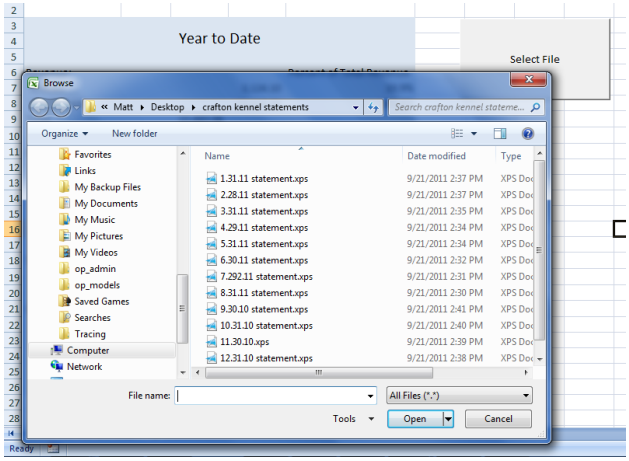
Cash Statements for Crafton Kennels

Executive Summary

Crafton Kennels is a small family business that provides boarding services for pets while their owners are away. They also have grooming faculties and limited pet care services. Because of the limited size of the business, maintaining awareness of cash is much more important than dealing with accrual accounting. To assist in this, the VBA program I wrote takes all cash deposits and withdrawals from the bank statements and creates a monthly cash statement. The program also asks the user to classify each expense (withdrawal transaction). It then sums the expense totals to be provided in the statement. Additionally, it common sizes the statement, taking relevant line items and dividing it by total revenue which highlights the leading cash sources and expenses. Finally, the workbook contains a year to date tab that sums line item amounts from all existing statements, giving the business owners a feel for their performance for the current time period of the year.

Implementation Documentation

This program's main steps are: obtain the information, parse the information, prompt the user to label expenses, create a new worksheet in the workbook with the statement template on it, update the template with the information, and update the year to date worksheet.



First, the user needs to reference the bank statement. If a check number has four digits, the user needs to put a "4" in the designated cell on the year to date (YTD) worksheet. This is important for parsing the expenses.

Next, the user must activate the program by clicking on the "select file" button on the YTD worksheet, a constant worksheet in the workbook which not generated by the program, but is updated. The user is prompted with a file dialog box to select a bank statement file located on the hard drive.

The user must be sure to select only XPS file formats, as this is critical to the program's functionality. XPS files are simple to work with because they are a type of zip file. By renaming the file extension, the XPS file changes to a zip file. One could then extract the zip file and use the files in the zip folder to retrieve the source code of the bank statement pages. Once a file is selected, the program converts the file as described above. The user should be sure to make certain that the file extensions are not hidden for this to work.

To extract the folder, the program creates a temporary folder and copies all the files out of the zip folder and pastes them into the new folder. With the new extracted folder the program then follows a designated file path to where the pages of the bank statement are located. It loops through these pages and using the agent class saves the source code from each page as a combined text. The program then deletes the temporary folder and renames the zip folder back to an XPS file so the user is unaware of any changes. See appendix part A.

Next the program parses each transaction and copies it into the statement as a new line item. This was the trickiest part of the program to get working. There are very few unique instances in the source code that allowed for parsing. The first issue was that statement headers did not precede the transactions in the source code (this would have been the easiest way to detect transactions and their type). Thus you couldn't get the agent to simply move to the top of a category when you needed the information from it. However, there are several patterns that could be exploited to parse the needed information. These patterns are:

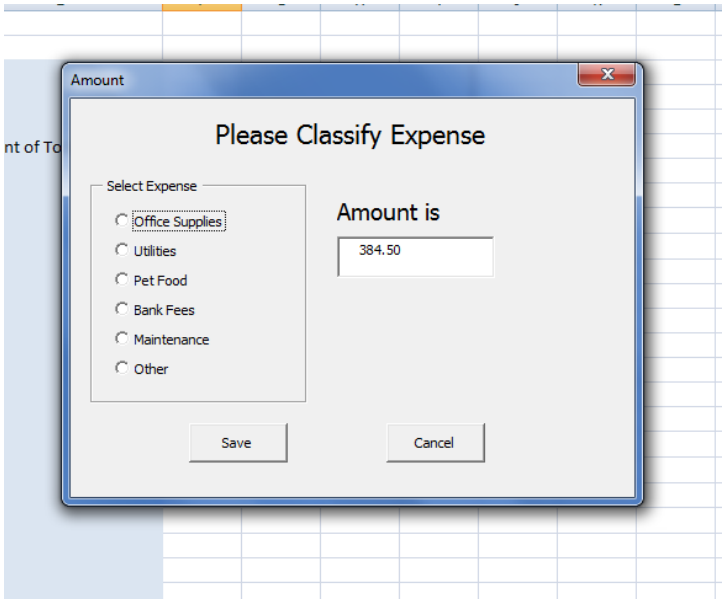
- The transactions in the source code followed the same order as they appeared in the statement.
- Deposit transactions always had the word "Deposit" in the description.
- Credit card additions always had the words "Corporate ACH Merch Dep" in a description.
- All transactions came right after "UnicodeString="" in the source code.
- All transaction followed this format "MM/DD###.##", with # being the amount of the transaction. Unless the transaction was a check, then the format was "MM/DDCCCC###.##", where the "C's" represent the check's number.

See appendix part B for examples of source code.

Different methods where used to accurately parse each type of transaction. For the deposits and credit card additions, I was able to use two similar "do loops" (one for each transaction type) to find the transactions. The do loop contains "DoEvents" and an "if" statement that allows the loop to avoid an endless loop. In the loops, the agent moves to the "Deposit" or "Corporate ACH Merch Dep" descriptions to find the transactions, then it moves back to the "UnicodeString="" to get ahead of the date and amount of the transaction in the source code. The agent then captures all the text up to the first letter of the description and then adds the text to a variable. Using the "Mid" function, the program captures just the transaction amount and excludes the date. The amount is then added to a variable that is a summation of all the cash or credit card deposits.

After all deposits and credit card additions are captured, the program looks for the expenses. This is accomplished with a third "do loop" with the same safeguards against infinite loops. However, the approach to find the expenses is very different. Because all expenses come after additions, the program must look for all the remaining "UnicodeString="" instances that occur after the additions. However, not every "UnicodeString="" instance is a transaction. Once the agent arrives at a "UnicodeString="" instance, it must check to make sure that a date follows. This is accomplished by looking for two numeric characters followed by a "/", and then two more numeric characters.

Once a transaction is identified, the program must differentiate which ones are checks, and which are not. It does this by capturing (in a variable) all the information between the agent's current position and the "." between the dollars and cents in the amount. It then strips out the date with the mid function. If the variable is greater than the designated characters, then the "Mid" function helps pull out the check number, so that only the transaction amount is captured in the variable. To complete the capture, the program uses the "Mid" function to capture the next two characters (the cents) and ads them to the variable.



When an expense amount is successfully found, the program prompts the user with a user form to classify the expense (see exhibit). The user form also uses option buttons so that only one expense classification can be selected.

Another function of the user form is to display the amount in a text box so the user can change the amount if necessary. For example if a bank statement contains check numbers with different lengths the program may not have captured the right amount. The text box allows the user to correct it.

When the user is done, he clicks the save button. The program then adds that expense amount to the variable that corresponds with that option. The program then moves to the next expense and repeats the process till all expenses are found. If the user clicks "cancel", the program does not add that expense to any of the variables.

Once the program has collected all the information and the transactions are properly classified, it then creates a new worksheet that is formatted with the cash statement. The new sheet is named with a sequential number that corresponds to the statement month. For example, the first sheet will be named "one" - for January. The name of the month will also display in the header. This is accomplished by looking at the name of the page and using "if" statements to select the proper month for the header.

The next step in this phase is to extract the information from the variables and display them in the spreadsheet. The percentages are then calculated along the side of the statement. The program uses an "if" statement to make sure that a number is displayed on a line item before it calculates the percentage. This is to ensure the program does not create errors by dividing by an empty cell (see appendix part c).

The final step is to update the YTD worksheet. The program does this using a "for each" loop. The loop is set to cycle through each worksheet (a variable number of sheets) in the workbook. The loop contains an "if" statement that checks the name of the sheet. If the name is not "YTD", then it selects that sheet and adds each line item to separate corresponding variables. Once the loop has gone through all the sheets and updated all the variables, it then displays the information on the YTD sheet and calculates the percentages the same as on the monthly statements.

The last function of this worksheet is an "if" statement that limits the total number of worksheets to thirteen, twelve months plus the YTD worksheet. If the user attempts to create a 13th month, the program will display a message box that says "This year is complete". Once a year is complete, a new workbook should be created to hold the next year. See Appendix part D.

Learning and Conceptual Difficulties

This project was full of struggles which also prompted many visits to the professor's office. The very first challenge was just exploring the possibility of working with an XPS file. Once I learned that XPS files can be changed into to a zip file and extract its contents the next challenge was to write the code. This

required learning a lot about working with objects. I learned coding that will copy and paste all items from one folder to another. I also learned the name function that allowed me to rename objects.

Working with the agent class was also a struggle. I first tried using it the same way as we did with web assignments and it didn't work. The open page method did not work because I was opening a file instead of internet explorer. Instead, I leaned that a variable could be used to retrieve the source code and then the agent's text was set equal to that variable so the code can be parsed.

Another unanticipated challenged manifested from working with objects. I quickly discovered how tricky it was to keep track of the many objects used and the variables used to manipulate those objects. This became especially difficult when variables where used to control more than one object and those variables changed.

The hardest challenge was finding a logic that would effectively parse the source code. A particularly difficult challenge was overcoming how to parse expenses as they may or may not have a check number associated. The method that is explained earlier was discovered after five or six trial and error attempts. Once I had code that would successfully identify all the transactions with a check number. However, I realized that this method made it impossible to retrieve the rest of the expenses because the agent was past many of the transactions and there was no way to uniquely identify to what point in the source code the agent needed to move back to. Thus I had to scrap hours of work.

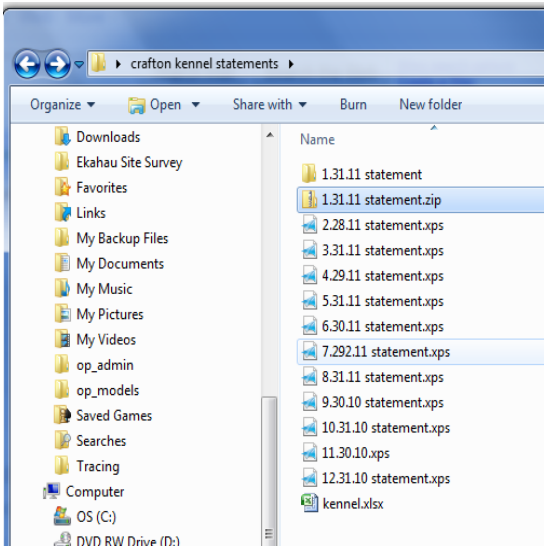
Another significant challenge was my limited knowledge of VBA. I had many times where I knew I had sound logic but didn't understand the tools well enough to execute the logic. For example I spent a significant amount of time trying to loop through all the worksheets to create the variables that update the YTD sheet. I originality created a "for each" loop and in the loop coded "activesheet.select". In my mind when the loop came to a sheet the code would then select that sheet, but it didn't. I realized that I needed to pass a variable to the sheets object that would uniquely identify each sheet. In my case it was the name of the sheet. Once I coded "Sheets(sheetname).select, with "sheetname" being equal to the worksheet name attribute, the code then would select the work sheets and I was able to get the rest of the code to work as expected.

Another challenge I had with my limited knowledge was that I often had a very narrow understanding of how the methods or attributes worked. This limited my creativity on how to use tools that I did know. For example with the agent class I thought the only way to capture text was to use "gettext". But later I learned that the mid function could be used with a variable and it worked better in some situations.

The final thing that I leaned was that once you think you have a completed code, it will probably only work about 75% as expected. When I was doing my final tests I was amazed at how many little problems started arising. Many of these problems may not have been critical to the programs functionality but in aggregate they really affected it. Some problems found where that I had referenced just one cell off, or that I and used clear instead of clear contents and all the formatting for the statement was lost.

Appendix

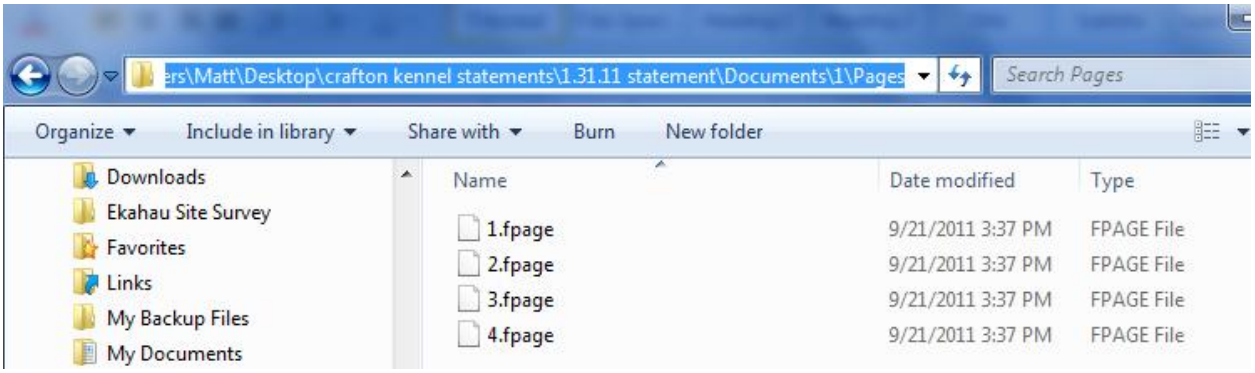
Part A



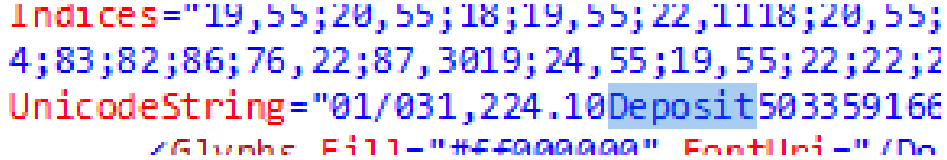
Here you can see that by changing the file extension you can convert the file to a zip format (see the 1.31.11 statement).

The other folder is the extracted folder that VBA creates to gain access to the information on the pages of the bank statement. Once the program has the needed information, the temporary folder is deleted and the zip file is renamed back to an XPS file.

The image below shows the file path that contains the course code information. The "\Documents\1\Pages" part of the file path is added to a string variable which was initially used for creating the temporary folder. This is how the program knows to come here. It then loops through each file using the agent to read the file and add the information from each page to a variable which ends up being a compilation of source code that is added to the agent text and ready to be parsed.



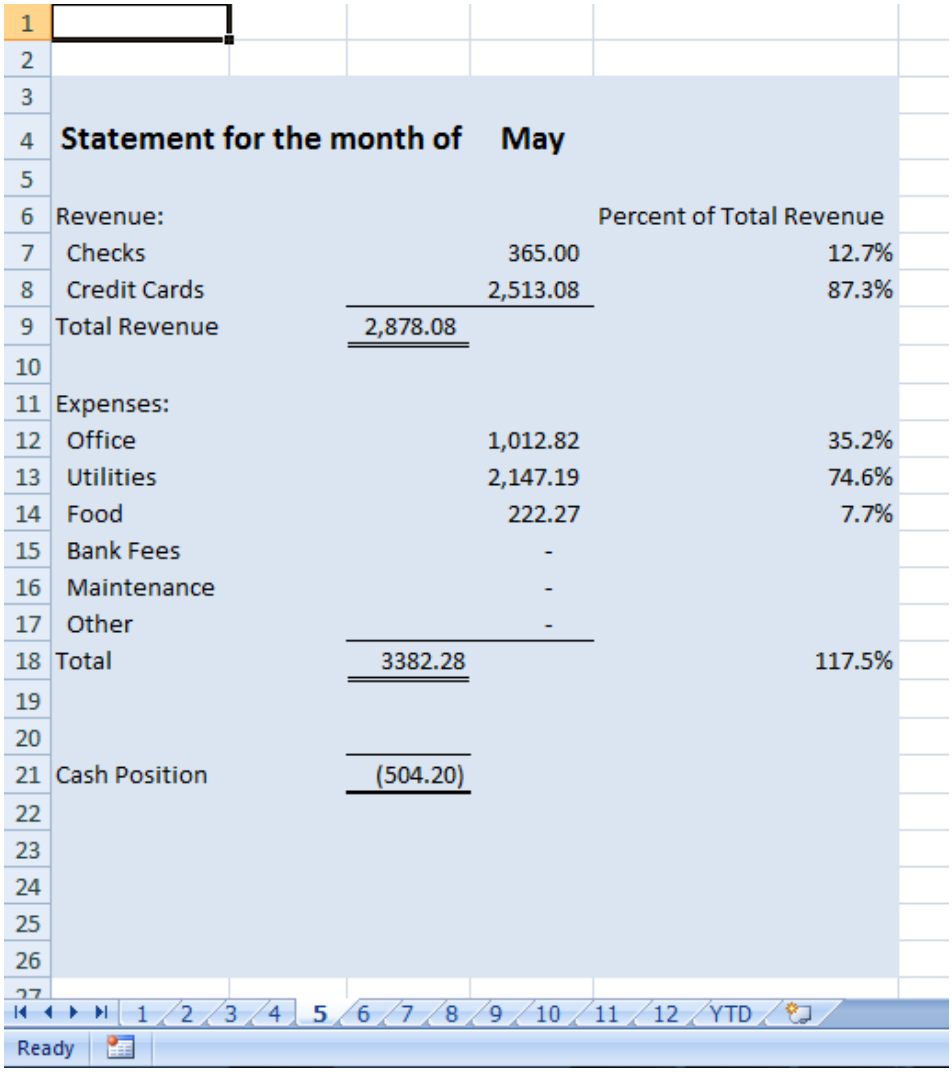
Part B



This image shows an instance of a transaction and how it appears in the source code. Notice that this

transaction was for a deposit and how the descriptions were associated with the transaction. This transaction occurred on January 31 and has the amount of $1,224.10.

Part C

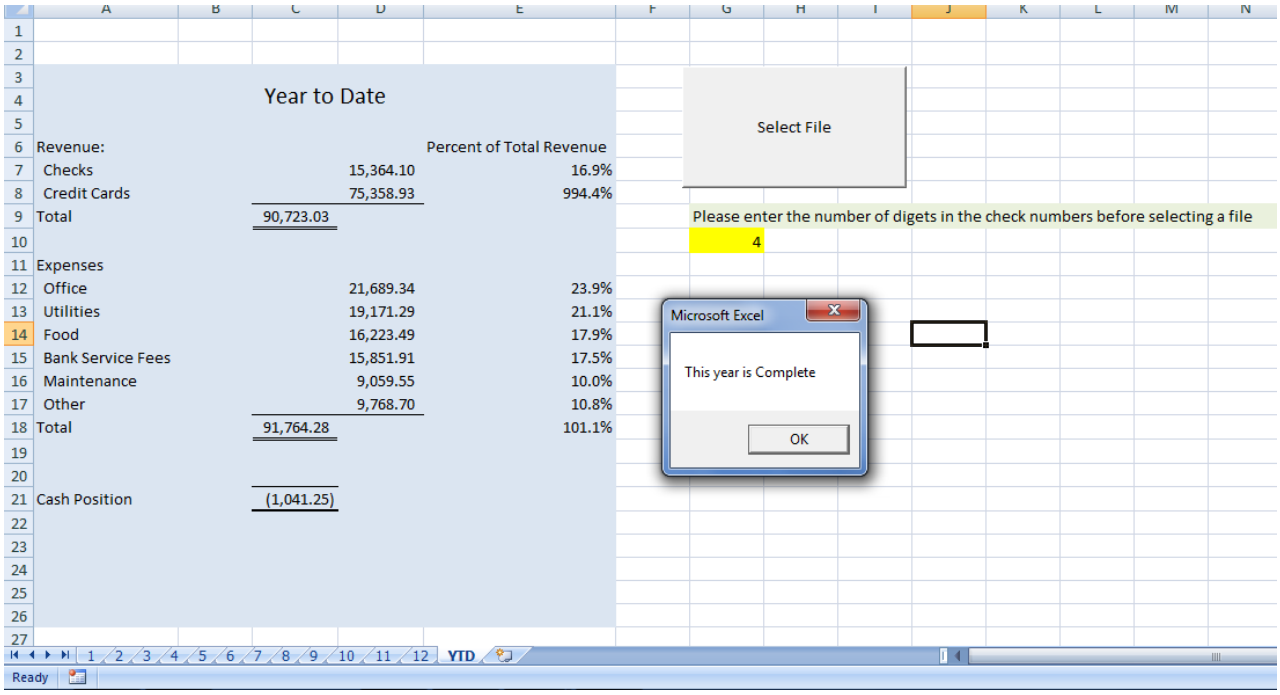


Statement for the month of		**May**	
Revenue:			Percent of Total Revenue
Checks		365.00	12.7%
Credit Cards		2,513.08	87.3%
Total Revenue	2,878.08		
Expenses:			
Office		1,012.82	35.2%
Utilities		2,147.19	74.6%
Food		222.27	7.7%
Bank Fees		-	
Maintenance		-	
Other		-	
Total	3382.28		117.5%
Cash Position	(504.20)		

This image shows the generated cash statement.

Note that the tab is titled “5” to represent May. Also note that the statement header has the name of the month. Notice that this month has no Bank fees, maintenance costs, or other costs. The program did not calculate the percentage for those costs because it would have caused an error (the program avoids dividing into an empty cell).

Part D



This image shows the year to date sheet.

It also displays the message box when a user tries to add more months to a completed year. Note that there are already thirteen worksheets for this to display. Also this sheet contains the button begins the program and the yellow cell where the user inputs the number for the number of digits in the check number.