Module 1 Capstone Success

Success Criteria

Does it work?

- Does it meet all requirements?
- Do all the requirements work as described?
- Are you able to add money?
- Are you able to make purchases?
- What happens if the user tries to buy a product that doesn't exist?
- What happens when the user runs out of money?
- What happens when a slot runs out of a product?
- Does it keep running for multiple "sessions"

Is the code clean? Is it understandable?

- Good variable, method, and class names
- Appropriate use of objects
- Appropriate use of methods
- Good formatting and indenting

Is it well structured?

- Appropriate use of the OO principles: Encapsulation, Inheritance, Polymorphism (especially Encapsulation)
- Data exists in one place (not multiple places)
- Functionality exists in one place (not multiple places)
- Methods "return", don't call other methods in a circular manner

Is it testable?

- Methods do their work based on parameters and return a value
- Minimum number of class-level variables
- ALL Console.ReadLine and Console.WriteLine statements isolated in one (un-testable) module.

What is not important?

- Clever code
- Additional features, no matter how amazing

Code Review

Good code matters! So the project will also be judged not only on the functionality but on the quality of the code. The code must also “pass” code review. During a code review, the code is examined for the following

Code Review Criteria

- Can you explain the code, including your partners?
- Does it align to OOP?
- Does it make good uses of classes and methods?
- Does it make good use of the language?
- Does it have any errors that would cause it to eventually fail?

Common Mistakes to Avoid

- Having Console.ReadLine or Console.WriteLine statements in more than 1 module
- Re-reading the inventory each time the items are accessed
- Not handling exceptions or having empty catch blocks
- Not having separate classes for File I/O
- After the transaction has finished, the balance of the machine is not emptied
- Creating multiple instances of classes that should have 1, like the VendingMachine, Menu, File IO, etc.
- Using a GOTO approach with menus that would eventually result in a Stack Overflow condition
- Writing everything in VendingMachineCLI
- Having functionality repeated in multiple places
- The vending machine quits after each transaction
- Writing code that is not unit testable
- Saving Unit Tests for the end
- No Unit Tests
- Trying to write everything at one time instead of dividing the functionality
- Not pairing
- Not communicating, especially concerning expectations and availability over the weekend
- Not asking questions when you are stuck or don’t understand how to do what you need to do