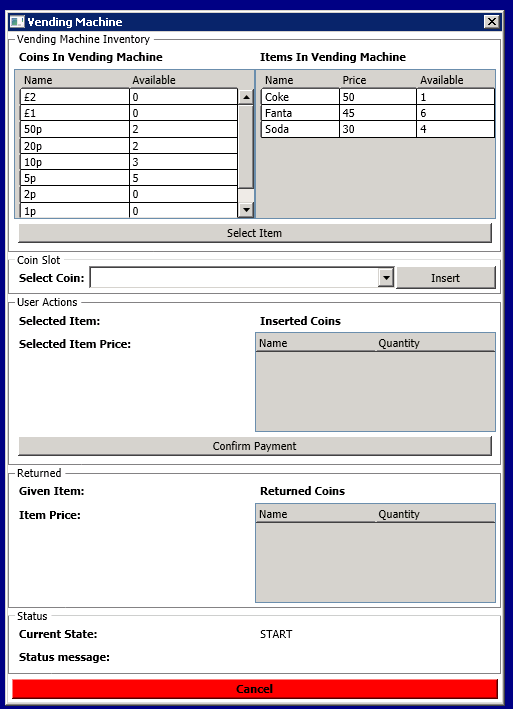
# Vending Machine Simulator

## NuGet Packages

Please enable “Restore NuGet Packages” before building the solution. I have deleted the Autofac and Moq dlls that are downloaded configured using NuGet.

## User Interface

Vending machine simulator is a standalone windows WPF (MVVM pattern) application developed using Visual Studio 2010.



It is a simple user interface with buttons representing the actions that can be performed on the vending machine.

Possible actions are:

1. Select Item
2. Insert Coin
3. Confirm Payment
4. Cancel

## Design

The application is designed using state pattern where in each state of the vending machine permits certain actions. When an action is triggered that is not permitted in that state, an error message is displayed to the user.

Possible States are:

1. Start: This is the state of the vending machine at start up.
2. Accepting Coins: This is the state when the vending machine is accepting coins.
3. Payment Complete: This is the state when the user has inserted required amount of coins for the item chosen.



Figure 1State Diagram

An abstract class called *State* is defined which has virtual methods for all the actions that are supported by the vending machine.

public abstract class State

    {

        public abstract string StateName { get; }

        public virtual State SelectItem(IMainWindowViewModel viewModel)

        {

            throw new NotSupportedException(string.Format("Cannot select item in '{0}' state", this.StateName));

        }

        public virtual State InsertCoin(IMainWindowViewModel viewModel)

        {

            throw new NotSupportedException(string.Format("Cannot insert coins in '{0}' state", this.StateName));

        }

        public virtual State Cancel(IMainWindowViewModel viewModel)

        {

            throw new NotSupportedException(string.Format("Cannot cancel in '{0}' state", this.StateName));

        }

        public virtual State ConfirmPayment(IMainWindowViewModel viewModel)

        {

            throw new NotSupportedException(string.Format("Cannot confirm payment in '{0}' state", this.StateName));

        }

        public override string ToString()

        {

            return this.StateName;

        }

    }

Every action throws a NotSupportedException hence concrete state class has to only worry about implementing the actions it is permitting.

The class also has an abstract get property that each concrete state has to implement in order to provide any description about itself. In our case, we use it for state name.

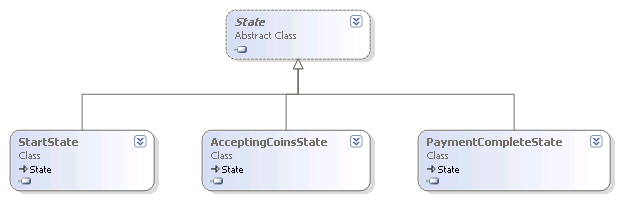


Figure 2 State class diagram

The MainWindowViewModel is bound to the MainWindow and uses the WPF command binding to trigger actions. The view model holds the current state of the vending machine. The commands that are bound to the buttons on the UI call the appropriate method on the CurrentState object of the view model.

e.g. SelectItemCommand

public class SelectItemCommand : ICommand

    {

        public event EventHandler CanExecuteChanged;

        public void Execute(object parameter)

        {

            var viewModel = (IMainWindowViewModel)parameter;

            try

            {

                viewModel.CurrentState = viewModel.CurrentState.SelectItem(viewModel);

            }

            catch (NotSupportedException ex)

            {

                viewModel.DisplayMessage(ex.Message);

            }

        }

        public bool CanExecute(object parameter)

        {

            return true;

        }

    }

Concrete state classes implement the actions that it supports and updates the view model state appropriately. The data binding deals with auto updating of the user interface.

e.g. StartState

public class StartState : State

    {

        public override string StateName

        {

            get

            {

                return "START";

            }

        }

        public AcceptingCoinsState AcceptingCoinsState { get; set; }

         public override State SelectItem(IMainWindowViewModel viewModel)

         {

             if (viewModel.SelectedItemStock == null)

             {

                 viewModel.DisplayMessage("Please select item from stock");

                 return this;

             }

             // We need to reset the coins returned/inserted coins and item given data

             viewModel.ReturnedCoins.Clear();

             viewModel.InsertedCoinStock.Clear();

             viewModel.GivenItem = null;

             viewModel.SelectedCoinStock = null;

             viewModel.SelectedItem = viewModel.SelectedItemStock.Item;

             viewModel.StatusMessage = "Waiting for coins";

             return this.AcceptingCoinsState;

         }

    }

Data classes are defined in the VendingMachine.Data project. Few extension methods are defined in the DataExtensions class.

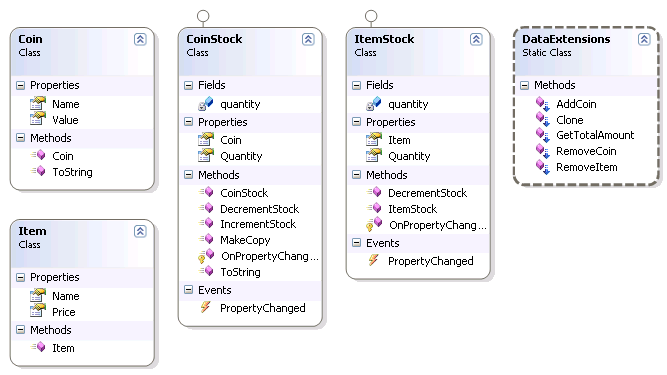


Figure 3 Data class diagram

A set of utility classes are defined in VendingMachine.Utilities project. The most important of these classes is the ChangeCalculator class. This class implements a highest denomination first approach to choose the change.

An open source IoC library, Autofac is used for dependency injection. We use property injection in most of the scenarios but it can be easily be restricted to construction injection only. Each project in the solution has a class called AutofacModule which registers its classes with the IoC container. The registration is set to PropertiesAutowired which can be removed to restrict to constructor injection only. However, this means we will have to change all the classes to have a constructor with the required dependencies getting injected.

## Unit Tests

Unit tests are written for few classes. Another open source mocking library called Moq is used for mocking dependencies. Unit tests are placed under Tests folder of the solution.