Comp 3300 – Fall 2024
Project 1, 100pts
Due: Sunday October 13th @ 11:55PM

Project 1 Instructions

Objectives:

Apply the application development principles and techniques from the first half of the semester to implement a small software application in C#.

Requirements:

Read through all the requirements before you begin.

Any project that will not compile will receive a zero.

Usage of GIT is required. It will not be mentioned within the instructions, however it is expected that there will be multiple commits at different stages of your development process. There is no specification as to the number of required commits, you are free to commit as you see fit. However, I will be looking at the commit history and comments to decide if GIT is being leveraged as intended. Do not simply make a start and end commit as that isn't utilizing GIT as it is intended. Not using GIT is considered a major issue and will significantly impact the final grade.

Background:

You will assume the role of a developer that is prototyping a simplified car lot management program. This prototype will allow a prospective car dealer the ability to manage a car lot inventory and for a single shopper to purchase a car.

Comp 3300 – Fall 2024
Project 1, 100pts
Due: Sunday October 13th @ 11:55PM

Instructions

- Since this is a project that applies what has been covered in the first half of the semester as well as in the prerequisite courses, specific implementation details may not be provided for all requirements as the goal of a project is to demonstrate that you can apply what you have previously learned.

- If there are concepts that have not been covered in C#, but were covered in the prerequisite courses, you may need to leverage the Microsoft documentation to find how to implement it within C#.

- I will not answer any detail/implementation questions - only high-level clarification questions for instructions – asked prior to 10/11/2024 @ 5:00PM. After that no questions will be answered regarding the project.

- The entire project must be done individually without consultation with any other individual other than the instructor. The TA's/CsX lab will **NOT** be able to help at all for any code related questions.

- The instructions will not be as detailed as they have been with the weekly assignments – at this point it is expected that you are able to utilize terminology to determine the appropriate code piece to use to meet the requirements. Unless otherwise specified – you are free to use anything we've covered up to this point in the semester to complete the code. Make sure you read the instructions in their entirety BEFORE writing any code.

- Make sure you are applying good OO concepts here. Things like proper spacing, meaningful names, etc.… are important and will be looked at. There should NOT be any ReSharper warnings present with the final submission.

Grading

There is one (1) part for this project with a total point value of 100. A 100 requires no issues with convention, code that is complete, and functions as expected without issues. **Any submission that doesn't compile due to errors is an automatic 0, no exceptions.**

As always here are some tips to help make the process a bit easier:

- **Start early!!**
- Take breaks – development isn't something you can sit there and power through. You'll hit a point where you're just repeating the same mistakes. It's ok to walk away and take 30 min, an hour, or whatever time needed to refresh your brain.
- I cannot stress this enough…*double check your work after submitting* – sometimes you think something is submitted correctly but then find out it's not. After uploading the final version, download it and re-import it to Visual Studio as a new project.
- There will be no instructions about method/property descriptions – it is expected that you add them in.
- Implement the three specified classes first and write some code to verify each class works correctly before moving onto implementing the UI.
- When implementing the UI, implement and test the easy items first and then progress to the more difficult options.
- You can implement the entire application using the properties and methods provided in the specifications for three model classes: CarLot, Car, and Shopper. If you need to add other methods to these classes, you may do so, but do so with a good reason.

Comp 3300 – Fall 2024
Project 1, 100pts
Due: Sunday October 13th @ 11:55PM

Setup:

1) Create a C# Windows Forms application as follows:
   a) Project Name: *FirstNameLastNameProject1*
   b) Make sure there is a GIT repository attached to the solution/project.
      i) There should be an initial commit at this point, with a descriptive comment.

Model: Car Class:

2) Create a new class called Car that is part of the Model namespace that meets the following requirements:

   a) Declares the following properties:

      i) Make
         (1) This will store the name of the car maker.
      ii) Model
         (1) This will store the name of the car model.
      iii) Mpg
         (1) This will store the number of miles per gallon the car gets in the form of xx.xx.
      iv) Price
         (1) This will store the numeric price of the car in the form of xx.xx.

   b) Add the following constructor:

      i) A public four-parameter constructor that accepts values to initialize the car's make, model, mpg, and price properties.

         (1) Enforce appropriate preconditions.

Model: CarLot Class:

3) Create a new class called CarLot that is part of the model namespace that meets the following requirements:

   a) Contains the following private data member:
      i) Inventory: List of Cars

   b) Declares a public constant that declares a tax rate of 7.8%.

   c) Add a default constructor that does the following:

      i) Instantiates the Inventory data member.
      ii) Invokes the StockLotWithDefaultInventory method (requirements below).

   d) Add the following private helper method(s):

      i) StockLotWithDefaultInventory which adds four new Cars to the inventory. The four Cars to add are:

         (1) Ford, Focus ST, 28.3, $26,298.98
         (2) Chevrolet, Camaro ZL1, 19, $65,401.23
         (3) Honda, Accord Sedan EX, 30.2 $26,780
         (4) Lexus, ES 350, 24.1 $42,101.10

   e) Add the following public methods:

      i) FindCarsByMake that returns a List of Car object and does the following:

         (1) Has a string parameter that indicates the make of the cars to search for within the inventory.
         (2) Searches through the inventory and returns all the Car objects that matches the specified make, if one or more are found then the List is returned, otherwise null is returned.
         (3) The search should be case insensitive.

ii) FindCarByMakeModel that returns a Car object and does the following:

(1) Has two string parameters that indicates the make and model of the car to search for within the inventory.
(2) Searches through the inventory and returns the first Car object that matches the specified make and model in the inventory, otherwise null is returned.
(3) The search should be case insensitive.

iii) PurchaseCar that returns a Car object and does the following:

(1) Has two string parameters that indicates the make and model of the car to purchase within the inventory.
(2) Checks if the specified car is in the inventory, if so, it removes the car from the inventory and returns the Car object. If the car was not found in the inventory, the method returns null.
(3) The search should be case insensitive.

iv) AddCar that does the following:

(1) Has the following parameters: make, model, mpg, price.
(2) Creates a new Car object based on the parameter values and adds the Car object to the inventory.

v) GetTotalCostOfPurchase does the following:
(1) Accepts a Car object and then returns the total cost of the purchase based on the price of the car and the tax rate.

vi) FindLeastExpensiveCar that returns a Car object and does the following:

(1) Iterates through the inventory and finds the cheapest car and returns that Car object.
(2) If there are not any cars in the inventory, this method should return null.

vii) FindMostExpensiveCar that returns a Car object and does the following:

(1) Iterates through the inventory and finds the most expensive car and returns that Car object.
(2) If there are not any cars in the inventory, this method should return null.

viii) FindBestMPGCar that returns a Car object and does the following:

(1) Iterates through the inventory and finds the car with the best MPG and returns that Car object.
(2) If there are not any cars in the inventory, this method should return null.

ix) FindWorstMPG that returns a Car object and does the following:

(1) Iterates through the inventory and finds the car with the worst MPG and returns that Car object.
(2) If there are not any cars in the inventory, this method should return null.

x) Count expression-bodied property that returns the total number of cars within the inventory.

xi) Inventory – expression-bodied property that returns the List of the store's inventory.

xii) For all the above elements, enforce appropriate preconditions.

Model: Shopper Class:

4) Create a new class called Shopper that meets the following specifications:

a) Declares the following properties:

i) Name
(1) This will store the name of the shopper.

ii) MoneyAvailable
(1) This will store the amount of money currently available for the shopper to purchase cars with, in the form of xx.xx.

b) Add the following private data member:

i) Cars : List of Cars

c) Add the following constructor:

   i) A public two-parameter constructor that accepts values to set the shopper's Name and MoneyAvailable property. This constructor should also instantiate the list of cars to a default value.

d) Add the following public methods:
   i) CanPurchase that returns a boolean and does the following:

      (1) It takes a parameter which is a Car object. This method checks to see if the shopper has enough money to purchase a car of that price including tax. If the shopper has enough money, the method returns true, otherwise false.

   ii) PurchaseCar that does the following:

      (1) Has one parameter, a Car object, and the car to the list of cars for the shopper.

      (2) It calculates and sets the amount of money the shopper has left after purchasing the car.

   iii) For all the above elements, enforce preconditions as appropriate.

View: Create a UI that provides basic CarLot functionality:

Once the above classes are implemented add functionality to use and interact with these classes to create a car lot and a shopper and allow the shopper to view and purchase a car. The class(es) and methods needed for this UI functionality are up to you to, but at least one of the classes must be the CarLotForm.

Note: Not every method or property implemented in the model classes above may be used to meet the requirements below.

Hint: *Make sure you consider how each of the classes we created previously interact with each other if this was a real-world implementation. I.E., a "Shopper" might not need to know (or care about) details about what a CarLot is/can do, but they (Shopper) do care about a Car.*

5) When the CarLotForm is created it should load the car lot with the default cars.

6) Add menu item to trigger a new dialog that will allow the addition of a new car to the current inventory of cars. This will include asking the details about the new car and properly validating and saving it.

7) Add a button to the form that will display a dialog that will allow the user to enter the name and amount of money for a Shopper. The Shopper should be able to then be returned from the dialog.

a) Upon creating a Shopper, the shoppers name and amount of money available should be displayed on the main form of the application.

8) All money should be displayed in standard currency format, e.g., $15,124.56.

9) The main form should display a listbox of the inventory of cars, the display in the list box should be as follows:

Make Model Price MPG

MPG should be displayed to one decimal place.

Money in currency format.

Example display string:

Ford Focus $14,123.32 23.2mpg

10) Add functionality to allow the shopper to purchase the selected car form the inventory.

a) If the shopper doesn't have enough money to purchase the selected car, then an appropriate message should be displayed.

b) If the shopper can purchase the car, a congrats message should be displayed displaying details about the car that was purchased. Additionally, the amount of money the shopper has left after the purchase should be updated in the form.

c) A shopper will be allowed to purchase additional cars if they so choose, but each purchase will just be a single car.

11) The form should have a button and menu option that will allow the detailed viewing of the inventory like the example format below. How you display this is up to you. However, consider the idea we've explored about making the UI as user friendly as possible – potentially this could be a lot of information so it may NOT make sense to include the display directly on the main form.

```
Inventory of 4 cars.
Ford Focus ST            $26,298.98  28.3mpg
Chevrolet Camaro ZL1     $65,401.23  19.0mpg
Honda Accord Sedan EX    $26,780.00  30.2mpg
Lexus ES 350             $42,101.10  24.1mpg

Most expensive:
Chevrolet Camaro ZL1     $65,401.23  19.0mpg

Least expensive:
Ford Focus ST            $26,298.98  28.3mpg

Best MPG:
Honda Accord Sedan EX    $26,780.00  30.2mpg

Worst MPG:
Chevrolet Camaro ZL1     $65,401.23  19.0mpg
```

File – Include a .txt file with information:

Add a notes.txt file to the project and note any known bugs or missing functionality in the project. If there are no known issues, then please indicate so in this file.

Extra Credit:

The following items are optional and will only be reviewed/accepted if the main requirements listed above are complete. If the program is missing major elements from above, the extra credit will NOT be applied/reviewed. If both extra credit items are attempted and completed, up to 5 extra credit points will be applied to the final grade of this project. These points cannot make the final grade exceed 100 points. The extra credit points are all or nothing for each (2.5 points per item). To be clear – the points possible for extra credit are: 0, 2.5, or 5.

You should indicate within the notes.txt file if you have attempted the extra credit and which (or both) one was attempted. If you do not indicate within the .txt what was attempted, they will not be reviewed.

1) Add functionality to prompt the shopper for a specific make and then to populate the listbox with all the cars by a specific make. If there are not any of that make found, then an appropriate message should be displayed. The shopper should still be able to select and purchase a car. Add functionality so that the entire CarLot inventory can be displayed in the listbox again, functionally "resetting" the display of cars visible.

2) Add functionality to so that the list of cars the Shopper has purchased can be seen in a listbox.

Submitting the Project:

Make sure you clean the solution and zip up the project/solution files. Make sure the entire folder structure is included (this means there should also be a .git folder if you are applying Git correctly). The easiest way to do this is to make sure you navigate to the highest-level folder in explorer that only contains the relevant files/folders for this project/solution.

The file name should be: *FirstNameLastNameProject1*.zip (for example, JohnDoeProject1.zip).

Submit the Zip file to the appropriate link in Moodle.