Fall 2015 CIS605 Semester-Long Project

Theme Park Management 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 fairly closely in addition to checking for proper functioning of the running system.

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

	Deliverable	Due	Points
1	User Interface	September 17	25
2	Class Design + Basic Code + Input Validation + hard coded test data	October 8	50
3	Custom Events + More code + hard coded test data	November 12	75
4	**Full System with Arrays + File I/O**	**December 10**	**100**
			250

For this project you will be developing a system in Visual Basic .Net, using Visual Studio 2012, 2013, or 2015, which implements some of the functionality that might exist with theme park management. 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 allow the purchase and use of theme park features. By the fourth project submission, you will have a completed working system that manages customers, features, and many logistics of theme park tracking.

The following use case diagram summarizes the various actors that may use the system. You do NOT need to create separate applications for each. In addition, you do NOT need to worry about user access, privileges, or passwords. You may assume that all actors will use the same program and that each actor will only use the parts of the UI that are applicable to them.

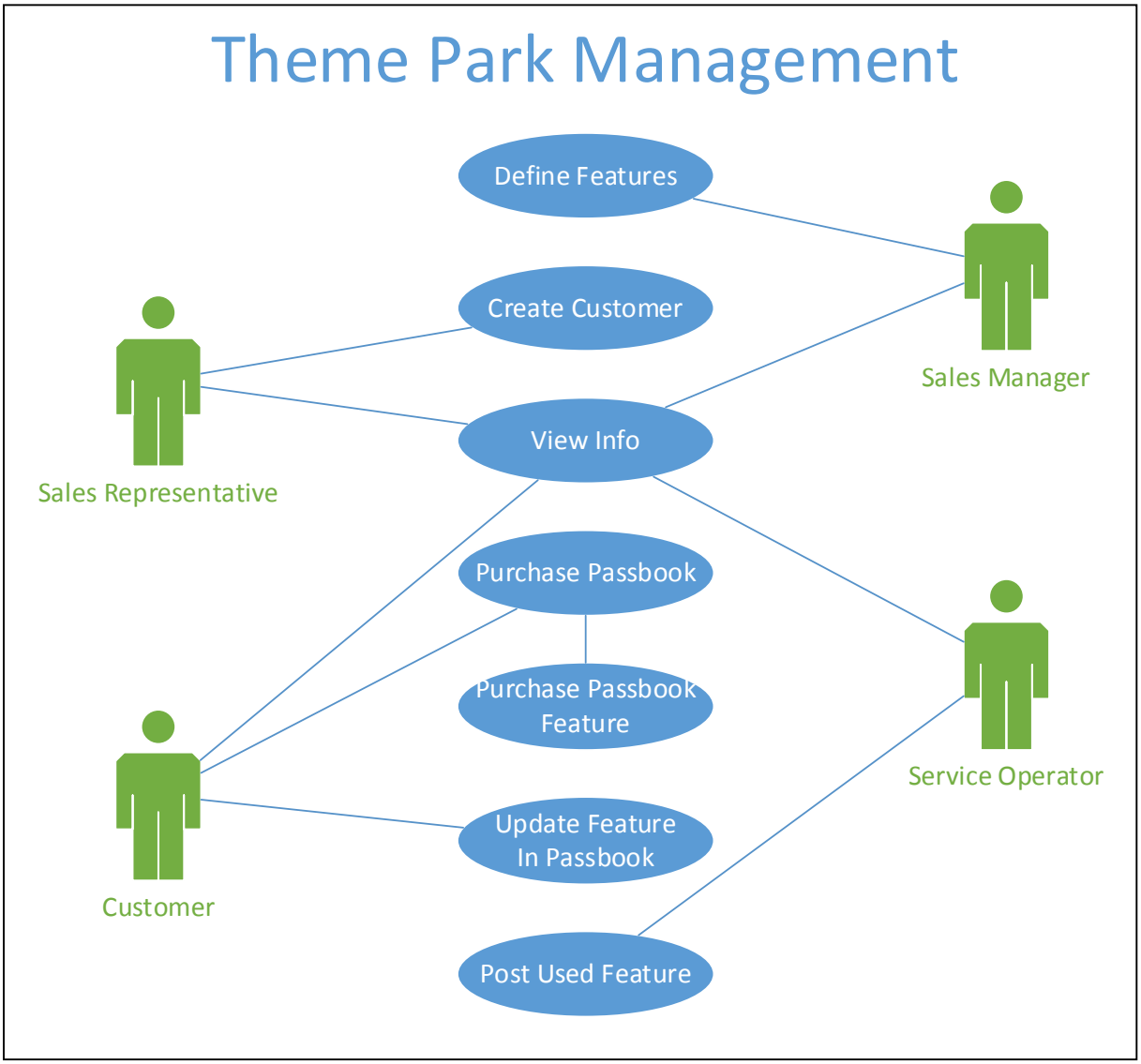

Figure 1: UML Use Case Diagram

The following high level use cases will help you to understand the user requirements better. This list represents functionality for the final project submission, and will be subject to change throughout the semester. Not all 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. **NOTE: some modifications/refinements will likely be made throughout the semester as the project matures.**

<table>
<tr><td colspan="2">Use Case #1: Define a Feature</td></tr>
<tr><td>Description</td><td>A Sales Manager will periodically add features (services) that are sold to customers.</td></tr>
<tr><td>Main Success Scenario</td><td>Manager enters an alphanumeric identifier that is assigned externally and the feature name, a unit of measure (how the service is sold), and the price for Adults and Children separately. To simplify the project, you can assume that the price will never change and no coupons or discounts will ever be used. You can also assume that features are never removed. The data is confirmed and added to the system.</td></tr>
<tr><td>Other</td><td>• Identifier must be unique.
• Feature name and Unit of Measure must be a free form text entry.
• Prices must be decimals.
• Data Examples:
<table>
<tr><th>Feature Name</th><th>Unit of Measure</th><th>Price (Adult/Child)</th></tr>
<tr><td>Park Pass</td><td>Day</td><td>$100 / $80</td></tr>
<tr><td>Parking Lot Pass</td><td>Day</td><td>$15 / $15</td></tr>
<tr><td>Meal Plan</td><td>Meal</td><td>$30 / $20</td></tr>
<tr><td>Early Entry Pass</td><td>Day</td><td>$10 / $5</td></tr>
<tr><td>VIP Pass (cut in line at attraction)</td><td>Attraction</td><td>$...</td></tr>
<tr><td>Birthday Gift Package</td><td>Each</td><td>$...</td></tr>
<tr><td colspan="2">Etc… Free form text should allow any number of services and units.</td><td>$...</td></tr>
</table></td></tr>
</table>

<table>
<tr><td colspan="2">Use Case #2: Create Customer</td></tr>
<tr><td>Description</td><td>A Sales Rep will create a Customer before they purchase their first Passbook. The Customer is typically the head of the household or the trip planner. Only an ID and the customer name is required. (Other typical fields such as address, email, etc… are not required for this project)</td></tr>
<tr><td>Main Success Scenario</td><td>Manager enters an alphanumeric utility identifier for the customer that is assigned externally and the name of the customer. The data is confirmed and added to the system.</td></tr>
<tr><td>Other</td><td>• Identifier must be unique
• Customer name must be free form text entry. (Given name and surname should be combined into just one entry for simplicity)</td></tr>
</table>

Use Case #3: Purchase Passbook	
Description	Customers can create any number of passbooks which will hold all the features that they buy. One passbook is created per visitor related to the customer. For example, one customer may be Joe with three of his passbooks belonging to Joe, Jen, and Jack. The Passbook is given an ID, a reference to the owner (by choosing the owner in a dropdown list), the date that the passbook was purchased (always defaults to the system date), the name of the person belonging to the passport, that person's birthdate. The birthdate is then used to calculate an age based on the system date and a determination if the visitor is an adult or a child. A child is defined as anyone under the age of 13.
Main Success Scenario	An alphanumeric number that is assigned externally is entered. A dropdown list is provided of customers and one is chosen as the person purchasing the passbook. A textbox should be provided to enter the visitor's name (given and surname can be combined into one field). A control should also be used to enter the visitor's birthdate. The data is confirmed and the passbook is added to the system.
Other	• Identifier must be unique. • The customer must already exist in the system. • Date Purchased does not need to be entered – in the GUI this can default to the current system date; in hard-coded test data this can be specified. • Birthdate must be entered. Age and IsChild must be calculated based on the birthdate entered and the current system date.

Use Case #4: Purchase Passbook Feature	
Description	The Customer must be able to buy features and apply them to their passbooks. Customers can buy any quantity of any given feature. Examples of features are provided in Use Case #1. The amount should be calculated and stored in the object. The amount is the price of the feature (based on adult vs child prices of the passbook visitor's IsChild status) x quantity purchased. The determination of whether to use a child or adult purchase price should be made based on the date of purchase as the reference date.
Main Success Scenario	An alphanumeric passbook feature ID that is assigned externally is entered. The customer selects the Passbook and Feature based on drop down lists. When a passbook is selected, the user should be able to see and verify the passbook visitor name, visitor age, IsChild status, and the customer name (which could be different than the visitor name). When a feature is selected, the user should be able to see and verify the feature name, unit of measure, and correct price of the feature based on the visitor's IsChild status. A quantity is then entered and the transaction is validated and added to the system.
Other	• Identifier must be unique. • The customer and the feature must already exist in the system. • Units must be numeric (decimal). • Quantity purchased should never be negative. • Price of the feature must match the visitor's IsChild status

Use Case #5: Update a Passbook Feature	
Description	The customer must be able to increase or decrease any given feature quantity. For example, a customer may have originally purchased 4 days of park entrance feature, but would like to add 2 more for a total of 6.
Main Success Scenario	The sales rep enters or selects the alphanumeric ID that is assigned to the passbook feature being updated. When the ID is selected, the user should see passbook and customer information, feature information, units remaining, expiration date, and a list of all used features (if any). The sales rep then enters the new amount of total features (not the delta). The update textbox should default to the original quantity purchased. The data is validated and changed in the system: amount and quantity are adjusted.
Other	• Identifier must exist. • Units must be numeric (decimal). • Quantity purchased should never be negative. • Price adjustments must consider the visitor's IsChild status, using the date of update as the reference date.

Use Case #6: Post a Used Feature	
Description	The Park Employees must be able to register the use of a feature. For example, when the visitor arrives at the park, the system is checked that the Park Ticket feature is in the Passbook and that sufficient quantity exist to allow entrance. The location where the feature was used is indicated in a free-form text field along with the quantity used. Quantities could be decimal. For example, late arriving guests may only be charged a half day of park ticket. Or, a lunch may only be a half quantity of a meal ticket whereas dinner may be a full quantity. This does not need to be kept in the system, you can assume the employee will know and enter the correct quantity used.
Main Success Scenario	The employee enters an externally assigned alphanumeric ID that identifies the usage of the entitlement. The employee selects the Passbook and Passbook Feature being used. After the employee selects this information, they should see visitor information, feature information, and the number of features remaining. The employee then enters the number used. Finally, the employee enters the location where the feature was used (e.g. "The 80's Diner" or "Parking Lot A" or "Super Rollercoaster"): this field is always free form text. The date used always is set to the current date (no opportunity to modify it). After the employee enters all the information, the system should check if enough quantity exist to allow usage of the feature and then post the transaction.
Other	• Identifier must exist. • Rejected requests (not enough features left) should display an error.

	• Approved requests should automatically decrement the number of features left remaining.

Use Case #7: View Info and KPI's (Key Performance Indicators)	
Description	All users need to see key indicators and transaction logs.
Main Success Scenario	The user views correct calculations for various types of information. These calculations will be provided as the project progresses. The user also needs to see scrollable lists of all data in the system: customers, services, and entitlements. Finally, the user needs to see a transaction log: a scrollable list of all transactions that have occurred in the system, as they occur.
Other	• Calculations should be accurate at all times. • Be careful to not divide by zero when calculating averages. • See KPI's below

Use Case #8: Test Button; Read / Write Files	
Description	The system should have a hardcoded test data button. The system must be able to import data files and export backups.
Main Success Scenario	Users should be able to load transactions from 1) UI per previous requirements, 2) hard coded data in a test button, 3) flat data files. Users should also be able to export all transactions in a flat data file.
Other	• Not required for Project 1 • Specific file formats will be provided later. • More details on this requirement will also be provided later.

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

- Average balance of unused PassbookFeatures In Dollars
- Sum of unused PassbookFeatures in Dollars
- Average number of Passbooks per Customer
- Most Popular Passbook Feature Purchased
- Percent of Passbook Features Used (Used / Total). Calculate this based on dollars of each feature purchased and used. Use the isChild status at date of purchase and date of use to determine the adult/child price to apply.
- Average Age of all Passbook holders
- Number of Passbook holders who have a birthday this month

All of these metrics must be calculated and not persistently stored in variables. The Summary screen should update anytime the data changes in the application.

The following **UPDATED** UML Class Diagram summarizes this structural information and the interrelationships between Classes. All data will be stored in memory in the application (i.e. no database management systems will be used). More details will be provided on the business logic classes as the project progresses, and the UML Diagram will be refined over the duration of the course.

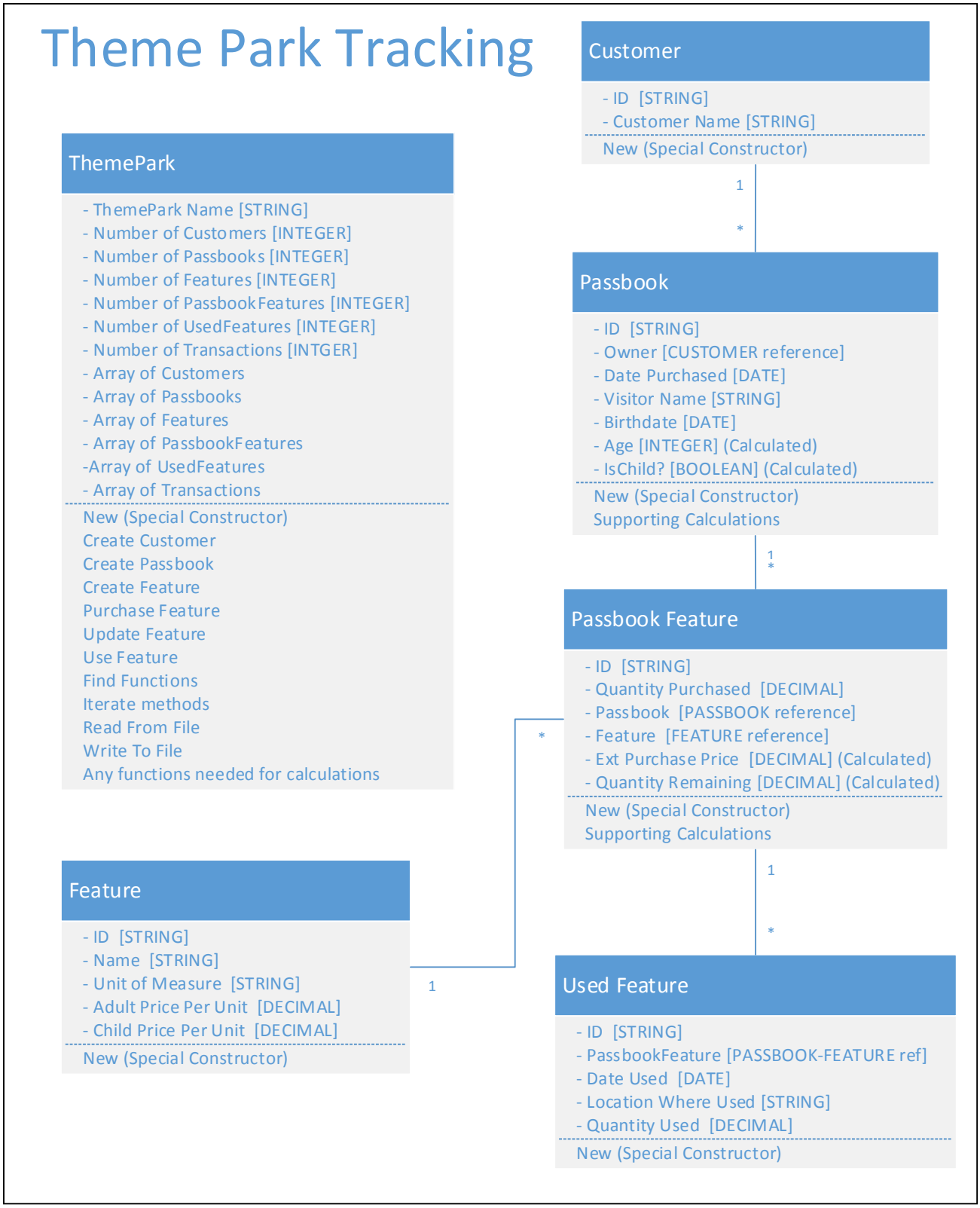

Figure: UML Class Diagram

Semester Project #4: Full System with Arrays and File I/O

Due: Thursday, December 10, 2015 at 11:59 PM Mountain Time

Points: 100

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

Project Name: ThemePark

> Copy your Proj03 directory and make a new copy in a directory called Proj04-LLLL-FFFF. Then rename the .sln and .suo files within it (to say Proj04 instead of Proj03) and then double click on the .sln file to open the new copy of your Solution. Do the same each time you start working on a new part of the project: Proj02, Proj03, Proj04.

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. Transactions should be able to be entered "equivalently" from the keyboard/GUI, the ProcessTestData behavioral method in your main form, and from a flat file in the format as the one provided.
2. You will be more successful if you work on getting simpler business process working before complex ones. Work the primary classes (customer, feature) before working on the dependent classes (passbook, passbook feature, used feature).
3. Arrays: this iteration will store data as the program is running.
    a. You will need arrays in your ThemePark class to store the data objects.
    b. You will also need an array in the ThemePark class to hold the transaction lines that will be needed to write back out to your output file.
    c. You need to design and implement these arrays using the approach illustrated in class over the past several weeks, manually managing the memory, etc…
    d. You will need to add Iterators for some 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 two files: (1) reading in transactions, (2) writing out 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" and the output file must be named "Transactions-out.txt" Note the exact naming used in these names.
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.
e. The sample flat file will provide column headers for your reference
f. The flat file used for grading will have the same structure. It will not contain "bad" data in terms of valid integers, decimals, and 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 things and handle them all gracefully.
g. The format of both the input file and the output file is identical. We should be able to export your flat file, rename it, and use it as an input file successfully.
h. The structure of each of the types of transactions is listed below.

<trx date>; <trx time>; FEATURE; CREATE, <id>; <description>; <units>; <adult price>; <child price>

<trx date>; <trx time>; CUSTOMER; CREATE; <id>; <name>

<trx date>; <trx time>; PASSBOOK; CREATE; <id>; <customer (owner) id>; <purchase date>; <visitor name>; <visitor birthday>

<trx date>; <trx time>; PASSBOOK_FEATURE; PURCHASE; <id>; <quantity>; <passbook id>; <feature id>

<trx date>; <trx time>; PASSBOOK_FEATURE; USE; <id>; <passbook feature id>; <date used>; <location used>; <quantity used>

<trx date>; <trx time>; PASSBOOK_FEATURE; UPDATE; <id>; <date updated>; <new quantity>

i. A sample transaction file is listed below.

```
# This is a sample transaction file for the CIS605 project for Fall 2014.
# When reading the file, your program should process lines starting with a "#" as comment lines.
# Comment lines and blank lines should not be processed as "transactions".
# While the sample transactions in this file show all UPPERCASE for the types and actions
# of the transactions, your program should be able to process transaction files that
# do not use all UPPERCASE.  They should handle MixedCase or all lowercase just as well.
```

The fields within a transaction are semicolon (";") delimited. While the examples
in this file show a single space after each semicolon, your program should work
equally well when there are any number of spaces (including none) before and/or
after each semicolon.

Normal/GOOD transactions (these are examples, other combinations need to be handled, too):

20151116; 0800; FEATURE; CREATE; F001(f); Park Pass; Day; 100; 80
20151116; 0801; FEATURE; CREATE; F002(f); Early Entry Pass; Day; 10; 5
20151116; 0802; FEATURE; CREATE; F003(f); Meal Plan; Meal; 30; 20

20151116; 0815; CUSTOMER; CREATE; C001(f); CName01
20151116; 0816; CUSTOMER; CREATE; C002(f); CName 02
20151116; 0817; CUSTOMER; CREATE; C003(f); Customer Name 03

20151116; 0830; PASSBOOK; CREATE; PB001(f); C001(f); 20150915; self; 19800101
20151116; 0831; PASSBOOK; CREATE; PB002(f); c002(F); 20150916; self; 19850601
20151116; 0832; PASSBOOK; CREATE; PB003(f); c002(f); 20150917; C002(f) Visitor Name";
20021209
20151116; 0833; PASSBOOK; CREATE; PB004(f); C003(F); 20150815; self; 19750101
20151116; 0834; PASSBOOK; CREATE; PB005(f); C003(f); 20150915; C03 Visitor 1; 20021210
20151116; 0835; PASSBOOK; CREATE; PB006(f); C003(f); 20151015; C03 Visitor 2; 20021211
20151116; 0836; PASSBOOK; CREATE; PB007(f); C003(f); 20151015; C03 Visitor 3; 20031225

20151116; 0845; PASSBOOK_FEATURE; PURCHASE; PBF001(f); 1; PB001(f); F001(f)
20151116; 0846; PASSBOOK_FEATURE; PURCHASE; PBF002(f); 2; PB002(f); F001(f)
20151116; 0847; PASSBOOK_FEATURE; PURCHASE; PBF003(f); 3; PB003(f); F001(f)
20151116; 0848; PASSBOOK_FEATURE; PURCHASE; PBF004(f); 1; PB004(f); F001(f)
20151116; 0849; PASSBOOK_FEATURE; PURCHASE; PBF005(f); 1; PB005(f); F001(f)
20151116; 0850; PASSBOOK_FEATURE; PURCHASE; PBF006(f); 1; PB006(f); F001(f)
20151116; 0851; PASSBOOK_FEATURE; PURCHASE; PBF007(f); 3; PB003(f); F002(f)
20151116; 0852; PASSBOOK_FEATURE; PURCHASE; PBF008(f); 9; PB003(f); F003(f)
20151116; 0853; PASSBOOK_FEATURE; PURCHASE; PBF009(f); 1; PB004(f); F001(f)
20151116; 0854; PASSBOOK_FEATURE; PURCHASE; PBF010(f); 3; PB004(f); F001(f)
20151116; 0855; PASSBOOK_FEATURE; PURCHASE; PBF011(f); 3; PB005(f); F001(f)
20151116; 0856; PASSBOOK_FEATURE; PURCHASE; PBF012(f); 5; PB005(f); F003(f)
20151116; 0857; PASSBOOK_FEATURE; PURCHASE; PBF013(f); 2; PB006(f); F001(f)
20151116; 0857; PASSBOOK_FEATURE; PURCHASE; PBF014(f); 2; PB006(f); F003(f)
20151116; 0859; PASSBOOK_FEATURE; PURCHASE; PBF015(f); 2; PB007(f); F001(f)

20151116; 0900; PASSBOOK_FEATURE; USE; UF001(f); PBF001(f); 20151020; Epcot Center; 1
20151116; 0901; PASSBOOK_FEATURE; USE; UF002(f); PBF002(f); 20151020; West Parking; 1
20151116; 0902; PASSBOOK_FEATURE; USE; UF003(f); PBF003(f); 20151020; France; 2
20151116; 0903; PASSBOOK_FEATURE; USE; UF004(f); PBF003(f); 20151020; American Pavilion; 1

20151116; 0915; PASSBOOK_FEATURE; UPDATE; PBF003(f); 20151021; 1

Other example transactions will be provided during the course of the final 3-4 weeks of classes.
Some possible variations are identified below:

Adults:
Children:
Children at purchase, but Adults at Use and/or Update:

BAD transactions (these are examples, other combinations need to be handled, too):

IDs already exist when they shouldn't:
IDs don't exist that should:
BDays in the future:
Trying to use a quantity of a PassbookFeature for which we have already used all of our remaining quantity or do not have the amount remaining that we want to use:
Trying to update the quantity on a PassbookFeature to be less than what we have remaining:
Etc:

5. Behavioral Methods
    a. Pass parameters of the required fields to ThemePark (Ex: ID, name for creating a customer)
    b. Use constructors within the behavioral methods to create the actual objects in the business logic layer. The only object that should be created in FrmMain is for ThemePark, and there should be only ONE modular level ThemePark variable defined in FrmMain.
6. 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.
7. 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. The search should be case-INSENSIVE. In other words, it should treat upper- and lower-case characters the same while it does the search. It will not change the upper-/lower-case values within the array; rather, it will simply consider them the same for searching purposes.
8. You should test for handling bad data in all locations: keyboard/GUI (including TextBoxes, ListBoxes, ComboBoxes), ProcessTestData method, and flat file import (except that you can assume that the flat file will have correct integers, decimals, and dates). You should test various combinations of invalid input data and should hard-code one bad data item in every ListBox and ComboBox in your program, so they can be used to demonstrate that your program handles these bad items gracefully. You should already have "bad" data in every List/Combo-Box if you included the "Customer ID Here", etc., information as indicated for Project03. This "bad" data in the List/Combo-Boxes will be only in the List/Combo-Boxes, but will have no corresponding data in the Arrays and we will use this fact to make sure we are testing in all places for good data that can be displayed and good error messages when data can't be found/displayed.
9. 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. In addition, selection of any information on the summary page should display the associated objects "ToString" data in a nearby, shared TextBox.
    c. An error message should be shown in the Textbox when lookups fail.
10. Include all calculations and summary information required in the Summary Tab and described in the business requirements.
11. Error Checking: You must perform error-checking with Try-Catch, IF, IIF, or Select-Case statements throughout your program.
    a. On the UI, bad data for integers, decimals, and dates 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. Birthdays in the future (after today) should never be allowed.
    e. Make sure you use Try-Catch for its intended use, and use selection (IF/IIF/Select-Case) for their intended uses. Specifically, don't use Try-Catch when selection would have been better.

Finally, clean up any errors from Project 3 based on feedback that was given to you. Put on the final "polish" to make your program professional. You must follow the techniques and designs that we have been discussing in class. See below for a sample rubric that will be used when grading your project.

A sample rubric for Project 4 is listed below.

CIS605 Project 4

Name: **Sample Rubric**

* Note: Feedback may refer to a specific code line number ("LN"). To view line numbers in Visual Studio, navigate as follows: (Main Menu) > Tools > Options > (Options Box) > Text Editor > All Languages > check "Line Numbers"

Topic	Expectations	Points Possible	Points Received	Specific Feedback
General Functionality *(applicable to all problem sets)*				
Zip File, Solution, Projects	Program should be zipped properly and include the solution and all the applicable file(s)	Mandatory to receive a grade		For all sections: · Fix any issues from previous project submissions. · Refresh comments for all code blocks that have changed. · Ensure best practices have been applied to all new content too. · Follow directions provided in the assignment and in class lectures.
Projects compile & bug-free	Each applicable project should compile cleanly (no compile errors, but compile warnings are sometimes okay)	-20% off of final point total		
	Program should not crash while running			
General Aesthetics *(applicable to all problem sets)*				
Look and Feel	UI elements align properly on form	10		Increased focus on end-to-end experience, final "polish", production-ready code.
	UI elements appropriately sized/spaced			
	Good creativity while maintaining professionalism			
	User friendly captions and messages with proper spelling/grammar			
	Understandable ToolTips as appropriate			
	Message log entries scroll and display the most recent entry			
	Justification as appropriate (e.g. numbers are right justified)			
Usability	All tab stops correct			
	Keyboard shortcuts as appropriate			
	Appropriate Accept/Cancel functionality			
	Cursor reset to fix user input errors			
	Form reset after valid input			
General Supportability *(applicable to all problem sets)*				
Template & Header	Official class template used in all files	10		
	Header completed for all files (LNs 3-15)			

	All code is in the proper template sections			
	Empty procedures are removed			
Internal Comments	All comments use clear business terms			
	Method comments exist			
	Section comments exist where longer blocks of code would benefit from them			
	Line-by-line code for very technical operations or where the code is not self-explanatory			
	End statements closed with a comment			
Naming convention	FrmMain and all other code files are named properly			
	UI elements are prefixed correctly and named in clear business terms (exception: elements not used in code, like many lables)			
	Variables and parameters are prefixed correctly and named in clear business terms			
	Methods and properties are named correctly			
Required Methods	_initializeUserInterface			
	_initializeBusinessLogic			
	ToString private and public override in classes			
Code Style	White space used effectively			
	Good logical blocks of code (e.g. local variables defined all together)			
	Good separation of UI, Business Logic, and Data functions			
Required End User Functionality (Graded as the project is running)				
FrmMain	**Summary Tab:**	10		
	Totals updated correctly after each transaction (keyboard or file)			
	KPI Calculations update correctly after each transaction			
	List boxes update correctly after each transaction			
	Info box displays correct data when selecting a list box item			
	Data Entry Tabs	10		
	Validation: required data checks, unique ID checks			

	Create button updates all appropriate UI controls and the transaction log			
	Shows details when data is picked from combo boxes (Passbook, PassbookFeature, UsedFeature)			
	Transaction Log	5		
	Transaction textbox updated correctly after each transaction.			
Flat Files	Flat Files named correctly.	10		
	Reading text file updates all necessary UI components and displays correct data throughout all aspects of the UI.			
	Writing text file results in correct output file.			
Required Code Elements (Graded as a code review)				
FrmMain	**Button Clicks:**	15		
	Correct data validation techniques			
	Send data to ThemePark class			
	Correct Error Handling			
	Input field clean up to be ready for the next input			
	Does NOT update UI controls (other than cleaning up the input fields)			
	List Box/Combo Box:			
	Find information from ThemePark class			
	Populate necessary fields based on what was selected			
	Use iterators where it makes sense			
	Bad list box selections don't crash program			
	Metrics:			
	Correct calculations and updated after each change			
	Custom Event Handlers:			
	Updates all relevant information throughout the UI (including KPI's, list/combo boxes, transaction log)			
Theme Park Class	**Correct attributes and properties**	20		
	Total counts for each object			
	Arrays:			
	Transactions, Customer, Utility, Bill			

	Correct "ith" logic using private properties only			
	Correct memory management including constants for defaults			
	Correct parameters and logic			
	Methods:			
	Add / Create methods			
	Find Methods			
	Iterator methods			
	Other methods as necessary for calculations or functionality			
	No UI code (including MessageBox)			
	File Handling:			
	Correct use of file streams			
	Correct error handling			
	Correct Loops			
	Case statements and file parsing logic			
	Proper write file format			
	Custom Events:			
	Fire at appropriate places			
	Structured correctly			
Other Business Logic Classes	Correct attributes and properties	5		
	Constructor			
	Behavioral Methods			
	ToString			
	Other methods as necessary for calculations or functionality			
EventArgs Classes	Created for every data transaction	5		
	Inherits System.EventArgs			
	Correct attributes and properties			
	Constructor			
	ToString			
Overall Feedback and TOTAL				
Sample Rubric		**100**	**0**	