A1: Implementation to Specification

You are given a specification and a partially working piece of (stubbed) code that represents a vending machine (the kind that vends pop). Your job is to write the code to fulfill the test specification with (likely) only a partial understanding of the system as a whole.

Learning Objectives

By the end of this assignment, you will:

- Implement a small to medium-scale software system conforming to a design.
- Overcome the initial fear of opening up a code base that is likely too difficult to understand in its entirety.
- Practice development and debugging in the Visual Studio IDE.
- Practice development with C#.

Deliverable / Submission

You will deliver a .zip file containing your source code into the Dropbox in D2L.

1. From the project directory, remove the bin\ and obj\ folders
2. Zip up the solution directory, ensuring that your source files and test scripts are included (if you are including test scripts, include these at test-scripts\)
3. Upload the zip file into the Dropbox in D2L by the deadline

We will extract your code and run it against a small battery of tests (some should pass, some should fail). These tests are designed against the specification below.

Steps

1. Clone the repo from the GitHub site
2. Read through the documentation below
3. Modify only VendingMachineFactory.cs and/or add new classes.
4. Zip it up, and submit via D2L.

Description

You are part of a team that is developing a vending machine -- the kind you put money into, and get pop out of. Your team is developing the vending machine hardware and software. For the purpose of this class, we are simulating this entirely with software.

The idea with this new class of vending machines is that they are deployed centrally. That is, you create them with a virtual "factory", and then you can configure them on the fly with different kinds of pops, or charge different amounts of pops on a per vending machine basis.

Your boss has already written code that simulates the *hardware*. This is represented by the set of files with the Frontend1 namespace. In particular, you'll notice an interface called IVendingMachineFactory -- this represents the interface that you will need to build a concrete class for. This class (by implementing the interface) is responsible for managing the

vending machines: creating them, configuring them, as well as dealing with what happens when things are loaded (coins, pops), or when pops are vended, and so forth.
Your job is to implement the class that implements the interface. Your boss has already stubbed out the file VendingMachineFactory.cs to get you going. You can add as many other classes as you see fit; however, you should not modify any of the other pieces of code that are already there.

Specification for the Vending Machine

The vending machine factory manages and controls several vending machines. You should be able to create as many of them as you like (we are just simulating them right now, after all).
A vending machine, when it is created, is created by specifying two things: (1) what kinds of coins it accepts (e.g. 1 cent coins, 10 cent coins, 25 cent coints, or even made up ones, like 32 cent coin), and (2) the number of buttons that it has on its face (for people to select pops from). There are a few restrictions: each coin value must be non-zero and non-negative, and the coin values must be unique.
The vending machine can then be configured to vend different kinds of pops (specified by name, e.g. "Coke"), as well as a price. Here, the names need to be an actual string, and the prices need to be greater than zero.
At this point, your vending machine is empty. You can load coins into the vending machine, and you can load pops into the vending machine. Our vending machine is not clever enough to detect when you load the wrong pop in, or when you load the wrong coins in. (This is a software defect, perhaps, but it also mimics real life, where a technician might put things into the wrong place.)
Now you can put in some coins, and press a button to try to vend a pop. The usual things happen: if you put in enough money, the pop will dispense; if you didn't have enough money, nothing gets vended; if there were no more pops of that kind, nothing gets vended. If you put in too much money, the machine will try to give you change based on what it has been loaded with (but if it doesn't have enough change, it will pocket the difference). If you put in money that it doesn't recognize (based on the kinds of coins it knows), the money will come right out the chute.
This vending machine has a few quirks:

1. The delivery chute is the same for both the money (i.e. change) and pop; also, it needs to be explicitly extracted from the vending machine (otherwise, the door stays shut, and everything stays inside).
2. Money inserted into the vending machine is not automatically "put into circulation" -- it's put into a "money we made" pile, and cannot be used to make change for later transactions.
3. Oh, and there's no "return change" on this vending machine (say if it didn't have your pop). Muahahaha.

Finally, you can unload everything from the vending machine when you want to tear it all down. This will give you all the unsold pops back, as well as the money that has been inserted, and the remaining change.

Specification for the Test Script Commands

Your boss has implemented a set of tests that you can use to test your implementation as you go. These are included in the tests/ folder, where some scripts are expected to pass, while others are expected to fail. You can run these tests by running ScriptProcessor.Main, and you can build your own test scripts.
The test scripts themselves are based on a "programming language", if you will. The way to think about it is to imagine that this script is something that a robot would go do -- to create a vending machine, to load it up, insert coins, vend a pop, etc. Each script has a number of "testing" instances--these are represented by the CHECK-DELIVERY and CHECK-TEARDOWN functions. CHECK-DELIVERY is for comparing what is in the delivery chute for the vending machine against what you expect -- if there is a match, the test will pass (otherwise, it will fail). For instance, suppose you put in $1 for a $0.50 Coke; you would expect the delivery chute to contain your Coke, and $0.50 in change (assuming the vending machine had enough money to vend the money). Similarly the CHECK-TEARDOWN function is to check how much change is left, how much money has been collected, and which unsold pops are left.
[value-of-change-remaining] ; [value-of-money-collected]
See the test script syntax.

Hints

This is a lot to look at all at once. Make sure that you can settle down and concentrate for an extended period of time as you read through the documentation.

1. Take notes based on the requirements and specification above. Make your own shorthand that is easier to interpret for you (rather than having to read the whole text).
2. Look at the test scripts in test/. Read through the to make sense of how they ought to behave if the vending machine factory (and vending machine) is working properly.
3. Write your own test scripts.
4. There's a lot of code here. It might be useful to take a look at the Frontend1 namespace code, but you shouldn't have to worry about the Frontend1.Parser namespace code.
5. Prepare to take multiple "passes" on writing a solution. For your first pass, take the VendingMachineFactory class, and simply write out to the console what is happening (i.e. use Console.WriteLine). You'll see how these functions are being called in relation to the scripts that are being processed. Once you understand that, *then* modify VendingMachineFactory to start to make it work.
6. Read the function signatures (and documentation) on the IVendingMachineFactory interface carefully. You will need to get these right for the integrating piece of code (SENG301VMAnalyzer) to be able to use your VendingMachineFactory properly. Take particular note of unloadVendingMachine() and extractFromDeliveryChute() which are a bit tricky.
7. It will be useful to understand how generic collections work (here's a short tutorial). In particular, look into List<T> and Dictionary<TKey, TValue>. These will provide

strongly typed collections with very easy manipulation methods. You'll note that unloadVendingMachine() returns an List<IList>, which contains three different lists (two for coins, and another for pops)--this is essentially a hack to return a 3-tuple (since typed lists all inheret from IList).

8. Work on the "normal" cases first before worrying about edge cases. (Though, ultimately, make sure you adhere to the specification above!)

Grading Rubric

You work will be run against a set of test scripts (much like the ones you have in test/) that are designed from the requirements laid out above. The more of these your code passes, the higher your grade.
You may be awarded some discretionary points for submitting interesting test scripts of your own (e.g. that consider boundary conditions).
Your code will not be assessed on its internal structure; however, we can provide verbal feedback on this if you ask.

No Late Assignments

Your assignment is due at noon on the specified due date. No late assignments will be accepted.
