Extended Functionalities for Automating Comic Book Pull Files

Jackson Cunningham
jncunningham@unomaha.edu

Follow this and additional works at: https://digitalcommons.unomaha.edu/university_honors_program

Part of the Software Engineering Commons

Recommended Citation
Cunningham, Jackson, "Extended Functionalities for Automating Comic Book Pull Files" (2020). Theses/Capstones/Creative Projects. 115.
https://digitalcommons.unomaha.edu/university_honors_program/115

This Dissertation/Thesis is brought to you for free and open access by the University Honors Program at DigitalCommons@UNO. It has been accepted for inclusion in Theses/Capstones/Creative Projects by an authorized administrator of DigitalCommons@UNO. For more information, please contact unodigitalcommons@unomaha.edu.
EXTENDED FUNCTIONALITIES FOR
AUTOMATING COMIC BOOK PULL FILES

by
Jackson Cunningham

Honors Thesis
Mentor: Dr. Harvey Siy

University of Nebraska at Omaha

Honors Department

2020
Abstract

This Honors Thesis project involves programming additional features for a Java application developed as a team for the UNO Computer Science Capstone Project. The purpose of this Capstone project was to develop an updated Pull File system for Dragon’s Lair Comics & Games, which has been running an outdated system in need of improvement. The pull file is an organizational system used by customers to reserve new issues of specific on-going comic book series as they are released. The comic book store can use a pull file system to maintain records of customers, store inventory, and order requests, gaining important statistical data and encouraging repeat customers. To extend this project to meet the criteria of an Honors Thesis project, additional efforts were made on an individual basis to push the application past its functional requirements in three key areas: improved database navigation via an integrated search bar, improved customer interaction via automated emails, and improved statistical readability via PDF table exports.
# Table of Contents

LIST OF FIGURES .............................................................................................................. 4  
GLOSSARY .......................................................................................................................... 5  
BACKGROUND .................................................................................................................. 6  
APPLICATION OVERVIEW .............................................................................................. 9  
SEARCH BAR .................................................................................................................... 13  
AUTOMATED EMAILS ...................................................................................................... 17  
PDF EXPORTS .................................................................................................................. 21  
CONCLUSION .................................................................................................................... 25  
REFERENCES .................................................................................................................... 26
List of Figures

1 ORIGINAL DATABASE ........................................................................................................... 6
2 GitHub Kanban ...................................................................................................................... 9
3 CUSTOMERS WINDOW ....................................................................................................... 10
4 CATALOG WINDOW ........................................................................................................... 11
5 REPORTS WINDOW ........................................................................................................... 12
6 ADD REQUEST SEARCH BAR ............................................................................................ 13
7 SAMPLE AUTOMATED EMAIL ............................................................................................. 20
8 CSV VS. PDF EXports .......................................................................................................... 21
9 MONTHLY BREAKDOWN EXPORT TOGGLE ........................................................................... 25
Glossary

**ACTIONLISTENER**

A piece of code that acts as an event handler, executing a designated chunk of code when a specific event occurs, such as a button being clicked.

**GRAPHICAL USER INTERFACE (GUI)**

An interface that allows the user to interact with a program through visual indicators such as buttons and text boxes rather than exclusively text commands.

**FUNCTION**

A piece of code which accomplishes a specific task when executed. Functions are called by name, and can be passed variables as parameters to be used as input.

**INTEGRATED DEVELOPMENT ENVIRONMENT (IDE)**

Software that combines all necessary code development tools in a single GUI.

**JAVA**

A general-purpose object-oriented programming language commonly used for developing applications

**METHOD**

Serve a similar purpose as Functions, with the difference being that Methods are associated with a specific object in the program.

**STRUCTURED QUERY LANGUAGE (SQL)**

A programming language used for altering and retrieving data from database management systems.

**VIRTUAL MACHINE**

Software that emulates an Operating System, allowing a computer to simulate a separate computer system within an application window.
Background

This Honors Thesis project acts as an extension of the Computer Science Capstone Project, the final project of the UNO Computer Science degree program. In the Capstone Project a team of students majoring in Computer Science-related fields work together with a third-party client in order to solve a real-world problem using skills learned over the course of their degree program. In this case, the client in question was Dragon’s Lair Comics & Games, a comic book shop local to the Omaha area. Dragon’s Lair reached out to the Capstone program due to a need for an overhaul of their archaic pull file system.

A pull file is a system used by comic book shops for the purpose of maintaining organized datasets relating to customers their comic book requests. From a customer’s perspective, creating a pull file allows a customer to make a list of the comics series they are interested in, and as new issues are released they will be added to a file for the purpose of being purchased at a later date. The pull file essentially ensures that a customer will not risk missing out on an issue that sells out quickly, and provides an easy system for collecting all wanted comics in a single location.
From a comic shop manager’s perspective, the pull file system provides invaluable customer information and incentivizes customers to convert individual purchases to repeat business as future issues are released at a steady increment. Using a pull file system, a comics shop can maintain a list of all repeat customers with their contact information, a list of all of the store’s inventory, and list individual orders of comic titles by specific customers. The system Dragon's Lair is currently using is a decades-old DOS-based program which is functional, but not efficient or user-friendly.

When Dragon’s Lair approached Capstone with this project, they provided a list of nine goals, which needed to be handled by the final product:

1. Maintain a list of Customers with their contact information (phone number, email, etc.)
2. Maintain a list of Comic Book Titles which includes new items, ongoing titles, and one-offs
3. Maintain a list of Orders which will show as Customer gets Title in x Quantity
4. Manage the weekly task of Flagging New Releases
5. Export a list of a Single Comic Book Titles’ Orders with Customers and Quantities sorted by last name
6. Export a list of a Single Customer with all Orders and Quantities sorted by Title
7. Export a list of all Flagged New Release Orders sorted by Comic Book Title with Customers and Quantities under each title sorted by last name
8. Export a list of all Comic Book Titles’ Orders with only Quantities sorted by Title
9. Have a clear distinction between which location a particular Customer or Order is for

Building a system which accomplished these nine tasks in a more efficient and approachable manner than the original system was the basis of the Capstone project which was accomplished by a group
of multiple students. In order to extend this project to meet the standards of an Honors Thesis project, additional tasks were executed on an individual level that pushed the final product past the functional requirements in a few key ways. First, it became clear over the course of the development process that a method for quickly locating individual comic book titles was crucial to improving the usability of the application. This became the first Thesis task executed without group aid. Second, the client had included as a bonus wish-list task, in addition to the mail goals, having the ability to send automatic emails to customers based on the status of their pull list. This was undertaken as an additional individual Thesis task. Third, as speed and accessibility were the primary impetus behind Dragon’s Lair’s need for system improvement, an extra effort was made to improve the readability of data exports, providing the ability to export data in organized PDF tables rather than only the raw CSV (Excel) files. These three tasks represent efforts to expand the base application in distinct key areas, requiring additional research and development time to bring them to fruition.
Application Overview

The new Pull File application was developed as a Java application, tied into a remote SQL database accessed via SQL Server Management Studio (SSMS), an application used for managing Microsoft SQL Servers. SSMS is exclusive to the Windows platform, so VMware Fusion was used as a Virtual Machine for accessing the Microsoft SQL Server on a MacBook computer. The pull file application’s Graphical User Interface (GUI) itself was constructed using both the Eclipse and IntelliJ Integrated Development Environments (IDEs), which allow all necessary code and additional components to be maintained in a clean, singular environment. The IDEs were tied into a GitHub Repository, which is a Version Control System (VCS) used for allowing multiple people to edit a single program by storing and integrating different versions throughout the development process. For group organization, a GitHub Kanban board was used, allowing tasks to be created in the ‘To-Do’ column, and shifted over to ‘In Progress’ and ‘Done’ as progress was made. This system allowed the group to break down larger goals into manageable steps and maintain perspective of the larger project.

![Fig 2. The GitHub Kanban project board used for Capstone task organization](image)

The pull file application itself is composed of a number of menu screens which provide tables for viewing data pulled from the SQL database, and allowing the user to edit stored data. The home
screen for the application is the Customers window, which displays information regarding customers and their individual pull files.

![Customers window](image)

Fig 3. The Customers window which acts as the homepage for the updated pull file application

On the left-hand side of the Customers page, a table displays a list of all customers that have been entered into the store’s database, along with their phone numbers and email addresses. A panel in the top right corner spreads this information out in a more accessible format, filling in information regarding a customer selected from the table at left. The bottom right corner of the Customers page is used for customer orders. When a customer is selected from the table at left, the titles which they have requested are displayed in the orders table. Buttons around the top and right borders of the window allow the user to add or delete customers, edit customer contact information, add, delete, or edit pull requests, email customers, and export table data.
The second page of the application is the Catalog menu, where the available comic book titles can be viewed, and the weekly process of flagging new releases is completed. The left half of this menu shows a list of all comic book titles held in the database, and the right half displays all information regarding a selected title, such as distributor, release date, and Catalog ID number. Buttons along the top border allow the user to add, delete, or edit titles, and filter the shown titles between ‘All’ and ‘Near’, which only shows titles which have come out recently. Additionally, this menu allows the user to import multiple titles from a CSV file, as received from a comic distributor.
Finally, the Reports page allows the user to access and export statistics based on recently added and requested issues. The ‘New Week Polls’ tab allows the user to view information corresponding to data changes made in the current week, and the ‘Monthly Breakdown’ tab relates to both titles requested over the last month, and overall statistics from the database. On the Monthly Breakdown tab, buttons allow the user to export data pertaining to all titles, all customers, and special orders, titles which have not had new issues in the last six months, and titles which currently have zero customer requests.

With the underlying functionalities of these three pages constructed as a group, the additional individual tasks were undergone as an extension. From the first Customers window, the search bar was added to the ‘Add Requests’ button, and the automatic customer emails functionality was created as its own button, and as an option when exiting the ‘Add Requests’ menu. PDF Exports were first added as an individual button on the Customers window, and then integrated into the Monthly Breakdown export buttons via a radio button toggle option on the Reports window (Chitranaya).
Search Bar

After the majority of the pull list application had been developed, it became clear that the growth of the database Catalog table over time would become an issue. Every week Dragon’s Lair receives a listing of the newly released comic issues from their publishers, detailing all of the new comics coming into stock. As a wide array of new comic titles are constantly being launched, the list of comics that customers can request will continually grow. From a back-end perspective, after a significant period of time passes the application could begin to slow, as each database query retrieving title data would have to search through a larger and larger list of titles. However, a more immediate concern was the way in which the pull list application’s user would interact with this titles list. The method by which the user created a new pull request involved selecting a title from a dropdown menu of all titles in the Catalog table, alphabetically sorted. The list of sample data being used for initial testing during the design phase was already over 1500 titles, and would certainly grow into the thousands when put into use – asking the user to scroll through a list of thousands of titles would not be even remotely feasible. In order to create a more streamlined process for the user, an extra effort was made to implement a search function so that the overall list could be narrowed to a more manageable level.

Fig 6. The ‘Add Request’ search bar, showing the expanded titles dropdown menu
At face value, the concept of a search bar appears extremely simple – they are used everywhere and are often taken as a given, as the more complicated process that constructs them never needs to be considered by the user. It was found through research that the process of constructing a search bar can essentially be broken down into three distinct steps, each representing a different aspect of the application’s design: GUI representation, SQL database querying, and Model refreshing.

The first step in constructing a search function is setting up the Graphical User Interface (GUI), where the user will actually interact with the program. The GUI is coded in Java, using Java’s JLabel, JTextField, and JComboBox components to represent the label, text field, and dropdown box on the application screen. This initial process is fairly simple, as constructing GUI elements is done using the same process and pre-defined elements as every other visual in the application. The label is defined and set to display the “Titles” caption above the text field on the ‘Add Request’ menu, the text field is displayed under the label, and the dropdown menu is displayed just below the text field. These GUI components are defined using the following code:

```java
JLabel titleLabel = new JLabel("Title");
titleLabel.setFont(new Font("Tahoma", Font.BOLD, 14));
titleLabel.setBounds(214, 25, 78, 23);
addRequestPanel.add(titleLabel);

JTextField titleFilterField = new JTextField();
titleFilterField.setColumns(10);
titleFilterField.setBounds(214, 48, 525, 20);
addRequestPanel.add(titleFilterField);
addRequestFrame.add(addRequestPanel);

JComboBox<?> titleField = new JComboBox<Object>(titles);
titleField.setBounds(214, 72, 525, 20);
titleField.setSelectedIndex(0);
addRequestPanel.add(titleField);
```

Initially, this dropdown menu (or combo box, as it is referred to in programming documentation) is filled with an alphabetical list of all comic book titles from the Catalog table. This list, called ‘titles’, is filled using the following code, which calls the getIndividualTitles() method from the controller class:
In the controller, the `getIndividualTitles()` method is defined as follows:

```java
public String[][] getIndividualTitles(String filter) {
    if (filter == null) {
        return select(String.format("SELECT DISTINCT [Series] FROM [newDLC].[dbo].[Catalog]");
    } else {
        return select(String.format("SELECT DISTINCT [Series] FROM [newDLC].[dbo].[Catalog] WHERE [Series] Like '%%" + filter + "%%'");
    }
}
```

This method reads the value of the ‘Add Request’ text field, and pulls the corresponding data from the Catalog. When the user first opens the ‘Add Request’ menu, the text field is empty (null), and as such, `getIndividualTitles()` retrieves a list of all comic titles from the database using the SQL query, “SELECT DISTINCT [Series] FROM [newDLC].[dbo].[Catalog]”, which returns the name of every series in the Catalog table. When the user types some characters into the text field and hits ‘Enter’, `getIndividualTitles()` instead runs the SQL query, “SELECT DISTINCT [Series] FROM [newDLC].[dbo].[Catalog] WHERE [Series] Like ‘%%’ + filter + ‘%%’”. That additional WHERE clause is what causes the title list to be filtered. Whatever text entered by the user is saved and passed to `getIndividualTitles()` as the ‘filter’ variable, and the WHERE clause uses this value by selecting all titles in Catalog which contain the entered value. For example, if the user types in “Spider-Man”, this function would return a list of all titles containing the phrase “Spider-Man”. Additionally, typing simply “Man” would return “Spider-Man”, “Batman”, “Manhunter”, etc.

Once the visual aspect of the search bar has been constructed, and the SQL query has been defined, the functionality of updating the combo box with only the requested titles must be written. This third part of the process was accomplished via the following code:
First, an ActionListener event is defined for the text field (How to). This means that when an event occurs in the text field (i.e., the ‘Enter’ key being pressed when the text field is currently selected), a defined chunk of code will be executed. Inside this event, the code for refreshing the combo box is written. Here, a new filtered list of titles is pulled from the database and stored in the String array variable called ‘titlesFilter’. Next, a DefaultComboBoxModel is defined based on the titlesFilter list.

A Model is essentially a data structure used to hold a specific kind of data which is to be referenced by another structure. In this case, the DefaultComboBoxModel is used to represent a new dataset to act as the basis of a combo box. Running “titleField.setModel(model)” resets the dataset being displayed in the dropdown menu with only the data found inside the defined model. Similarly, Models were used for all tables displayed on the app, as updating the data directly without using a Model would not immediately update the data on screen – without a Model, the application would need to be closed and re-opened to display any changes made. By combining the visuals defined by Java components, data retrieved by SQL queries, and the refreshing system powered by the DefaultComboBoxModel, a working search bar was built, allowing the user to retrieve a specific comic by title almost instantly, removing the needing to first wade through thousands of titles.
Automated Emails

When the customer first outlined their requirements for the updated pull list application, they listed numerous specific tasks which had to be addressed in order for the product to suit their needs. In addition to these musts, they provided a few bonus requirements which would take a bit more work and were not integral to the functioning of the application, but would make the program even more useful. One of such bonus requests was having the ability to send automated emails to customers notifying them of comic titles in their pull lists. This bonus request was tackled as one of the extended functionalities for this Honors Thesis.

The process of creating an automated email in Java can be broken into numerous steps. First, the ultimate goal of the email needed to be defined. For this project, it was decided that the user of the application would have the ability to send an automated email to a customer at the push of a button, sending them a list of all comic titles currently in their pull list. Additionally, whenever the user adds or removes titles from a pull list, they would be prompted with the option to notify the customer of this change via email.

After the functional definition of the email automation was finalized, the first step was to retrieve all data required for the email: the customer’s email address, the list of titles in their pull list, the text for the subject of the email, and the text for the body of the email. In the Customers menu of the pull list application, a list box displaying all Customers is shown on the left, and a ‘Customer Details’ section in the top right displays the name, phone number, and email address of a customer selected from the list at the left. The text field in ‘Customer Details’ corresponding to a customer’s email address can be accessed by the email function to retrieve the recipient’s email address. The list of titles in this customer’s pull list is retrieved in a similar manner to how the Search Bar retrieves a list of all comic
titles, only using the getCustomerOrders() method instead of getIndividualTitles().

getCustomerOrders() is defined as follows:

```java
public String[] getCustomerOrders(int customerCode) {
    String[][] orders2D = select(String.format("SELECT [Title] FROM [newDLC].[dbo].[Order] WHERE [Customer Code] = " + customerCode));
    String[] orders = new String[orders2D.length];
    for (int i = 0; i < orders2D.length; i++) {
        orders[i] = orders2D[i][0];
    }
    return orders;
}
```

This method is passed the Customer ID of the selected customer as a parameter (‘customerCode’), and runs the SQL query, “SELECT [Title] FROM [newDLC].[dbo].[Order] WHERE [Customer Code] = ” + customerCode. This query returns a list of all titles from the Order table which correspond to the selected Customer ID.

In order to create a full email template automatically, an object known as a ‘mailto link’ is created. A mailto link is essentially a string of characters containing all data required for an email in one condensed location, separated by special characters used to differentiate between elements. Building a mailto requires gathering all of the disparate elements separately, then combining them into one long string to be converted. The following Java code collects these elements and combines them into the full mailto link, ‘mailString’:

```java
public void emailCustomer(JFrame frame) {
    int result = JOptionPane.showConfirmDialog(frame, "Email Customer?", "Email Customer?", JOptionPane.YES_NO_OPTION, JOptionPane.QUESTION_MESSAGE);
    if (result == JOptionPane.YES_OPTION) {
        try {
            String[] requests = control.getCustomerOrders(Integer.parseInt(ccodeBox.getText()));
            String mailString = "mailto:";
            String subject = "Dragon's Lair Pull List Update";
            String toEmail = emailBox.getText();
            String body = "Hello " + fNameBox.getText() + " " + lNameBox.getText() + "%2C%0A%0A%0A";
            body = body + "The following titles are currently in your Pull List:\n\n\n"; //The following titles are currently in your Pull List: 
```
In mailto links, symbols such as apostrophes and colons are encoded using representative strings led by percent signs, such as ‘%27’ and ‘%3A’, respectively (HTML URL). In this way, the subject line defined above as “Dragon%27s%20Lair%20Pull%20List%20Update” translates to “Dragon’s Lair Pull List Update”. The body text of the email is coded in the same way after the list of titles is prepared. As an example, the mailto link:

```
mailto:exampleEmail@unomaha.edu?subject=Dragon%27s%20Lair%20Pull%20List%20Update&body=Hello%20Jackson%20Cunningham%2C%0A%0AThe%20following%20titles%20are%20in%20your%20Pull%20List%20 currently%20in%20your%20Pull%20List%3A%0A%0AACTION%20COMICS%0A%0A%0AIMMORTAL%0AHULK%0A%0AVENOM%0A
```

is used to create the following email:
The above image shows a screenshot taken of an email created in Apple's Mail application, as the automated email function was created using MacBook Pro. However, Dragon’s Lair exclusively uses computers running the Windows operating system. To account for this discrepancy, the emailing function was created in such a way that accommodates all systems. When actually processing the mailto link, the emailCustomer() method calls “Desktop.getDesktop()”, a function from the imported java.awt.Desktop class (Class Desktop), in order to determine the nature of the system on which the program is being run. “Desktop.mail(mailto)” then transfers the prepared mailto link to whichever email application has been made default for the system in question. In this way, the email can be generated on both Mac and PC computers.
PDF Exports

One of the main purposes the client had for overhauling their system was needing their flagging process to be made more accessible. One way in which this was accomplished was by implementing the ability to export data tables into an easily printable PDF format. A major purpose of the pull list is customer benefit – customers don’t need to worry about missing out on new comics if they are put on hold as they are released. Another major benefit of using pull list systems is for business analytics. By compiling lists of ongoing titles each customer has requested, the comic shop can identify which titles are most popular and order in appropriate quantities. In order to effectively analyze the pull list statistics, processes need to be in place to export data based on specified criteria. Exporting from the database to a CSV file is a fairly simple method for quickly exporting data, but reading raw data can be a bit difficult. Adding the additional ability to export to PDF allows for the formatting of data into more easily read and more importantly, more easily printed tables.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Last Name</td>
<td>First Name</td>
<td>Phone</td>
<td>Email</td>
<td>Number</td>
<td>Email Address</td>
</tr>
<tr>
<td>Cunningham</td>
<td>Jackson</td>
<td>(402)111-1111</td>
<td><a href="mailto:exampleEmail@unomaha.edu">exampleEmail@unomaha.edu</a></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wayne</td>
<td>Bruce</td>
<td>(402)111-1111</td>
<td><a href="mailto:notBatman@gmail.com">notBatman@gmail.com</a></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shrek</td>
<td>Jesse</td>
<td>(402)<a href="mailto:545-5J_schrack@gmail.com">545-5J_schrack@gmail.com</a></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jenkins</td>
<td>Billyyyy</td>
<td>(111)111-1 <a href="mailto:bjenk@gmail.com">bjenk@gmail.com</a></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sortino</td>
<td>Logan</td>
<td>(402)111-1323</td>
<td><a href="mailto:lsortino@unomaha.edu">lsortino@unomaha.edu</a></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maxwell</td>
<td>Joseph</td>
<td>(402)936-4081</td>
<td><a href="mailto:joemax94@comcast.net">joemax94@comcast.net</a></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Name2</td>
<td>Test</td>
<td>(402)224-3184</td>
<td><a href="mailto:myemail@aol.com">myemail@aol.com</a></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The first PDF export process written was for the purpose of exporting a list of all customers in the pull list database. First, a button was set up on the Customers menu to trigger the export event, using the following code:

```java
exportCustomersBtn = new JButton("Export PDF");
exportCustomersBtn.setFont(new Font("Tahoma", Font.PLAIN, 13));
exportCustomersBtn.set Bounds(860, 555, 107, 48);
exportCustomersBtn.setEn abled(true);
add(exportCustomersBtn);
```
When this button is clicked, it triggers the code associated with the `exportCustomersBtn` `ActionListener`, defined as follows:

```java
exportCustomersBtn.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        if (exportCustomersBtn.isEnabled()) {
            exportCustomersBtn.setEnabled(true);
            Document pdfDoc = new Document();
            // Create new File
            String filePath = control.saveFile(customerDetails, "Customers_List", ".pdf");
            File file = new File(filePath);
            try {
                file.createNewFile();
            } catch (IOException ioException) {
                ioException.printStackTrace();
            }
            FileOutputStream fop = null;
            try {
                fop = new FileOutputStream(file);
            } catch (FileNotFoundException fileNotFoundException) {
                fileNotFoundException.printStackTrace();
            }
            try {
                PdfWriter.getInstance(pdfDoc, fop);
            } catch (DocumentException documentException) {
                documentException.printStackTrace();
            }
            pdfDoc.open();
            String[] header = new String[] {"Last Name", "First Name", "Phone Number", "Email Address", "ID"};
            String[][] body = control.getCustomers();
            // Table for header
            PdfPTable pdfTableHeader = new PdfPTable(header.length);
            for (int j = 0; j < header.length; j++) {
                Phrase frase = new Phrase(header[j]);
                PdfPCell cell = new PdfPCell(frase);
                cell.setBackgroundColor(new BaseColor(Color.lightGray.getRGB()));
                pdfTableHeader.addCell(cell);
            }
            try {
                pdfTableHeader.setWidths(new float[]{2, 2, 2, 4, 1});
            } catch (DocumentException documentException) {
                documentException.printStackTrace();
            }
            try {
                pdfDoc.add(pdfTableHeader);
            } catch (DocumentException documentException) {
                documentException.printStackTrace();
            }
            // Table for body
            PdfPTable pdfTable = new PdfPTable(header.length);
            for (int i = 0; i < body.length; i++) {
                for (int j = 0; j < body[i].length; j++) {
                    pdfTable.addCell(new Phrase(body[i][j]));
                }
            }
            try {
                pdfTable.setWidths(new float[]{2, 2, 2, 4, 1});
            } catch (DocumentException documentException) {
                documentException.printStackTrace();
            }
        }
    }
});
```
When this ActionListener event is triggered, a pop-up menu appears allowing the user to select a location for the exported PDF to be saved (How to). A file, called ‘file’, is created with the end location of the user-selected path, and a FileOutputStream, an object which receives data and sends it to a defined location, is created to allow data to be written to the created file (Class File). For the purpose of writing out data to a PDF, the com.itextpdf.text library was imported (iText PDF). This library provides tools that allow the creation of PDF files in java (PDF Export). A PdfWriter is an object defined by the itextpdf library which allows editing pdfs. After creating the FileOutputStream, the export function creates a PdfWriter object and links it to the FileOutputStream, so that a created pdf can be saved to the file at the specified location.

After all of this set-up has been completed, the PDF export function begins designing the table to be printed on the PDF. Using the itextpdf.text.pdf library, a PdfPTable is created, which is a table that can be printed to a PDF. This table is defined with background and header colors for easy readability. For this Customer table, the column headers are defined with the values “Last Name”, “First Name”, “Phone Number”, “Email Address”, and “ID”. The actual customer data is collected through the
getCustomers() method, which pulls all customer data corresponding to the table columns. The getCustomers() method is defined as follows:

```java
public String[][] getCustomers() {
    return select("SELECT [Last Name], [First Name], [Phone #1], [Email], [Customer Code] FROM [newDLC].[dbo].[Customer]");
}
```

Once the data has been collected, it is fed into the table row by row. The column widths are adjusted to give more space to fields with longer values, and the finalized table is written to the PDF. All of the created files are closed, and the newly-written pdf is saved to the specified location for easy access.

After creating this template for PDF exports using the Customers table, the process was repeated for the other criteria requested by the client: all Titles, the number of pending issue requests per title, the number of titles that have not had a new release in over 6 months, and titles with zero customer requests. Since these exports had already been written for the CSV format, a radio button selector was created to allow the user to select filetype using the following code:

```java
JRadioButton csv = new JRadioButton("CSV");
JRadioButton pdf = new JRadioButton("PDF");
csv.setBounds(869, 383, 89, 23);
pdf.setBounds(869, 418, 89, 23);
ButtonGroup bg = new ButtonGroup();
bG.add(csv);
bG.add(pdf);
csv.setSelected(true);
monthly_breakdown.add(csv);
monthly_breakdown.add(pdf);
```

This code creates two buttons (one for CSV and one for PDF), and a button group, which ensures that only one button can be selected at a time. When the CSV button is selected, the CSV export process is run. When the PDF button is selected, an else-statement runs the PDF export process instead.
Fig 9. The Monthly Breakdown export buttons with CSV and PDF radio button options

**Conclusion**

Over the course of this Honors Thesis project, numerous new skills were developed and applied, using the Capstone project as a jumping point for additional individual growth. Through researching and constructing code in new areas, the pull file application was able to accommodate a number of new purposes that it wasn’t originally fulfilling. Tangential fields such as email automation, GUI design, Database Management System maintenance, and data visualization were all explored through this Honors Thesis project, and this more in-depth, specific research allowed for growth in all aspects of computer science that will be beneficial in pursuing a career in the Computer Science field.
References


“How to Write an Action Listener.” Oracle Java Documentation, Oracle, docs.oracle.com/javase/tutorial/uiswing/events/actionlistener.html.

HTML URL Encoding Reference, Refsnes Data, www.w3schools.com/tags/ref_urlencode.ASP.
