528346871)

[CashInAction class 25](#_Toc528346872)

[Model class 25](#_Toc528346873)

[ChangeDisperser class 29](#_Toc528346874)

[SnacksFactory class 32](#_Toc528346875)

[ISnack interface 32](#_Toc528346876)

[Snack abstract class 32](#_Toc528346877)

[Crisps class 33](#_Toc528346878)

[Crisps MarsBar class 33](#_Toc528346879)

[CocaCola class 34](#_Toc528346880)

[Eugenia class 34](#_Toc528346881)

[Water class 34](#_Toc528346882)

[ESnackTypes enum 35](#_Toc528346883)

[ECoinTypes enum 35](#_Toc528346884)

[Defines class 35](#_Toc528346885)

[SyncUpData class 35](#_Toc528346886)

[AdminReport class 35](#_Toc528346887)

[Test Documentation 35](#_Toc528346888)

[Appendix A: Source Code 40](#_Toc528346889)

[Appendix B: References 40](#_Toc528346890)

# Project Details

## Project Name

Ten Thousands Steps Snack Machines

## Description

The Project Ten Thousand Steps Snack Machines is a cross platform software solution designed as a vending machine product for consumers who are engaged in loosing/maintaining their weight or just want a healthier life. The application will be distributed in universities across London to introduce to the end user a new and intriguing range of low fat, salt and sugar snacks with competitive prices.

## Requirements Details

### Specifications

The software application has to work as any vending machine with a user friendly interface, having a given set of snacks with their related prices. The current prices are £0.75, £0.7, £1, £0.5 and £0.85 for their associated snacks Crisps, Mars Bar, Coca Cola, Eugenia and Water, respectively. The company buys the snacks at the price of £0.45 and replenishes the vending machines with a maximum of 10 per individual article.

The snack machines should give change from the current pool of denominations and the coins should be inserted one by one and should be added to the existing stock of change. The accepted change denominations are 5p, 10p, 20p, 50p and £1.00 with a initial stock of 20, 20, 20 ,20 and 10, respectively.

### Requested Features

The user should have a choice in selecting his quantity for the current selected snack and should be able to choose a denomination from the 5 options available before he is ready to purchase. After the stock and payment have been checked, meaning that the snacks are available, and the price is met or exceeded, the item should be dispensed, and the new quantity should be updated and displayed in the user interface. If there is any change remaining from the total amount of money inserted for the snack, after it’s quantity has been dispensed, it should be handed out to its owner.

As is expected from any vending machine, there should be a admin page which is used for presenting some crucial data like total profit, losses and total money contained in the machine. Before accessing the report page the user should be prompted for entering the user with the input “10976” and his predefined password “1234”.

The automat should show error messages informing the user that the requirements has not been met. For example if the total price of the snacks is not equal or above the coins inserted, or if a snack is out of stock, or maybe deny a denomination which doesn’t exist.

# Application Design

## Architecture

As you can see below in Figure 1 the main architectural pattern adopted in this project is the Model View Controller known as short for MVC. I chose this design because of it’s well known advantages it brings to the table like efficient code reuse, parallel development, good division between components, user interface decoupling from processing data and internal implementation logic and least but not the last because of Java’s embedded Swing Framework which inherently has a good MVC API.

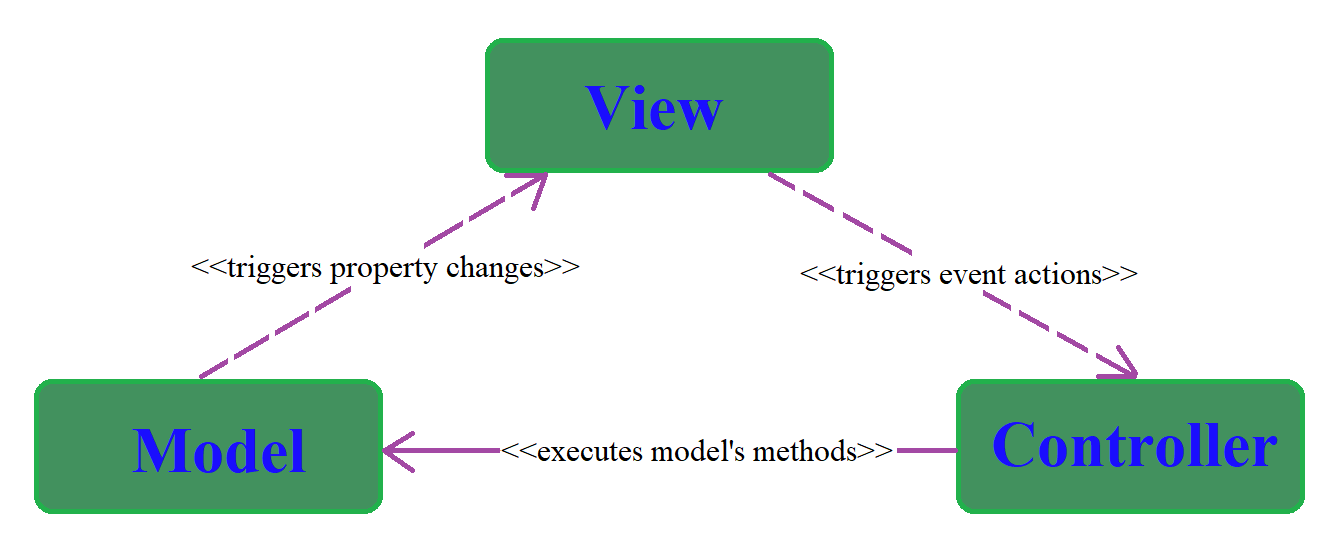


Figure 1

## Design Patterns

### Singleton Design Pattern

As shown in Figure 2 every type of snack from the vending machine is designed as a singleton class. Singleton pattern is used because of its unique property to allocate memory only once for a particular class, creating every type of snack as a common entity. This limitation is needed because allowing multiple instances of the same snack will give way to bugs and alter the applications logic if other software developers will choose to do it.

Another reason why this design pattern was implemented is because it helps with memory optimization by setting the quantity of each type of snack inside the object. So there is no need to create multiple instances of the same snack to represent their amount inside the vending machine therefore Singleton is actually the best choice for this scenario.

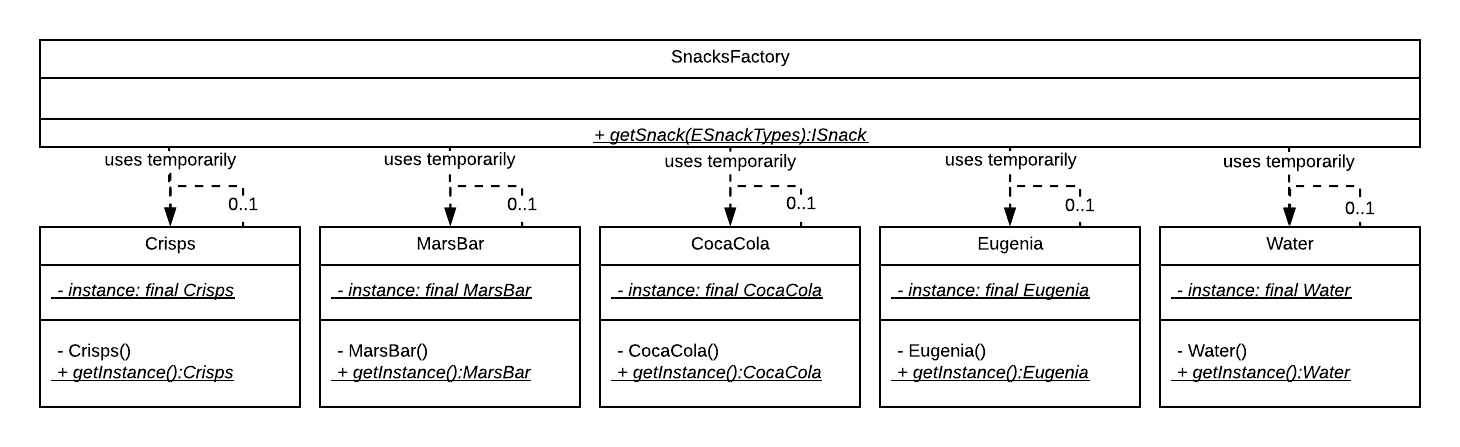


Figure 2

### Factory Method Design Pattern

As presented in Figure 3 the class SnacksFactory represents the core logic behind Factory Method design pattern and it is used to create snack objects according to their type. This pattern is useful for deferring creation logic from the model and interprets all the snacks as one generalized template, in our case ISnack interface.

Due to the magic of polymorphism all snacks are defined by one abstract layer but behave accordingly to their type, and because of this they can be stored in a variety of lists and can be accessed one by one at any given time.

As deducted from the figure below the snack classes Crisps, MarsBar, CocaCola, Eugenia and Water extend the Snack abstract class which in turn implements the ISnack interface, being bounded to acknowledge and override all its methods. The SnackFactory class uses the commonly defined static method getInstance(), residing in all snack classes, to create and return the instance of a particular snack object.

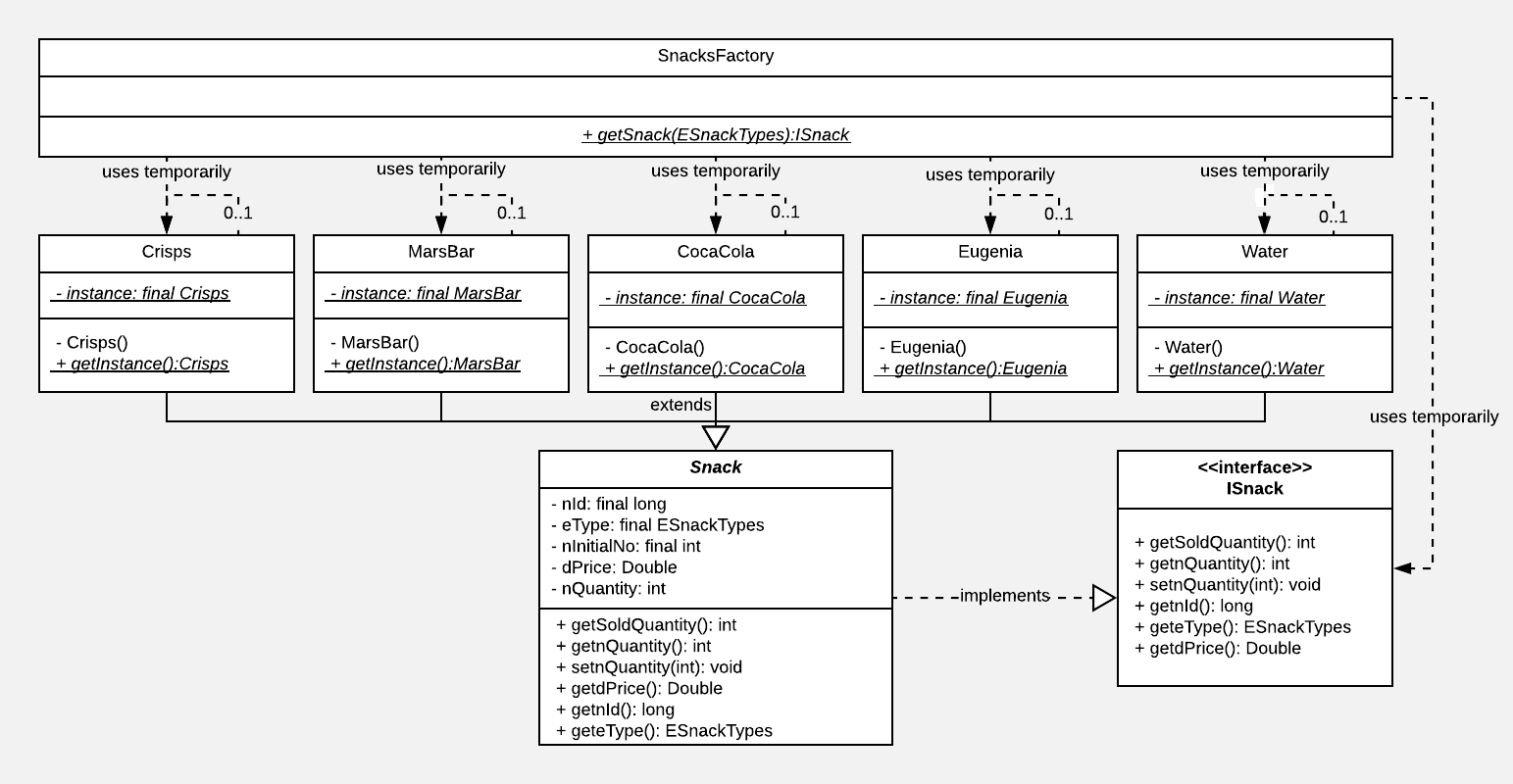


Figure 3

## Class Diagram

The project’s Class Diagram represents the schematic of all existing classes, their relationships and their association multiplicity.

As shown below in Figure 4, the main view represented by the MainJFrame class contains or has access to the admin view class, all the controller classes and the module class. This is represented by the composition association and the multiplicity value of 1.

The second view class AdminJDialog is contained by the main view as stated earlier and because it doesn’t directly create the model it needs to hold a reference of the model. This is defined by the aggregation association between these two classes. The admin view has its own controller class called LoginAction and is expressed through the composition relationship between them.

The remaining controllers like InsertCoinAction, PayForSnackAction, CashInAction, CrispsSelectionAction, MarsSelectionAction, CokeSelectionAction, EugeniaSelectionAction and WaterSelectionAction classes are part of the central view and are expressed as composition associations. All controllers contain a reference of the model for management purposes and this is symbolized in the image below as aggregation relationships with the multiplicity factor of 1.

The Model class contains all the data that is needed to be stored and/or presented to the user interface. Some of the objects that it contains are the list of snacks, a change disperser, the user and password for the login page and so on. All these objects are used temporarily or have a composition relantioship between them.

The enum structures ESnackTypes and ECoinTypes are utilised in almost all classes, from viewers to controllers to model and data classes. These enums are used for category defining, for hash map searching, inserting and extracting information, for object memory allocation, for sending user interface states and so on.

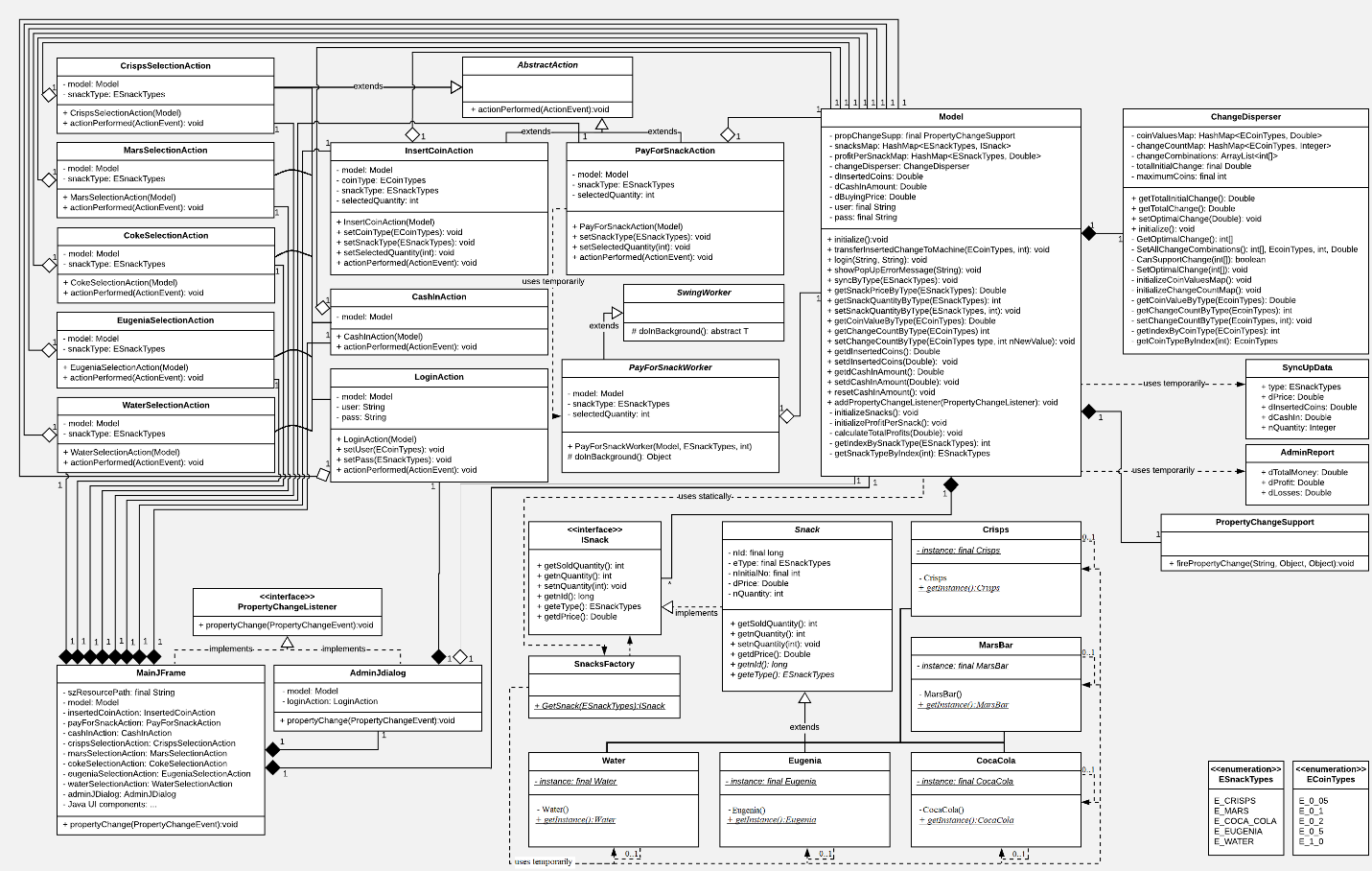


Figure 4

# User Interface

The User Interface was created using Java’s UI components like JFrame, JDialog, JButton, JToggleButton, JLabel, JSlider and custom images edited and/or created manually with the Microsoft’s Windows 10 Paint application. The only images which were copied from the internet are the ones which represent the snacks.

## Initial State of Main Menu

As represented in Figure 5, when the application is first opened, the main menu starts with a small introductory title, a label which encourages the user to select a snack and the actual snacks, which are ordered vertically in the left part of the view from top to bottom as buttons containing their representative image.

The main canvas is divided into two different backgrounds for differentiating the snack selection and the actual representation of the vending machine. The white background helps the button’s image blend in, and in contrast the green colour defines and embodies the companies healthy snack ideology.

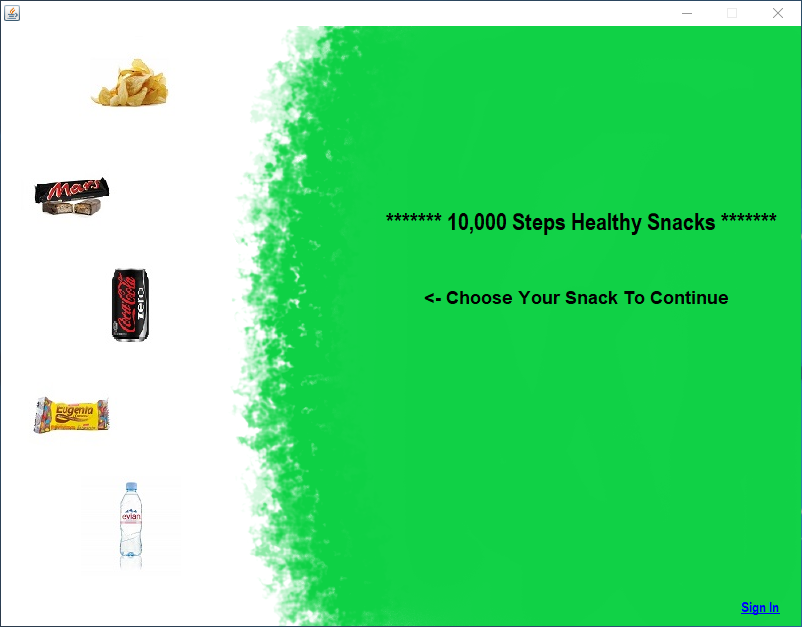


Figure 5

## Selecting Crisps Snacks

When selecting the Crisps Snack the button is highlighted with a green contour and a green coloured badge which has the current quantity of snacks available. On the green background a functional snack machine appears containing the image of the chosen item.

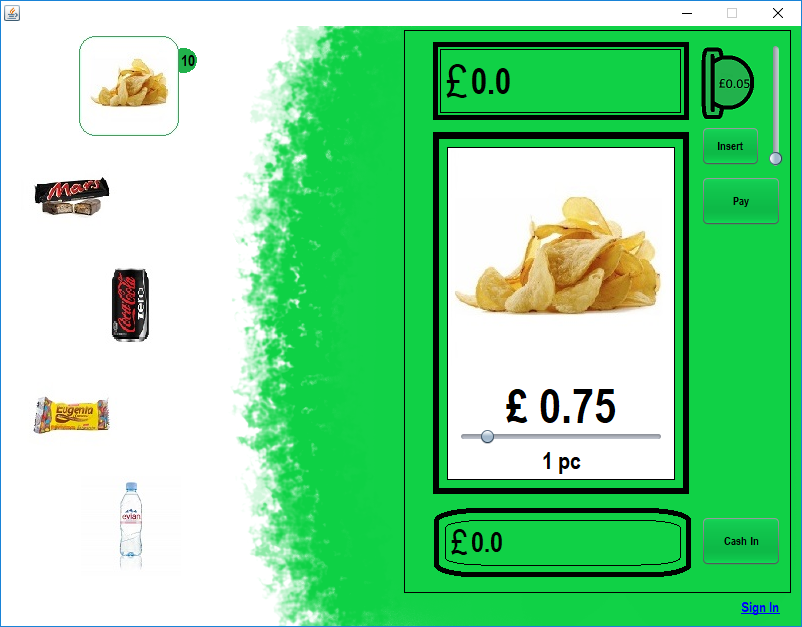


Figure 6

## Selecting Mars Bar Snacks

When selecting the Mars Bar Snack the button is highlighted with a green contour and a green coloured badge which has the current quantity of snacks available. On the green background a functional snack machine appears containing the image of the chosen item.



Figure 7

## Selecting Coca Cola Snacks

When selecting the Coca Cola Snack the button is highlighted with a green contour and a green coloured badge which has the current quantity of snacks available. On the green background a functional snack machine appears containing the image of the chosen item.



Figure 8

## Selecting Eugenia Snacks

When selecting the Coca Cola Snack the button is highlighted with a green contour and a green coloured badge which has the current quantity of snacks available. On the green background a functional snack machine appears containing the image of the chosen item.



Figure 9

## Selecting Water Snacks

When selecting the Coca Cola Snack the button is highlighted with a green contour and a green coloured badge which has the current quantity of snacks available. On the green background a functional snack machine appears containing the image of the chosen item.



Figure 10

## Selecting Snack Quantity

Quantity selection of a chosen snack is represented in the image below by the slider inside the red rectangle. The slider’s maximum value is automatically set according to the current snack’s quantity. The depicted amount auto updates in real time when slid left or right.



Figure 11

## Selecting Coins

The selection of the value of a coin is embodied by the slider inside the red rectangle from Figure 12. The slider has a maximum of 5 coins and automatically changes the image of the coin when selecting up or down.



Figure 12

## Insert Coin

The insertion of a coin action is represented in figure below by the button inside the red rectangle.

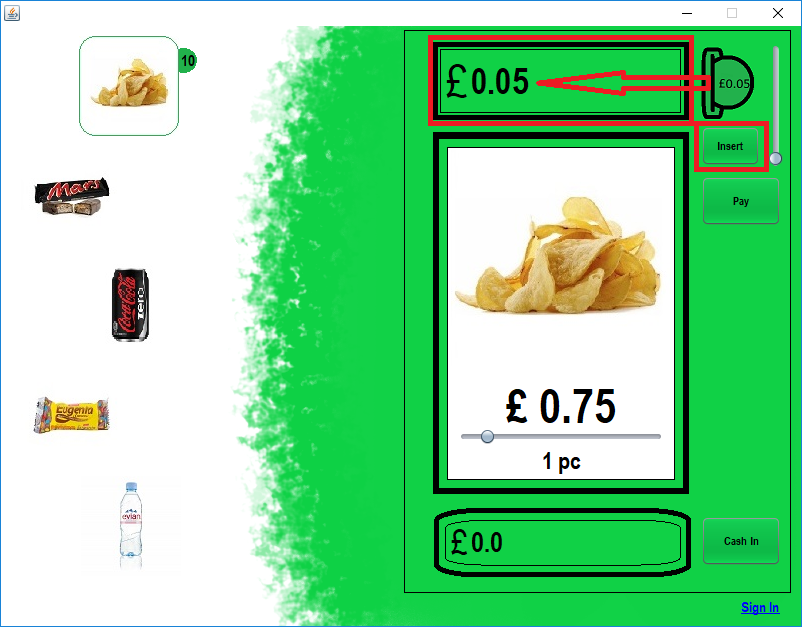


Figure 13

## Pay For Snacks

The paying action is represented in figure below by the button inside the red rectangle.

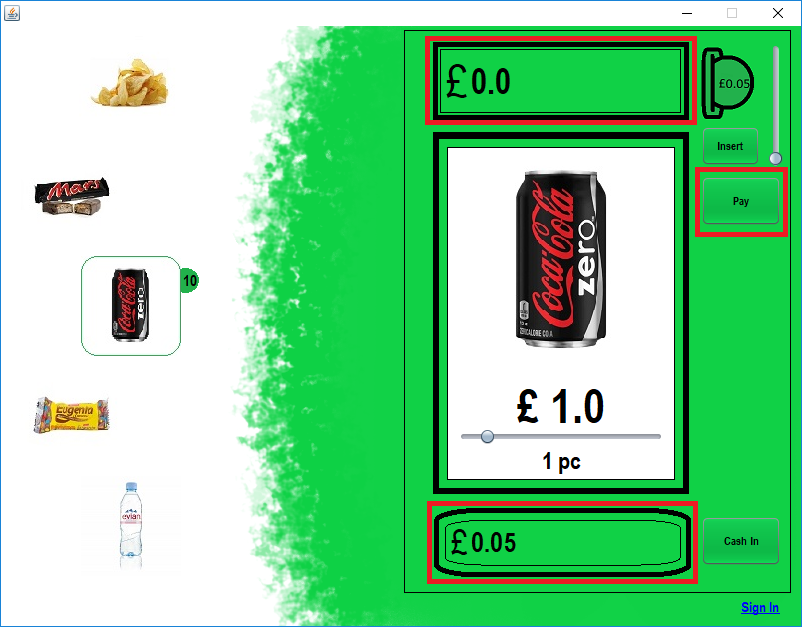


Figure 14

## Cash In Change

The cash in action is represented in figure below by the button inside the red rectangle and it just resets the users remaining change.



Figure 15

## Login Page

The Login Page is a modal dialog that requests user and password credentials to sign in into the Report Page.

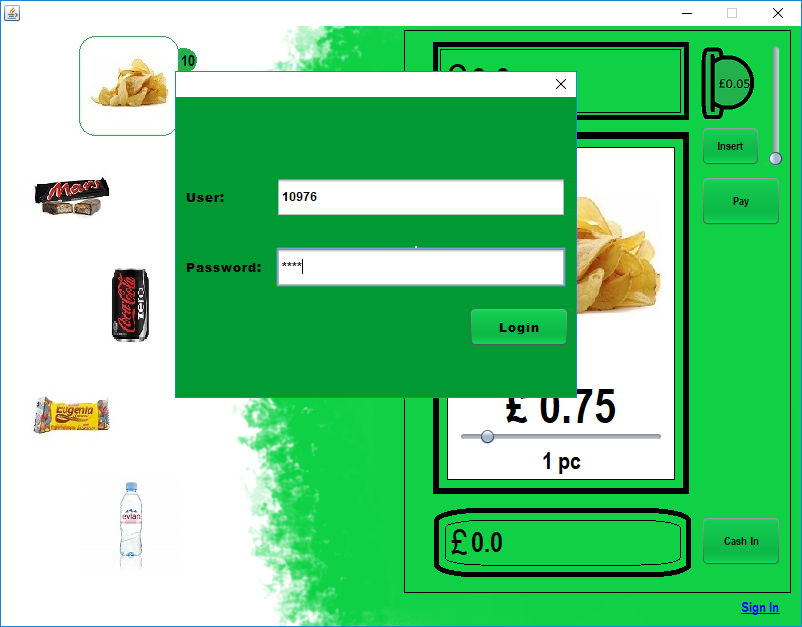


Figure 16

## Report Page

The Report Page is in actuality another state of the Login Page and it extracts data from the model like profit, losses and total to present them on the page.



Figure 17

## Popup Error Message for Selected Snack Quantity

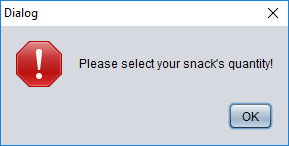


Figure 18

## Popup Error Message for Out Of Stock Failure

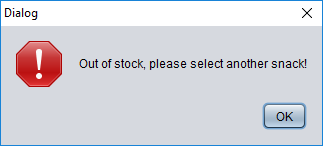


Figure 19

## Popup Error Message for Not Enough Coins Inserted

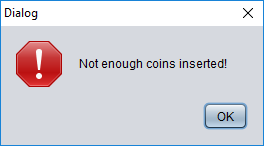


Figure 20

## Popup Error Message for Maximum Coins Allowed

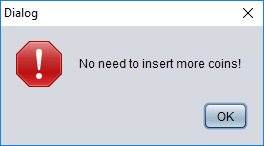


Figure 21

## Popup Error Message for Invalid Login

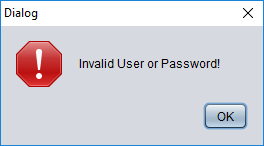


Figure 22

## Popup Error Message for No Change Available

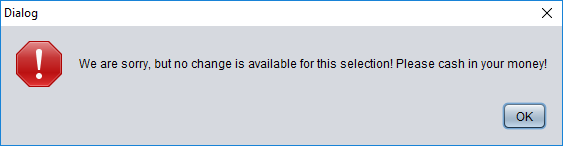


Figure 23

# Implementation

## Packages

The Ten Thousands Steps Snack Machines project is divided into four main packages and two secondary packages.

### Main Packages

* <default package> : it only contains the Main Java class and is the entry point for the entire project;
* view : it contains the MainJFrame and AdminJDialog classes. These classes represent only the user interface.
* controller : it contains the InsertCoinAction, PayForSnackAction, CashInAcftion, CrispsSelectionAction, MarsSelectionAction, CokeSelectionAction, EugeniaSelectionAction and WaterSelectionAction, LoginAction and PayForSnackWorker classes. These classes only manage the data logics.
* model : it contains the Model and SnacksFactory classes. These classes constitute the logic of data manipulation.

### Secondary Packages

* data : it contains Crisps, MarsBar, CocaCola, Eugenia, Water, ChangeDiperser, SyncUpData, AdminReport and Defines classes, Snack abstract class, ISnacks interface and the enums ESnackTypes, ECoinTypes. They amount to the actual hard data where everything is stored.
* resources : this package contains all the images the project uses.

## Classes, Abstract Classes, Interfaces and Enums

### Main class

This class is used as a entry point because it contains the main method, and it allocates memory for a MainJFrame object and sets the page as visible.

### MainJFrame class

The MainJFrame class extends JFrame java class and implements the PropertyChangeListener interface. The PropertyChangeListener interface is used for indirectly connecting the model to the viewer by registering to catch events when data changes.

The class contains the instances of the AdminJDialog report view, all the controllers except LoginAction, the Model and of course the view components.

In the constructor all the class members are instantiated, initialized, all contained controllers are bound with the view components and the initial state of the view is set.

Public Methods:

/\*\*

\* Catches property change event

\* @param pce

\*/

@Override

public void propertyChange(PropertyChangeEvent pce);

Private Methods:

/\*\*

\* Selects Crisps snack, sets up the state of the main view

\* and sets the coin type for some controllers.

\* @param evt

\*/

private void jToggleButton1ActionPerformed(java.awt.event.ActionEvent evt);

/\*\*

\* Selects Mars Bar snack, sets up the state of the main view

\* and sets the coin type for some controllers.

\* @param evt

\*/

private void jToggleButton2ActionPerformed(java.awt.event.ActionEvent evt);

/\*\*

\* Selects Coca Cola snack, sets up the state of the main view

\* and sets the coin type for some controllers.

\* @param evt

\*/

private void jToggleButton3ActionPerformed(java.awt.event.ActionEvent evt);

/\*\*

\* Selects Eugenia snack, sets up the state of the main view

\* and sets the coin type for some controllers.

\* @param evt

\*/

private void jToggleButton4ActionPerformed(java.awt.event.ActionEvent evt);

/\*\*

\* Selects Water, sets up the state of the main view

\* and sets the coin type for some controllers.

\* @param evt

\*/

private void jToggleButton5ActionPerformed(java.awt.event.ActionEvent evt);

/\*\*

\* Selects the coin type in the slider, sets up a image according to the chosen coin

\* and sets the coin type for InsertedCoinAction controller.

\* @param evt

\*/

private void jSlider1StateChanged(javax.swing.event.ChangeEvent evt);

/\*\*

\* Changes the amount for selected snack in the slider, sets up the value in a label

\* according to the chosen amount and updates the amount for some controllers.

\* @param evt

\*/

private void jSlider2StateChanged(javax.swing.event.ChangeEvent evt);

/\*\*

\* Works as a button for the sign in. Resets the state of the admin page and sets it as visible

\* @param evt

\*/

private void jLabel21MouseClicked(java.awt.event.MouseEvent evt) ;

/\*\*

\* Sets the state of Crisps badge

\* @param bValue

\*/

private void setButton1BadgeState(boolean bValue);

/\*\*

\* Sets the state of Mars Bar badge

\* @param bValue

\*/

private void setButton2BadgeState(boolean bValue);

/\*\*

\* Sets the state of Coca Cola badge

\* @param bValue

\*/

private void setButton3BadgeState(boolean bValue);

/\*\*

\* Sets the state of Eugenia badge

\* @param bValue

\*/

private void setButton4BadgeState(boolean bValue);

/\*\*

\* Sets the state of Water badge

\* @param bValue

\*/

private void setButton5BadgeState(boolean bValue);

/\*\*

\* Sets the state of the snack machine as visible or invisible

\* @param bValue

\*/

private void ShowSnackMachineInFront(boolean bValue);

/\*\*

\* Gets the type of a coin by his referenced integer value

\* @param nValue

\*/

private ECoinTypes transformIntToCoinType(int nValue);

/\*\*

\* Catches property change event

\* @param pce

\*/

@Override

public void propertyChange(PropertyChangeEvent pce);

### AdminJDialog class

The AdminJDialog class extends JDialog java class and implements the PropertyChangeListener interface. Same as in the main view the PropertyChangeListener interface is used for indirectly connecting the model to the.

The class contains the instances of the Model class, LoginAction controller and the view components.

In the constructor all the class members are instantiated, initialized, the login controller is tied with the button which signs in and the initial state of the view is set.

Public Methods:

/\*\*

\* Catches property change event

\* @param pce

\*/

@Override

public void propertyChange(PropertyChangeEvent pce);

/\*\*

\*Resets the report page to look like a login page

\*/

public void resetToLogin();

### CrispsSelectionAction class

The CrispsSelectionAction class extends AbstractAction java abstract class so that it can be set as a listener to a view component, in our case a button, and to catch the action event of that button.

The class contains the reference of the Model object and the type of the selected snack.

In the constructor the reference of the model is assigned to its member variable.

Public Methods:

/\*\*

\*triggered action event from view component

\* @param ae

\*/

@Override

public void actionPerformed(ActionEvent ae);

### MarsSelectionAction class

The MarsSelectionAction class extends AbstractAction java abstract class so that it can be set as a listener to a view component, in our case a button, and to catch the action event of that button.

The class contains the reference of the Model object and the type of the selected snack.

In the constructor the reference of the model is assigned to its member variable.

Public Methods:

/\*\*

\*triggered action event from view component

\* @param ae

\*/

@Override

public void actionPerformed(ActionEvent ae);

### CokeSelectionAction class

The CokeSelectionAction class extends AbstractAction java abstract class so that it can be set as a listener to a view component, in our case a button, and to catch the action event of that button.

The class contains the reference of the Model object and the type of the selected snack.

In the constructor the reference of the model is assigned to its member variable.

Public Methods:

/\*\*

\*triggered action event from view component

\* @param ae

\*/

@Override

public void actionPerformed(ActionEvent ae);

### EugeniaSelectionAction class

The EugeniaSelectionAction class extends AbstractAction java abstract class so that it can be set as a listener to a view component, in our case a button, and to catch the action event of that button.

The class contains the reference of the Model object and the type of the selected snack.

In the constructor the reference of the model is assigned to its member variable.

Public Methods:

/\*\*

\*triggered action event from view component

\* @param ae

\*/

@Override

public void actionPerformed(ActionEvent ae);

### WaterSelectionAction class

The WaterSelectionAction class extends AbstractAction java abstract class so that it can be set as a listener to a view component, in our case a button, and to catch the action event of that button.

The class contains the reference of the Model object and the type of the selected snack.

In the constructor the reference of the model is assigned to its member variable.

Public Methods:

/\*\*

\*triggered action event from view component

\* @param ae

\*/

@Override

public void actionPerformed(ActionEvent ae);

### InsertCoinAction class

The InsertCoinAction class extends AbstractAction java abstract class so that it can be set as a listener to a view component, in our case a button, and to catch the action event of that button.

The class contains the reference of the Model object, the type of the selected snack, the type of the coin type and the quantity of the selected item.

In the constructor the reference of the model is assigned to its member variable.

Public Methods:

/\*\*

\* Sets the selected coin type

\* @param coinType

\*/

public void setCoinType(ECoinTypes coinType) ;

/\*\*

\* Sets the selected snack type

\* @param snackType

\*/

public void setSnackType(ESnackTypes snackType) ;

/\*\*

\* Sets the selected quantity

\* @param selectedQuantity

\*/

public void setSelectedQuantity(int selectedQuantity) ;

/\*\*

\* triggered action event from view component

\* @param ae

\*/

@Override

public void actionPerformed(ActionEvent ae);

### PayForSnackAction class

The PayForSnackAction class extends AbstractAction java abstract class so that it can be set as a listener to a view component, in our case a button, and to catch the action event of that button.

The class contains the reference of the Model object, the type of the selected snack and the quantity of the selected item. It uses PayForSnackWorker class to temporarily instantiate and run a thread for change calculations. This action is safe because the Model class is synchronized and because the PayForSnackWorker brief instantiation runs in a system managed pool of threads.

In the constructor the reference of the model is assigned to its member variable.

Public Methods:

/\*\*

\* Sets the selected snack type

\* @param snackType

\*/

public void setSnackType(ESnackTypes snackType) ;

/\*\*

\* Sets the selected quantity

\* @param selectedQuantity

\*/

public void setSelectedQuantity(int selectedQuantity) ‘

/\*\*

\* triggered action event from view component

\* it instantiates a PayForSnackWorker to process calculations in a separate thread

\* @param ae

\*/

@Override

public void actionPerformed(ActionEvent ae) ;

### PayForSnackWorker class

The PayForSnackWorker class extends SwingWorker java abstract class so that change calculations can be processed in a separate thread.

The class contains the reference of the Model object, the value of the selected quantity and the type of the snack.

In the constructor the reference of the model is assigned to its member variable, also the selected quantity and the type of the chosen snack.

Protected Methods:

/\*\*

\* Processes change calculations on a separate thread and updates the data from the model

\* @return

\* @throws Exception

\*/

@Override

protected Object doInBackground() throws Exception;

### CashInAction class

The CashInAction class extends AbstractAction java abstract class so that it can be set as a listener to a view component, in our case a button, and to catch the action event of that button.

The class contains the reference of the Model object.

In the constructor the reference of the model is assigned to its member variable.

Public Methods:

/\*\*

\* triggered action event from view component

\* @param ae

\*/

@Override

public void actionPerformed(ActionEvent ae);

### Model class

The Model class contains the instantiation of a PropertyChangeSupport class, used for triggering events to UI listeners when a property changes. Another instantiation is for ChangeDisperser class utilised for managing remaining change, and internal change calculations.

The class also has two hash maps used for storing snacks objects and profit per snack. Other member variables are the user’s name and password stored as strings, the total amount of inserted coins, the cash in amount and the buying price of the snacks.

Another thing to mention is that all public methods are stated with the keyword synchronized because of the use of PayForSnackWorker thread and for future threads which will surely access their needed data from the model.

Public Methods:

/\*\*

\*Initialises the object and all it's containing objects

\*/

public synchronized void initialize();

/\*\*

\* Sets the counter for each type of change

\* @param type

\* @param nValue

\*/

public synchronized void transferInsertedChangeToMachine(ECoinTypes type, int nValue);

/\*\*

\* Checks the credentials and fires data for the report page

\* @param user

\* @param pass

\*/

public synchronized void login(String user, String pass);

/\*\*

\* Triggers event for showing error message

\* @param szMessage

\*/

public synchronized void showPopUpErrorMessage(String szMessage);

/\*\*

\* Triggers event for updating the view by snack selection

\* @param type

\*/

public synchronized void syncByType(ESnackTypes type)

/\*\*

\* Gets the price from the snacks hash map by type

\* @param type

\* @return

\*/

public synchronized Double getSnackPriceByType(ESnackTypes type);

/\*\*

\* Gets the quantity of a snack from the snacks hash map by type

\* @param type

\* @return

\*/

public synchronized int getSnackQuantityByType(ESnackTypes type);

/\*\*

\* Sets the quantity of a snack to the snacks hash map by type

\* and triggers an event to update the UI (badge, quantity slider)

\* @param type

\* @param nQuantity

\*/

public synchronized void setSnackQuantityByType(ESnackTypes type, int nQuantity);

/\*\*

\* Gets the value of a particular coin by type from the ChangeDispenser object

\* @param type

\* @return

\*/

public synchronized Double getCoinValueByType(ECoinTypes type);

/\*\*

\* Gets the amount of coins for a particular type from the ChangeDisperser object

\* @param type

\* @return

\*/

public synchronized int getChangeCountByType(ECoinTypes type);

/\*\*

\* Sets the amount of coins for a particular type in the ChangeDisperser object

\* @param type

\* @param nNewValue

\*/

public synchronized void setChangeCountByType(ECoinTypes type, int nNewValue)

/\*\*

\* Gets the total amount of inserted coins for a given transaction

\* @return

\*/

public synchronized Double getdInsertedCoins() ;

/\*\*

\* Sets the total amount of inserted coins for a given transaction

\* @param dInsertedCoins

\*/

public synchronized void setdInsertedCoins(Double dInsertedCoins);

/\*\*

\* Gets the cash in amount of a particular transaction

\* @return

\*/

public synchronized Double getdCashInAmount();

/\*\*

\* Gets the cash in amount

\* @return

\*/

public synchronized Double getdCashInAmount();

/\*\*

\* Sets the cash in amount

\* @param dCashInAmount

\*/

public synchronized void setdCashInAmount(Double dCashInAmount);

/\*\*

\* Triggers event for reseting the cash in amount label

\* and sets the cash in amount to 0

\*/

public synchronized void resetCashInAmount();

/\*\*

\* Adds a listener to the PropertyChangeSupport object

\* The listener can be any UI component which implements PropertyChangeListener

\* @param listener

\*/

public synchronized void addPropertyChangeListener(PropertyChangeListener listener);

Private Methods:

/\*\*

\* Uses SnacksFactory class to instantiate the snacks and adds allocated objects to the

\* snacks hash map

\*/

private void initializeSnacks();

/\*\*

\* Initialises the profit per hash map

\* snacks hash map

\*/

private void initializeProfitPerSnack();

/\*\*

\* Extracts the profits from the has map and calculates the total

\*/

private Double calculateTotalProfits();

/\*\*

\* Converts the enum snack type to integer

\*/

private int getIndexBySnackType(ESnackTypes type);

/\*\*

\* Converts a integer index to a enum snack type

\*/

private ESnackTypes getSnackTypeByIndex(int index);

### ChangeDisperser class

The ChangeDisperser class is used for storing denomination data and change calculations. It contains two hash maps utilised for storing coin values and the quantity for available change in the automat, two constants for total initial change and maximum number of coins and a array list used for temporary storing the change combinations.

Also, for extracting all the change combinations for a given amount of coins, the class uses a recursion algorithm to populate the array list, and after that selects the best optimal solution with the help of a custom function.

The function’s formula is defined below:

3 4

∑((2 \* Ci) + (M – Ni)) - ∑ 1

i=0 i=0

Ni>0 Ni>0

Where:

* M - represents the initial amount of coins for the 0.05, 0.1, 0.2 and 0.5 denominations. The M variable can only have the value 20 because the £1 denomination is excluded from the first sum to force an optimal solution which includes as many one-pound coins.
* Ni - represents the current number of coins in the snacks machine for a certain type of denomination given by the “i” index. Ni must be bigger than 0 so that the sum can take place for a certain index.
* Ci – represents the current count of coins from the selected change combination for a certain type of denomination given by the “i” index.
* (M – Ni) – represents the subtraction of Ni from M, and if Ni is bigger this means that the result will be lower, and the hole function will be smaller and more optimal.
* The second sum is a counter of all the values, bigger then 0, from Ni and it represents how spread are the denominations of that particular change combination. The bigger the counter will be, the lower the overall result of the optimal function will be. Because of this we force the function to select coins from the change combination which have a better spread.

Public Methods:

/\*\*

\* Extracts the optimal combination of change

\* @param dValue

\* @throws Exception

\*/

public void substractOptimalChange(Double dValue) throws Exception;

/\*\*

\* Initializes the hash maps

\*/

public void initialize();

/\*\*

\* Gets the total initial change

\* @return

\*/

public Double getTotalInitialChange();

/\*\*

\* Gets the total change in the machine

\* @return

\*/

public Double getTotalChange();

/\*\*

\* Gets the coin value by type

\* @param type

\* @return

\*/

public Double getCoinValueByType(ECoinTypes type);

/\*\*

\* Gets the change count by type

\* @param type

\* @return

\*/

public int getChangeCountByType(ECoinTypes type);

/\*\*

\* Sets the change count by type

\* @param type

\* @param nNewValue

\*/

public void setChangeCountByType(ECoinTypes type, int nNewValue);

Private Methods:

/\*\*

\* Gets the optimal combination of change by applying a custom minimisation function

\*/

private int[] getOptimalChange();

/\*\*

\* Gets the optimal combination of change by applying a custom minimisation function

\* @param combination

\* @param type

\* @param coinIndex

\* @param nRemainingTotal

\*/

private void setAllChangeCombinations(int[] combination,

ECoinTypes type,

int coinIndex,

Double nRemainingTotal);

/\*\*

\* Checks if the combination can be subtracted from the current change pool

\* @param combination

\*/

private boolean canSupportChange(int[] combination);

/\*\*

\* Subtracts the optimal change combination from the current coin pool

\* @param combination

\*/

private void setOptimalChange(int[] combination)

/\*\*

\* Initialises the type of coins in the hash map

\*/

private void initializeCoinValuesMap();

/\*\*

\* Initialises the change count in the hash map

\*/

private void initializeChangeCountMap()

/\*\*

\* Gets the index by the coin type

\* @param type

\*/

private int getIndexByCoinType(ECoinTypes type);

/\*\*

\* Gets the coin type by the index

\* @param type

\*/

private ECoinTypes getCoinTypeByIndex(int index);

### SnacksFactory class

The SnacksFactory class is used for returning the address of a snack instance like Crisps, MarsBar, CocaCola, Eugenia and Water, according to their type, as a ISnack pointer. The returned object’s address can be added and managed in a hash table because ISnack interface is a common template for all snack objects.

/\*\*

\* Gets the instances of snack objects by their type

\* @param eSnackType

\*/

static ISnack GetSnack(ESnackTypes eSnackType);

### ISnack interface

The ISnack interface is used as a contract for the Snack abstract class by deriving from it and for using its definition as a common template to store the concrete snack instantiation as collectables into a hash map.

### Snack abstract class

The Snack abstract class implements the ISnack interface and contains member variables like the id, the type of the snack, price, current quantity and initial amount as constant. Implementing the ISnack interface forces the developer to take into consideration the inherited methods and override them.

The constructor initialises its members through its received parameters.

Public Methods:

/\*\*

\* Gets the quantity which was sold

\* @return

\*/

@Override

public int getSoldQuantity() ;

/\*\*

\* Gets the current quantity

\* @return

\*/

@Override

public int getnQuantity() ;

/\*\*

\* Sets the current quantity

\* @param nQuantity

\*/

@Override

public void setnQuantity(int nQuantity);

/\*\*

\* Gets the id

\* @return

\*/

@Override

public long getnId();

/\*\*

\* Gets the type of the snack

\* @return

\*/

@Override

public ESnackTypes geteType();

/\*\*

\* Gets the price

\* @return

\*/

@Override

public Double getdPrice();

### Crisps class

The Crisps class extends the Snack abstract class for code reusability and uses singleton pattern to be instantiated only one time. The constructor is defined as private so that it can only be instantiated from within and sets its super class default variable members. The class only contains the private instance of its own type of object as a static constant.

Public Methods:

/\*\*

\* Gets the singleton instance

\* @return

\*/

public static Crisps getInstance();

### Crisps MarsBar class

The MarsBar class extends the Snack abstract class for code reusability and uses singleton pattern to be instantiated only one time. The constructor is defined as private so that it can only be instantiated from within and sets its super class default variable members. The class only contains the private instance of its own type of object as a static constant.

Public Methods:

/\*\*

\* Gets the singleton instance

\* @return

\*/

public static MarsBar getInstance();

### CocaCola class

The CocaCola class extends the Snack abstract class for code reusability and uses singleton pattern to be instantiated only one time. The constructor is defined as private so that it can only be instantiated from within and sets its super class default variable members. The class only contains the private instance of its own type of object as a static constant.

Public Methods:

/\*\*

\* Gets the singleton instance

\* @return

\*/

public static CocaCola getInstance();

### Eugenia class

The Eugenia class extends the Snack abstract class for code reusability and uses singleton pattern to be instantiated only one time. The constructor is defined as private so that it can only be instantiated from within and sets its super class default variable members. The class only contains the private instance of its own type of object as a static constant.

Public Methods:

/\*\*

\* Gets the singleton instance

\* @return

\*/

public static Eugenia getInstance();

### Water class

The Water class extends the Snack abstract class for code reusability and uses singleton pattern to be instantiated only one time. The constructor is defined as private so that it can only be instantiated from within and sets its super class default variable members. The class only contains the private instance of its own type of object as a static constant.

Public Methods:

/\*\*

\* Gets the singleton instance

\* @return

\*/

public static Water getInstance();

### ESnackTypes enum

The ESnackTypes enum is used for differentiating between snacks

### ECoinTypes enum

The ECoinTypes enum is used for differentiating between coin denominations.

### Defines class

The Defines class contains common constants used all over the project. This optimizes the code reusability by setting one point of modification for common data.

### SyncUpData class

The SyncUpData class is used as a simple structure with public member variables to send it’s data from the model to the main view.

### AdminReport class

The AdminReport class is used as a simple structure with public member variables to send it’s data from the model to the report page.

# Test Documentation

|  |  |  |  |
| --- | --- | --- | --- |
| **Description** | **Input** | **Output** | **Result** |
| Start Application | Double click the compiled jar file or run the project from NetBeans | The application starts with the UI state from Figure 5, with no snack selected and no vending machine. | Successful |
| Press the Crisps button | Click the Crisps button | As in Figure 6 the button is highlighted with a green line and a green badge which contains the quantity of the snack. The two informational labels are replaced with a vending machine which presents a bigger picture of the snack. If there is no snacks available the quantity slider will be set to 0. | Successful |
| Press the Mars Bars button | Click the Mars Bars button | As in Figure 7 the button is highlighted with a green line and a green badge which contains the quantity of the snack. The two informational labels are replaced with a vending machine which presents a bigger picture of the snack. If there is no snacks available the quantity slider will be set to 0. | Successful |
| Press the Coca Cola button | Click the Coca Cola button | As in Figure 8 the button is highlighted with a green line and a green badge which contains the quantity of the snack. The two informational labels are replaced with a vending machine which presents a bigger picture of the snack. If there is no snacks available the quantity slider will be set to 0. | Successful |
| Press the Eugenia button | Click the Eugenia button | As in Figure 9 the button is highlighted with a green line and a green badge which contains the quantity of the snack. The two informational labels are replaced with a vending machine which presents a bigger picture of the snack. If there is no snacks available the quantity slider will be set to 0. |  |
| Press the Water button | Click the Water button | As in Figure 10 the button is highlighted with a green line and a green badge which contains the quantity of the snack. The two informational labels are replaced with a vending machine which presents a bigger picture of the snack. If there is no snacks available the quantity slider will be set to 0. | Successful |
| Unselect the Crisps button | Click the Crips button | The Crisps button deselects and the main view shows as in Figure 5. | Successful |
| Unselect the Mars Bar button | Click the Mars Bar button | The Mars Bar button deselects and the main view shows as in Figure 5. | Successful |
| Unselect the Coca Cola button | Click the Coca Cola button | The Coca Cola button deselects and the main view shows as in Figure 5. | Successful |
| Unselect the Eugenia button | Click the Eugenia button | The Eugenia button deselects and the main view shows as in Figure 5. | Successful |
| Unselect the Water button | Click the Water button | The Water button deselects and the main view shows as in Figure 5. | Successful |
| Selecting quantity of a snack | Move the quantity slider right or left | While moving the slider left or right, the quantity of the snack is shown below. If the snack is out of stock the slider won’t have any values set and the quantity label will be set to 0. The maximum selection value, of the quantity slider, is set automatically accordingly to the current amount of snacks. | Successful |
| Selecting coin denomination to insert into the snack machine | Move the denominations slider up or down | While moving the slider up or down the coin image representation will change accordingly to the selected coin value. The slider only has a maximum of 5 values to choose from: £0.05, £0.1, £0.2, £0.5 and £1. | Successful |
| Inserting a coin in the vending machine | Press the insert button | Pressing the insert button will add the coin value to the existing total of the entered coins. As shown in Figure 13 the inserted coins label from the top right of the main view will represent the new total of added coins. If the inserted coins value is bigger than the actual total value of the snack’s selected quantity an error will pop up, as in Figure 21, stopping the coin insertion action. | Successful |
| Paying for the selected snack | Press the pay button | As represented in Figure 14, pressing the pay button will reset the inserted coins label to 0, and will set the cash in label according to the change calculations. Paying with an incomplete amount of inserted coins a popup error message will appear as in Figure 20. If there is no quantity selected and the snack is not out of stock an error message will show as in Figure 18. If there is no quantity selected and the snack is out of stock an error message will show as in Figure 19. If there is no change available for a selection, an error message will appear as in Figure 23, the inserted coins label will be reset to 0, the quantity will remain the same and the coins will be returned and displayed in the cash in label. | Successful |
| Cash in the remaining coins | Press the cash in button | Pressing the cash in button resets the cash in label to 0 as seen in Figure 15. | Successful |
| Pressing the sign in link | Press the sign in label | Pressing the sign in label will show a modular dialog requesting credentials to login to the report page. Every time the dialog is opened the username and password text fields are reset. | Successful |
| Entering credentials to login into the report page and logging in | Press the login button from the admin page. | As shown in Figure 16 credentials are requested for advancing to the report page. The default username is “10976” and the password is “1234”, but will still sign in if the text fields will be left empty. If the above text input cases don’t pass the credentials check, a popup error message will appear as in Figure 22. If the username and password are correct the report page will show up with the profit, losses and total informational data. | Successful |
| Closing the login page | Press the close button from the top right corner of the dialog | Pressing the close button of the admin page will reset the page to its initial status and close the dialog. | Successful |
| Closing the report page | Press the close button from the top right corner of the dialog | Pressing the close button of the report page will reset the admin page to its initial status and close the dialog. | Successful |
| Closing the main window | Press the close button from the top right corner of the dialog | Pressing the close button of the main view will release the allocated memory and exit the application. | Successful |
| Testing data correctness from the report page | Selling 10 bottles of water at a reselling price of £0.85, knowing that the original price is £0.45 and that the initial total amount of coins is equal to £27. | On the report page the Profit is £4.0 and the Total amount of inserted coins is £35.5. | Successful |
| Testing data correctness from the report page | Selling 10 bottles of water and 10 Eugenia’s at a reselling price of £0.85, £0.5 respectively, knowing that the original price of both snacks is £0.45 and that the initial total amount of coins is equal to £27. | On the report page the Profit is £4.5 and the Total amount of inserted coins is £40.5. | Successful |
| Testing if the change, from the cash in label, is correctly displayed after the snacks had been paid for | Selling 10 bottles of water at a reselling price of £0.85 with nine coins of £1. | In the cash in label the £0.5 value is displayed. | Successful |
| Testing if the last uncashed coins value adds to the current change transaction. | First Step: Buy 10 bottles of water at a reselling price of £0.85, knowing that the total amount of the inserted coins is £9, sets the value £0.5 in the cash in label.  Second Step: Buy two mars bars at the reselling price of £0.7 with two £1 coins | In the cash in label the £1.1 value is displayed. | Successful |
| Testing if the snack machine shows an error and invalidates the transaction if there is no change for a given selection. | Buy with £1 coin a bottle of water. Repeat this until the water is out of stock. | At some point, depending of what has been bought until then, the machine will display an error message, as in Figure 23, will return the coins in the cash in label and will show the same stock before the transaction, refusing the pay. | Successful |
| Testing credentials with the username “10976” and password “1234” | Press Sign In button from the main page, type the required credentials and press the Login button. | A report page appears with no error message. | Successful |
| Testing credentials with empty text fields | Press Sign In button from the main page, leave the credential text fields empty and press the Login button. | A report page appears with no error message. | Successful |
| Testing credentials with any username input except empty text field or “10976” and any password except empty text field or “1234”. | Press Sign In button from the main page, type any credentials except the required ones and press the Login button. | A popup error message appears as in Figure 22 denying the access to the report page. | Successful |

# Appendix A: Source Code

The TenThousandsSteps application was created as a simple Java Application Project with the NetBeans IDE version 8.2. Besides the embedded Swing Framework there were no other external packages or libraries used or integrated into the project.

Notable to mark is that accessing the report page can be bypassed for a faster testing experience by logging in with the credential text fields empty. The username “10976” and password “1234” can be entered to test if the credential check is done.



# Appendix B: References

* Swing Framework Documentation:

<https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html>

* Java Tutorial:

<https://docs.oracle.com/javase/tutorial/java/index.html>

* IBM’s Class Diagram Tutorial:

<https://www.ibm.com/developerworks/rational/library/content/RationalEdge/sep04/bell/index.html>

* Design Patterns Tutorial:

<https://www.oodesign.com/>