Fall 2017 CIS605 Semester-Long Project

Filling Station Sales System

Overview

This is a semester-long project, with parts of the project due at various points during the term. The code and user-interface design will be looked at closely in addition to checking for proper functioning of the running system.

The four deliverables, their due dates, and their brief deliverables are summarized in the table below:

	Deliverable	Due	Points
1	User Interface	September 17	25
2	Class Design + Basic Code + Input Validation	October 8	50
3	Custom Events + hard coded test data	November 12	75
4	Full System with Arrays + File I/O	December 7	100
			250

For this project you will be developing a system in Visual Basic .Net, using Visual Studio, which implements some of the functionality that might exist within the retail industry, specifically the operation of gasoline/filling station. Some functionality and data elements have been simplified in order to adequately scope the project for completion during the semester. As the semester progresses, you will be given more details on the project.

The basic function of this system is to track sales of different products, loyalty customers, and their transactions. By the fourth project submission, you will have a completed working system that manages many logistics of the application.

The following use case diagram summarizes the various actors that may use the system. You do NOT need to create separate applications for each. A real application would be very concerned about privacy and security. HOWEVER, to simplify this project, you do NOT need to worry about user access or privileges. You should assume that all actors will use the same program and that each actor will only use the parts of the UI that is applicable to them.

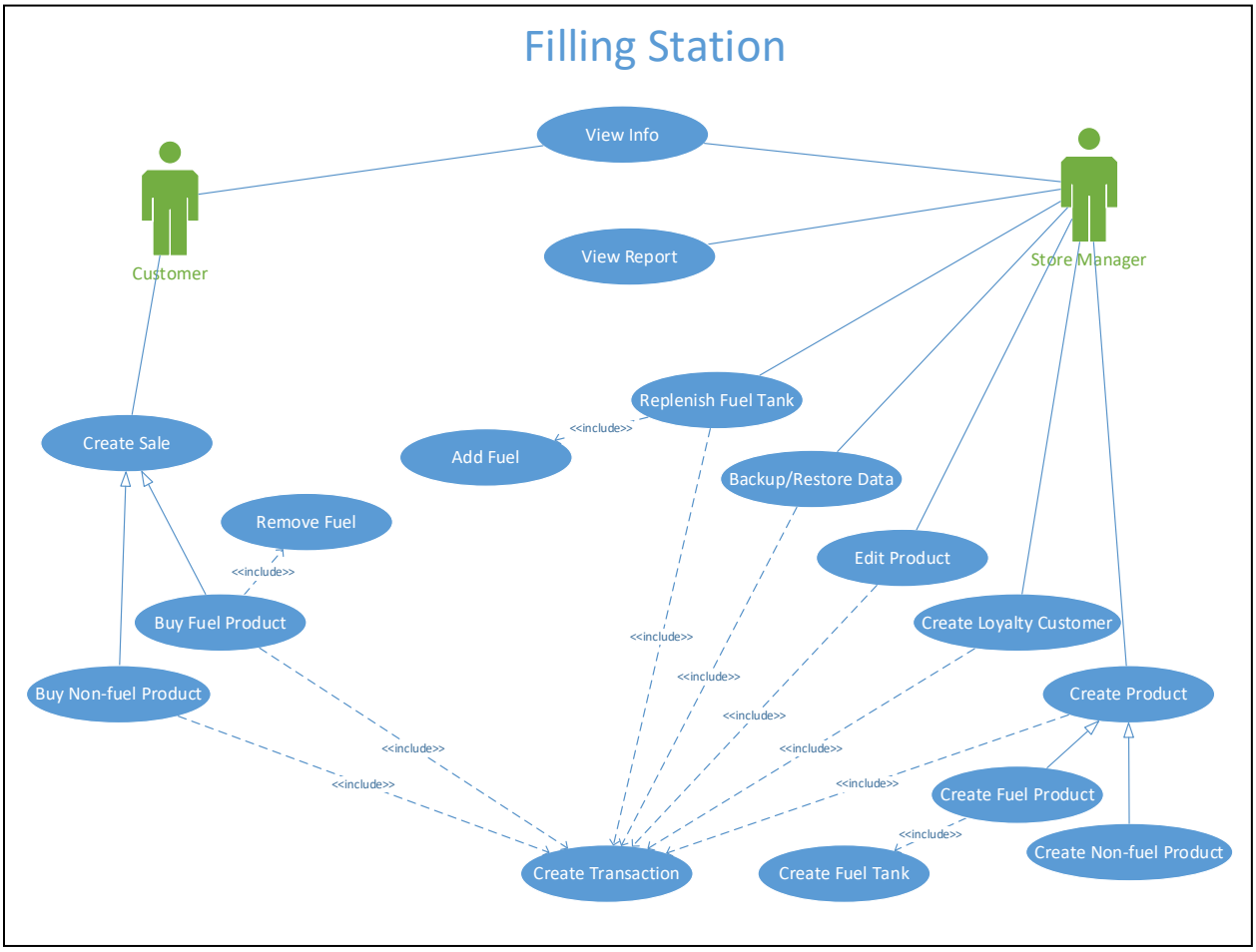

Figure 1: UML Use Case Diagram

The following high level use cases will help you to understand the user requirements better. This list represents the general functionality that you can expect for the final project submission. The requirements will change throughout the semester depending on the pace of material covered in lectures. **Not all of these requirements are expected until the end of the semester**; however, they are provided up front to keep you from going astray early. Carefully read each project assignment to understand the specific functionality required for that particular assignment. Details on these requirements will also be covered in lectures and help sessions throughout the semester.

ONCE AGAIN: some modifications and refinements WILL be made throughout the semester as the project matures. Be sure to understand the specific requirements for each project phase. Contact your IC if you have questions or concerns on what is required for any project phase.

Use Case #1: Create a Loyalty Customer	
Description	A Store Manager will create a new loyalty customer in the application before that customer may accrue rewards or receive discounts. Customers do not have to be loyalty customers. The customers who do not have a profile in the system are considered regular or non-loyalty customers.
Main Success Scenario	Manager correctly enters the following data into the application: • Alphanumeric ID unique to the customer (Example: CUST-001) • Name of the customer (Given name and surname combined into just one entry for simplicity) • Phone Number of the customer • Security Pin • Member Since. Defaults to the current system date. • Membership Age (in whole years) calculated based on current system date and member since date. • Accrued Reward Gallons. Defaults to zero.
Other	• Identifier must be unique and alphanumeric. • Name must be a free form text entry. • Phone must be unique and a proper US phone number format. • Security Pin characters should never be shown in plain text on the UI. Use the "UseSystemPasswordChar" property to hide the data input. Security Pin should always be encrypted while in data objects or in flat files. The security pin is alphanumeric. • Member Since must be a valid date not in the future. • Management is interested in knowing how long loyalty customers have been enrolled (in years). • Address and other demographic information is NOT required.

Use Case #2: Create Product	
Description	The Filling Station sells a variety of products. They currently offer four types of fuel: Regular, Premium, Super, and Diesel. They also offer three types of car washes: Standard, Enhanced, and Deluxe. Finally, they offer a five minute block of compressed air. The owners of the gas station may consider selling other products in the future (such as motor oil or wiper

	fluid). The products should be classified into "Fuel", "Car Wash", or "Miscellaneous" categories to enable reporting roll ups. To encourage customer loyalty, those customers who are registered as a Loyalty Customer enjoy a discount on all fuel products, all the time – this is called the "Loyalty Discount" (Currently, the loyalty discount is 3 cents per gallon but is subject to change). Loyalty customers also receive a "Loyalty Discount" on all car wash products all the time (Currently, this discount is $1 for each carwash, subject to change). In addition, after a loyalty customer accrues 100 gallons of gas purchases (of any grade), they receive a per-time "Reward Discount" on their next fuel purchase. This discount is instead of the loyalty discount (i.e. they are non-additive). Currently, the Reward Discount is 10 cents per gallon, also subject to change. The Reward Discount is applied every 100 gallons. Only one reward is allowed at a time (in other words, if a customer were to buy 300 gallons of fuel at one time, they would only qualify for one reward redeemable at their next visit). Example: Loyalty Customer, Adam, receives a 3¢ per gallon discount for every gallon they purchase. When he reaches 100 gallons in total purchases over any number of visits, his next visit will allow for a 10¢ per gallon discount on that one fill up. His accrued reward gallons resets to zero and he goes back to his 3¢ per gallon reward until he reaches the next 100 gallons in purchases. There are no limits to how many rewards Adam can earn. Regular (non-loyalty) customers always pay full price. See a sample advertisement below to further illustrate business rules. The Store Manager will create entries in the application to be able to sell these products. When a Fuel product is created, the Store Manager must also create data for a Fuel Tank for storing fuel at the same time. Fuel tanks are located under the filling station and hold a large quantity of saleable fuel. Fuel tanks will have their own unique ID as well as the maximum quantity of fuel they hold (popular fuel types typically have larger tanks of reserves), and the current quantity of fuel. A fuel product is not allowed to be created without an associated fuel tank. Likewise, fuel tanks are not allowed to be created unless they are associated to a fuel tank. Car Washes and miscellaneous products (non-fuel products) do NOT have fuel tanks.
Main Success Scenario	Manager correctly enters the following data into the application: • Alphanumeric ID unique to the Product (Example: PROD-001) • Name of the product (Free form text, such as Diesel Fuel) • Classification of product (One of three values: "Fuel", "Car Wash", "Miscellaneous")

	• Unit of Measure (such as “Gallon” or “Each”)
	• Price Per Unit of product in US Dollars (can change over time)
	• Loyalty Discount Per Unit in US Dollars – example $0.03 (can change over time)
	• Reward Discount Per Unit in US Dollars – example $0.10 (can change over time)
	• Tax Amount in Percent (5% would be .05) Taxes could be different for each product based on government regulations and can also change over time.
	
	Fuel products also have:
	• Their own unique Alphanumeric ID (Example: TANK-001)
	• Max Quantity of Fuel they can hold (In Gallons). Default value is 5000 gallons.
	  o 1 US gallon = 0.133680556 foot^3
	  o 1 gal = 0.133680556 ft^3
	  o So, for example, 5,000 US gallons would be approximately 668.4 cubic feet or approximately 24.8 cubic yards. (1 cubic yard = 3ftx3ftx3ft = 27 cubic feet.)

Other	• Identifier must be unique and alphanumeric • Product Name must be free form text entry. • Classification is chosen from one of three static values in a dropdown: Fuel, Car Wash, and Miscellaneous. • Unit of Measure is a free form text entry. • Prices are all in US Dollars. Discount and Reward values should never be more than the Price per Unit value. • Tax Rate should be a percent. Taxes are never negative, but some products may not be taxed at all. • The trigger number for reward gallons is always 100. Management is confident that this number will be constant over time. • The ID and Max Quantity of a fuel tank is entered into the UI when a Fuel Product is created. • The current level of fuel in tanks should never be entered in the UI manually. When a fuel tank is created, you can assume it will be automatically filled to capacity based on the specified Max Quantity.

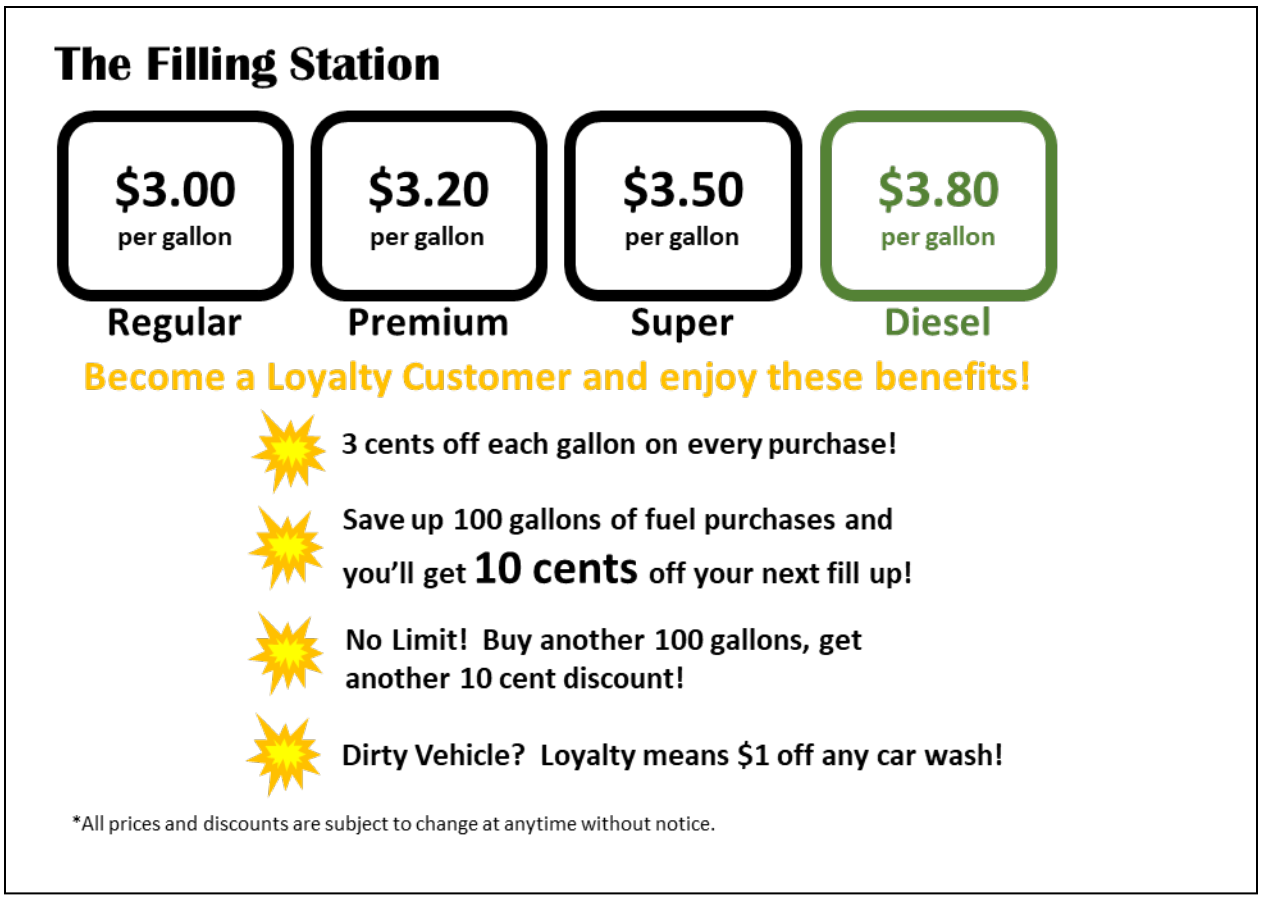

Figure: Sample Marquee to illustrate business rules

Use Case #3: View Info	
Description	A Store Manager may look at current information at any time by selecting a valid ID from the UI. Information relevant to the record should be shown in various list boxes and text boxes. This would apply to a Product, a Loyalty Customer, a Fuel Tank, or a Sale.
Main Success Scenario	An alphanumeric number that is valid in the application is selected from a list. Once selected, the various data elements for that object should be displayed on the form.
Other	• The application should pull up correct data along with any relevant lists and calculations as discussed in lectures as the project progresses. • The program should gracefully handle a situation where the selected ID could not be found in the application. (Specific details will be covered in lecture.)

Use Case #4: Edit Product	
Description	After a Store Manager successfully views a current product info by selecting a valid Product ID from a dropdown list (see Use Case 3), some data elements may be edited and updated in the application.
Main Success Scenario	Assumption: Use Case #3 for View Product is completed successfully. Once the correct data is displayed on the screen, data elements may be modified and saved back into the application where they will overwrite the previous data elements. Specifically, the Price Per Unit, Loyalty Discount Per Unit, Reward Discount Per Unit, and Tax Rate can be changed by the store manager at any time. The ID, Name, Product Class and Unit of Measure should not be allowed to be changed. Fuel Tanks are not allowed to be edited. All changes should overwrite the current product offerings but not change the accounting for previous sales.
Other	• UI should enable controls applicable to editing the product and disable controls not valid when editing a product.

Use Case #5: Create Sale / Buy Product (including Buy Fuel Product AND Buy Non-Fuel Product)	
Description	The customer may purchase a product at any time. Purchasing is a complex operation, so be sure to read the rules here carefully. 1) The Customer enters a unique ID for this sales transaction (For example: SALE-0001) 2) The customer must indicate if they are a Regular Customer or a Loyalty Customer. If the customer selects a loyalty customer phone number, continue with step 3. Regular customers will leave the phone number dropdown blank and continue with step 5. 3) The loyalty customer provides their security pin. If a customer's name/PIN value is invalid, the system should present an error and not allow the transaction to proceed until the proper pin is selected. (Or, the customer changes to a regular transaction) 4) Once a loyalty customer is validated, the system should provide the current accrued gallon rewards as well as a list of previous sales for that customer. 5) The customer then chooses the product that they wish to purchase. a. If the product selected is a **fuel product**, the system should identify the correct discount rate based on customer type (regular vs loyalty) and the accrued reward gallons. If the accrued reward gallons for the customer is greater than 100 (this number never changes), then the customer is entitled to the larger reward discount. b. If the product selected is a **non-fuel product** (car wash, miscellaneous), then the number of accrued reward gallons becomes irrelevant for that purchase, but a loyalty discount may still apply.

	6) The customer selects how much of the product they wish to purchase. If it is a fuel purchase, the system must check that enough product exists in the fuel tanks (the system can only sell what is available!) 7) The Sale Date is defaulted to the current system date/time. 8) Calculations are made and the sale is posted. NOTE: Unlike most retail stores, taxes at filling stations are included in the customer's price. In other words, the Extended Price is calculated first (Price Per Unit – Discount Per Unit) * Quantity Purchased. Then, taxes are deducted from that amount. For example, if a customer purchases 10 gallons of gas at $5 per gallon (after all the discounts are applied) then the Extended Price is $50. If the tax rate on that transaction is 20%, then the Net Amount to the filling station is $41.67 and the Tax Amount is $8.33. Net Amount = Extended Price / (1 + tax rate). In this case, 20% would be 0.20 in decimal, and 1 + tax rate would be 1.20. 9) Price Per Unit, Discount Per Unit, and Tax Rate must be captured and stored at the time the sale was posted in order to keep accurate information as the price fluctuates. 10) After the sale is posted, for **fuel products** only: a. Accrued reward gallons are incremented for loyalty customers b. Current fuel quantities are decremented for fuel tanks. See the Remove Fuel use case below. c. If a loyalty customer was entitled to use their 100-gallon reward discount, then their accrued reward gallons total becomes zero. (i.e. they reset and start accruing for their next discount) d. If a tank falls below 5% full, then a warning should appear in the UI and/or the transaction log. In this case the Manager may choose to initiate the Replenish Fuel Tank process manually. e. If a tank falls below 2% full, then the system will initiate the Replenish Fuel Tank process automatically itself. See the Replenish Fuel Tank use case below for more details.
Main Success Scenario	The customer initiates a sale by identifying themselves from a drop down box + Security PIN (or accepting regular customer terms), the product and quantity they want to purchase, and supplying a unique alphanumeric ID for the transaction. The system identifies correct discounts and applies the correct calculations to create a Sale object and adjust quantities as necessary.
Other	• Identifiers must be unique and alphanumeric. • Customer and Product should be found in the application. If not, the application should handle the error gracefully. • Sale Date should default to current system date, but be allowed to be changed.

	• Quantity should be a decimal and always positive. • The customer can only buy as much gasoline as is available at the start of the purchase. • Security PIN characters should not be visible on the UI. The provided PIN must be compared to the encrypted PIN stored in the customer object to validate the customer.

Use Case #6: Remove Fuel	
Description	After every successful fuel purchase, the amount of fuel that was purchased must be emptied from the fuel tank. In addition, the application must react to certain business event triggers related to the level of fuel in the tanks. If the current fuel level in the fuel tank falls below 5% full based on its maximum capacity, then a warning message is issued to the store manager and a command button is enabled to allow for a reorder to replenish that fuel tank. If the current fuel level in the fuel tank falls below 2% full based on its maximum capacity then the system automatically orders and replenishes the fuel for filling station without store manager interaction.
Main Success Scenario	The fuel product purchased is associated with a specific fuel tank. That tank must have the correct quantity of fuel deducted. After the deduction, the system should check for two conditions: less than 5% full and less than 2% full. The 5% full condition triggers a warning level that the fuel tank is low. The 2% full condition triggers the Replenish Fuel Tank use case automatically.
Other	• Consider using custom events as discussed in lecture to trigger the event actions.

Use Case #7: Replenish Fuel Tank and Add Fuel	
Description	The fuel in Fuel Tanks are only able to be increased by ordering more fuel from the distributor. This can be done one of two ways: either the manager selects a tank from the UI and manually requests a reorder, or the system automatically issues a reorder. In reality, an order would be placed and tanker truck would appear later. For this simulation, you can assume the tanker truck will appear immediately and the fuel will be automatically available for the next sale. A tanker truck can only carry 4000 gallons of a fuel for any order.
Main Success Scenario	Replenish Fuel Tank is called from one of two methods but resolves exactly the same for both method: the amount of fuel is increased until the fuel tank is full or the tanker truck is out of fuel.

Other	• Remember that a fuel tank can never have a current fuel level less than zero or greater than that tank's maximum capacity.

Use Case #8: Create Transaction	
Description	Every data change in the application must be captured in order to accommodate Backup and Restore processes. As each process completes, an entry with a specific string format will be recorded. The transactions should be visible in the transaction log that is displayed in the UI as well as in an array in the application. For transactions that are logged from UI input: errors should be caught interactively and displayed to the user for immediate correction. For transactions that are logged from the input of a flat file (see Backup/Restore Data below), errors are noted in the transaction log through the use of an "Is Error" flag.
Main Success Scenario	All transactions logged in the proper format.
Other	• Specific text formats will be provided later and discussed in lectures.

Use Case #9: View Report	
Description	The store manager needs to see certain reports and transaction logs.
Main Success Scenarios	1) The user needs to see scrollable lists of all data in the application: customers, products, etc. 2) The user needs to see a running transaction log: a scrollable list of all transactions that have occurred in the application, as they occur. 3) Calculations/metrics should appear on the summary and/or report page and update automatically as transactions happen.
Other	• Calculations should be accurate at all times. • Be careful to not divide by zero when calculating averages. • Specific report requirements will be provided below as soon as management has determined which key performance indicators (KPIs) are most important. Reports may include (but not limited to): sales reports per category of product, total discounts offered, average sale amount, etc… • More details on the reports will follow as the semester progresses.

Use Case #10: Backup/Restore Data	
Description	The application must be able to backup and restore in multiple ways. A Refresh button will be required that will reset the UI and all data objects to an initial state. A requirement to reset all data objects and then replay the transactions in the transaction array may be required as well.

	A “Save” button will export the transactions to a flat file for offsite storage. A “Load” button will be used to import and replay transactions from a flat file. Finally, a “Test Data” button is required for the QA department to hardcode some transactions for testing purposes.
Main Success Scenario	The application should be able to load/process transactions from any of these methods as well as through standard UI inputs through the normal course of business. The application should be able to export transactions to a flat file in the proper format.
Other	• Specific file formats will be provided later.

Management would like the following metrics/calculations shown on the Summary Page:

Required:

- Grand total gross total dollar amount of each type of sale (Fuel, CarWash, and Misc). (One public method, returns a single dollar amount, depending on the single Enum Product Type parameter.)
- Grand total dollar amount of tax collected on each type of sale (Fuel, Car Wash, and Misc). (One public method, returns a single dollar amount, depending on the single Enum Product Type parameter.)
- Gross dollar average sale amount. (One public method, returns a single dollar amount, no parameters.)
- Grand total dollar amount and number of sales for specified customer. (Two public methods, one returns a single dollar amount and the other returns a single integer value, depending on the single Customer ID parameter.)

Optional (5 extra-credit points if all – required and optional – are completed correctly):

- Grand total percentage of dollar amount and percentage of number of each type of sale. Ex: Dollars: 88% fuel, 2% car wash, 10% misc.; Number: 25% fuel, 10% car wash, 65% misc. (One public method, returning a single percentage amount, depending on the Enum Product Type parameter and the string parameter that specifies "$" or "N". "$" indicates that the percentages returned should be for dollars, "N" indicates that the percentage returned should be for the number of sales. The return value should be in decimal – Ex: return 0.10 for 10%, etc., but display 10% on the Summary Page.)
- Gross dollar smallest and largest sale amounts, and what type. (Two public methods, each returning a single dollar amount and making the Enum Fuel Type available in the single pass-by-reference parameter.)

All of these metrics must be calculated and not persistently stored in variables. The Summary Page should update anytime the data changes in the application. Dollar amounts should allow for, track, and provide dollars and cents.

The following UML Class Diagram (**subject to change**) summarizes this structural information and the interrelationships between Classes. All data will be stored in memory in the application (i.e. no databases will be used).

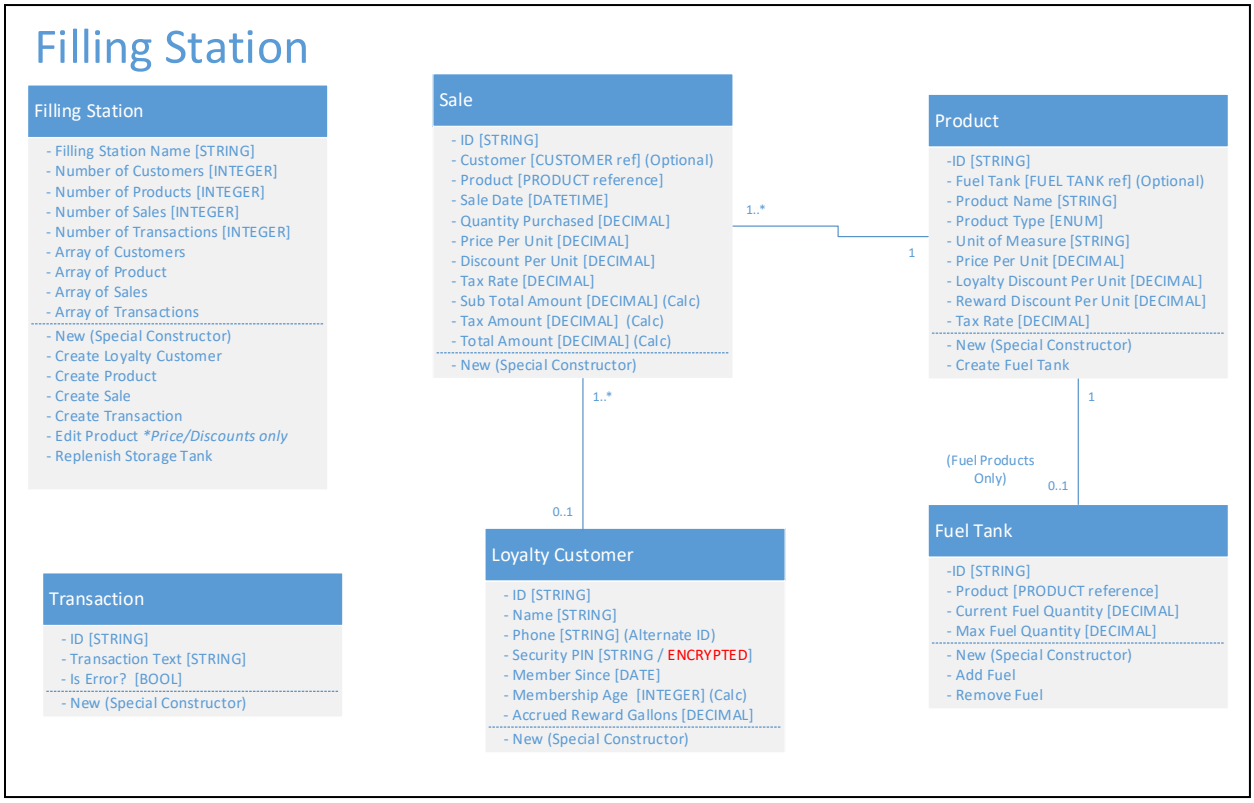

Figure: UML Class Diagram

Semester Project: Deliverable 4 – Full System

Due: December 12, 2017 at 11:59 PM Mountain Time

Points: 100 (weighted in the Project Category)

Solution Name: Proj04-FillingStation-LLLL…-FFFF… (LLLL = your last name, FFFF = your first name, CamelCase)

Project Name: FillingStation

Continue to use the same project that you submitted for Project 3. (1) Rename the solution directory to Proj04-LLLL-FFFF, where LLLL and FFFF are your last and first names using CamelCase. (2) Rename any .sln and .suo files within it (to say Proj04 instead of Proj03 and to use your LLLL & FFFF name). (3) Double click on the .sln file to open the new copy of your Solution. You will then be ready to work on Project 4.

For this increment of the project, you will finalize the project with complete working functionality. This includes all functionality described on the business requirements, any additional requirements discussed during lecture, and use of proper techniques demonstrated in lectures. The focus in this increment is on flat files (input and output), arrays, and complete coordination between the UI layer and the business logic layer.

There is a lot to do here, but most of the individual pieces will be relatively straightforward. However, because so much is going on now, that can lead to complexity and mental confusion. It is important that you work steadily in an iterative/incremental way to build up these final pieces. Below are some recommendations that may help with this:

1. All data should be able to be entered equivalently from:
    a. Keyboard/UI
    b. Process Test Data method
    c. Flat File in the format provided

2. Work on the simpler business processes (Customer) before working on the complex ones (Sale).

3. Arrays: this iteration will store data as the program is running.
    a. You will must use arrays in your classes to store the data objects. (Refer to the class diagram and information provided in lectures).
    b. You must design and implement these arrays using the approach illustrated in class over the past several weeks, manually managing the memory, etc…

c. You must add Iterators for arrays so that you can get information to display on the GUI. (The design must not expose the arrays to the other parts of your system outside of the class in which they are defined.)

4. Flat Files
    a. You will need to manage several files: (1) reading in transactions, (2) writing out all transactions, and (3) writing out all error transactions. These MUST be named as specified and located in the bin\Debug subdirectory of your solution directory hierarchy. The input file MUST be named "Transactions-in.txt", the main output file must be named "Transactions-out.txt", and the error file MUST be named "Transactions-errors.txt". Note the exact naming used: points will be deducted if your IC has to rename their test file in order to grade your project.
    b. A sample Transactions-in.txt file will be provided as part of this assignment. It will show the format that the input and output files will be in when we grade your assignment. You can use this sample file to help test the execution of your program. You should also edit it to further test the functionality. When we grade, we will use our own different transaction file to test your program.
    c. Notice the file is semi-colon delimited (not comma delimited).
    d. Notice the file can have blank lines and comment lines (lines that start with a # as the first character). All of these lines should be "skipped over" when reading the file and processing the transactions – specifically, these lines are not "processed" but they are saved in the transactions array but not "counted" as actual transactions. Only successfully-processed transactions are "counted" and displayed in list/combo boxes.
    e. When saving output, your program should save both the "all transactions" file and the "error transactions" files at the same time.
    f. The sample flat file will provide column headers for your reference
    g. The flat file used for grading will have the same structure. It will not contain "bad" data in terms of invalid integers, decimals, or dates. It will also not contain bad data in terms of the file structure itself. However, there WILL be "bad" data in terms of attempts to add duplicate objects and to add references to non-existent objects, etc. Your program must check for these types of errors and handle them all gracefully.
    h. The format of all the files is identical. We should be able to export your flat file, rename it, and use it as an input file successfully.

5. Event and EventArg classes
    a. You will need to continue to use EventArg classes to pass event data along to FrmMain
    b. Clean up and add any EventArg classes based on feedback from Project 3

6. Search Methods
    a. Public search methods (for finding items in arrays) should take a single ByVal parameter specifying the "ID" to search for and should return a reference to the found item (or Nothing, if not found).

b. Private search methods should work as Public methods do, but in addition should have a ByRef parameter telling the array location in which the item is found. This location will be meaningless if the item is not found.
c. These methods should be implemented as demonstrated in class.

7. You should test for handling bad data in the UI, ProcessTestData method, and the Flat File input and make sure your program handles the bad data gracefully.

8. Display look up information
    a. Note that in many places when an ID is selected, then the associated information should be displayed in a TextBox nearby.
    b. When an ID is selected from a combo box, all other data on that tab should be filtered based on that selection. For example, when selecting a Customer ID in the Customer tab: the Trx list should only include transactions carried out by that selected customer.
    c. The summary page should display the associated objects "ToString" data in a nearby, shared TextBox.
    d. An error message should be shown in the Textbox when lookups fail.

9. Include all calculations and summary information required in the Summary Tab and as described in the business requirements. Remember that these calculations should not store the data persistently in variables. The information should be updated whenever any business process occurs. Look at the identified "KPIs/Metrics" that are summarized above right after the Use Case Descriptions and right before the Class Diagram.

10. Ensure the "Modify Product" business process is working correctly. A new object is NOT created when an account is modified. Instead, find the existing object and update it with the changed data.

11. Error Checking: You must perform error-checking with Try-Catch, IF, IIF, or Select-Case statements throughout your program.
    a. As in previous projects: bad data provided in the UI should not cause the program to crash. (The flat file will always have valid integers, decimals, dates in the right places).
    b. Blank ID's should never be allowed.
    c. Duplicate ID's should never be allowed.
    d. Min/Max rules should be followed, etc…
    e. Use Try/Catch and selection for their intended uses. Specifically, don't use Try-Catch when If/Then selection would have been better.

12. Add a Trx ID field to each tab where inputs are required, two to the Product tab (one for the product and one for the Fuel Tank). Update the business process methods to pass these in and use them when creating objects.

13. Modify Customer and Sale ToString methods so that there are two options: (1) Indicate member since age based on current date as the reference date, and (2) indicate member since age based on reference date provided as a single parameter to a new ToString method. To do this you will need two ToString methods: (1) the one you probably currently have which has no parameters and uses the current system date as the reference date, and (2) a new one that will have a single parameter that specifies the reference date to use. The member since age method of the Customer class should already have a single parameter that provides the reference date; if yours does not, update it so that it does.

14. Clean up any errors from Project 3 based on feedback that was given to you.

15. Put on the final "polish" to make your program look and behave professionally.

Required Elements (check these off before you submit)

- ☐ Zip File with all necessary files included
- ☐ Project compiles correctly with no compile errors
- ☐ Project runs with no run time errors (test every data entry and file operation)
- ☐ Usability: tab stops, message boxes, etc…
- ☐ Supportability: comments, correct use of template, etc…
- ☐ Code Template for ALL classes with header information filled in
- ☐ Data entry validated and all business rules implemented
- ☐ Correct output in the Transaction Log
- ☐ Correct data in ToString text boxes
- ☐ Custom Events / Event Arg classes cleaned up as necessary
- ☐ Click events for all buttons: everything must work
- ☐ File Read processes implemented
- ☐ File Write process implemented
- ☐ Required calculations and metrics complete and accurate
- ☐ Arrays implemented according to approach discussed in lecture including proper usage, memory management, "Find" methods, Iterators, etc…
- ☐ Proper passing of data between classes