

Prop Search, Hire and Management Program

Candidate Name: Matthew Cook

Candidate Number: 6437

Centre Number: 12460

Centre Name: Twyford COE High School

Table of Contents

Analysis	6
Introduction	6
Background and Problem Identification:	6
Investigation	7
Research Methods:	7
Interview With The Client To Clarify How The Current System Works:	7
Interview To Determine Improvements:	8
Interview With Colleagues to Clarify The Issues With The Current System:	9
Survey Questions:	10
Current System:	15
Matrix of User Roles and Responsibilities:	16
Matrix of Actions Performed by Job Roles:	16
Data Sources and Destinations:	16
Conversion to New System	17
Analysis Data Dictionary:	17
Current System Flow Charts:	20
Algorithms:	23
Objectives:	24
Objective 1:	24
Objective 2:	26
Objective 3:	27
Objective 4:	28
Objective 5:	29
Project Limitations & Constraints:	29
Data Volumes:	29
System Considerations:	30
Limitations:	31
Critical Path:	31
Gantt Chart:	31
Design:	32
Project Overview and Problem Deconstruction:	32
Structure / Hierarchy chart:	32

Hiring a Prop Flow Chart:	33
Login Flow Chart:	34
Login Pseudocode:	35
Prop Return Flow Chart:	36
Recommendation Flow Chart:	37
Basket Flow Chart:	38
Information Transfer Diagram:	39
Database Normalisation:	39
0NF:	39
1NF:	40
2NF:	40
3NF:	41
Entity Relationship Diagram:	42
Creating database tables:	42
Prop Table:	42
Employee Table:	43
Orders Table:	43
Shoot Location Table:	44
Order Details Table:	44
UI Design:	45
Prop Search Page – Phase 1	45
Prop Search Page – Phase 2	46
Prop Details Page Design:	47
Login Page Design:	48
UI and UX Interview With Client:	49
Class Diagram:	50
Testing Table:	51
Pseudocode:	55
Item Hire Pseudocode:	55
Recommendation System Pseudocode:	56
Technical Solution:	57
Objective 1 Login Page:	57
Objective 1 Login Page:	58

Objective 1 Account Creation Page:.....	59
Objective 1 Hash Function:	60
Objective 1: SQL Account Creation.....	61
Agile Methodology for Objective 1:.....	62
Objective 2 Search functions:	64
Objective 2 Search Functions:	65
Objective 2 SQL Functions:	67
Objective 2 Detail Page:.....	68
Objective 2 Add order:	71
Objective 2 Convert Email to Orders:	73
Agile Methodology for Objective 2:.....	74
Objective 3 Recommendation System:	76
Objective 3 Display Recommendations:.....	77
Objective 3 Generate Recommendations:.....	77
Objective 3 Number of Props:	80
Objective 3 List Functions:	81
Objective 3 Retrieve Prop Details:	83
Agile Methodology for Objective 3:.....	85
Objective 4 basket system:.....	86
Objective 4 Checking Duplicates:.....	87
Objective 4 Hire Basket Button:	88
Objective 4 View and Edit Basket:	89
Agile Methodology for Objective 4:.....	90
Objective 5 Prop Management:	92
Objective 5 Hire Prop Button:.....	93
Objective 5 Returning Props:	93
Agile methodology for objective 5	94
Testing:.....	96
Search Function:	96
Account Creation:.....	102
Hire Function:	105
Prop Detail Page:	108
Recommendation System:.....	110

Basket System:	112
Add Prop Function:.....	116
Evaluation:	118
Evaluation Table:.....	118
Objective 1	118
Objective 2.....	118
Objective 3.....	119
Objective 4.....	119
Objective 5.....	120
Stake Holder Feedback:	121
Main Client (Set decorator)	121
Other Stakeholder (prop master).....	121
Possible Future Extensions for the Program and Improvements:.....	122
Appendix:.....	123
UI Manager Class:.....	123
Add Prop Class:	124
Add Existing Prop Class:	128
Basket Manager Class:	130
Create Account Class:.....	132
Detail Manager Class:	137
Hash Function Class:	141
Login Page Class:	141
Recommendation System Class:.....	144
Reverse List Class:.....	148
Search Page Class:.....	150

Analysis

Introduction

Background and Problem Identification:

My client is Mrs Harvey who works as a set designer which means that she frequently needs to organise the purchase and use of furniture for decorating scenes on Netflix productions. These pieces of furniture are referred to as props and are used on scenes, which are also called sets. The props which she uses to decorate the sets are all stored in a warehouse which is owned by Netflix. While my client sometimes does have to hire props from external companies such as Super Hire, the process for rental from those companies is not going to be covered within this project. The props which my client selects are then removed from the warehouse and transported to the location to be used in the sets. The management and storage of these props is what I hope to improve upon through this project.

The current process for hiring props from the Netflix warehouse is confusing and inefficient. It starts with my client walking around the unorganised warehouse to find items which they may need for the current set, this can range from paintings to chairs. They will then send a list of the props they want via email to the prop master (the employee responsible for the transfer and management of the prop warehouse) who transports the props from the warehouse to the location where the scenes will be shot. This is all done without any formal transfer of information such as records as there is no central computer system.

The issue with the current system is that it is inefficient and leads to errors which impacts almost everyone who works at the prop house. These issues affect both the set decorators who have to try and find the props and the prop master who has to transport the props. This is because the props are often hard to find and difficult to track, in addition there is also confusion about who wants to hire what, and for which set. Overall, this leads to problems for all and can often result in props going missing, costing Netflix money constantly.

The main problem which my client hopes for me to resolve is the process of hiring. This is solved through improving the amount of information available. Currently it is difficult to determine which props are available, how long they are hired for and where they are. The aim is to improve the whole process through a database which stores the characteristics of the props and their current status. The process for transporting the props to and from the sets seems to be already streamlined so this project will focus on the issues surrounding the warehouse instead.

Investigation

Research Methods:

I believe that the most suitable form of collecting information would be to interview due to the level of detail it provides the client when explaining exactly what could be improved. In terms of having a meeting with the broader team I believe this would not be suitable due to their busy schedule and strict Covid restrictions. However, sending a questionnaire may be suitable due to the large team who would use the system, by having a questionnaire I would be able to gather an average of their responses.

Interview With the Client To Clarify How The Current System Works:

This interview was able to give me insight into how my client views the current system and the flaws which need to be improved upon.

1. What is the process for hiring a prop?

Typically, the set decorators first walk around the warehouse to find the props they want. Photos are then taken of each one and sent to the prop-master who finds them and collects them to be sent to the correct set. They are also returned to the warehouse by the prop master after filming is done, roughly a week after hire.

2.Are the props organised in a clear way?

There is a loose form of organisation such as there is an area for smaller items and an area for sets of chairs. However, there is no formal system such as having labelled rows and columns which makes finding a specific item sometimes more difficult and time consuming.

3. How are the props transferred from the prop house to the location?

Once prop master has got all the props they are placed in a large group and loaded into a van for transport. I believe that the system for transporting the props is already operational and functional so I don't think we should focus on that during this project.

4. What are the roles in the department?

I am the head set decorator, so I have a team of other set decorators which I organise. They perform similar jobs such as finding props and dressing sets. There is also a prop master who manages the prop men, this is a team of employees who organise the warehouse and track the

inventory of items in the warehouse, I think that this project will also improve their workflow by making it more organised and easier to track.

5. What are the issues with the current system?

From the perspective of a set decorator the main issue is when several people select the same prop for different sets. This causes a clash because shooting often takes place on similar dates which means that one of them have to compromise. Currently we resolve this by using coloured stickers so that it is clear who wants which item, however this could be improved in the digital system. In addition, it is hard to know which props are available due to the size of the warehouse which makes finding items hard, by creating a search system it may make the process easier.

Interview To Determine Improvements:

These questions were used to determine how the new system would improve upon the current process:

1. Which elements would need to be improved in the new system?

I think that the ability to see the props, their details and whether they are available to hire without having to walk around the warehouse would be the biggest improvement seen through the implementation of a new system.

2. Do you think a barcode system would improve the workflow of hiring props?

I think the idea of it is good but I'm not sure it would work in practice due to how often the props are handled and transported. Also, the props are used for filming which might mean that they would need to be removed which could make tracking the items difficult if the barcodes are frequently removed.

3. Are there any new parts which could be added?

The addition of a recommendation system may be helpful since I often dress sets with a thematic colour so being recommended related items could allow for a more efficient process. I enjoy the system used for other sites such as amazon so something like that would be great.

4. What is currently the most frustrating part of work?

Probably when there is a clash where you attempt to hire a prop which is already going to be used for another shoot. It wastes time and leads to compromises being made since the 2 set decorators have to discuss who should use the prop. Maybe when browsing the items in the new system, there could be a way of telling if the item was available.

5. How could the system be improved for the prop master?

By having a formal order system it would make the tracking of items easier since currently there is nothing stored about which props are in the warehouse. In addition, by having a database it could make the tracking of items easier since it would store whether the item is there and how long it will be gone.

Interview With Colleagues to Clarify The Issues With The Current System:

While the previous interview with my client allowed me to understand the current system, it only provided the perspective of a set decorator rather than the prop master who would arguably be equally impacted by the project. Therefore, for this interview I decided to talk to Owen Harrison who is responsible for the organisation and transport of the props.

1. What is the largest issue you face on a regular basis?

I think that the most annoying thing is the tracking of which props set decorators want. The sending of emails means that it's hard to update and make changes, it also leads to a lot of messages being lost in a chain of emails. I think that having a separate method of documenting the orders would help make things clear.

2. Do you find it hard to locate the props in the warehouse?

Since it is often my team replacing the props in the warehouse it is often not hard to find the props, but I can see how it may be difficult for the set decorators who are less involved in the process.

3. Is there a record of what is currently being stored in the warehouse?

There isn't really a method of recording which props are available, it normally isn't an issue since we can look to see if there is anything missing. However, there are sometimes problems with missing props when departments can't communicate effectively

4. Would you find a barcode system helpful for tracking the props?

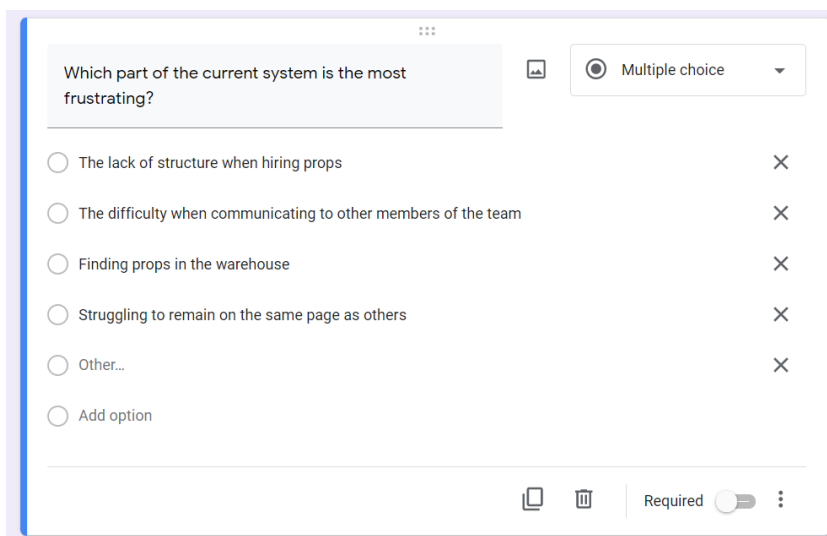
I don't have experience with using them but I suspect that it may generate more issues than solutions such as with the stickers becoming damaged or being removed when shooting. It also introduces another step which can go wrong in processing the props. I think that keeping the process as minimal as possible would be helpful.

5. How comfortable are you with using a new user interface?

I don't find it particularly easy to understand new technology so keeping the system as easy as possible would be appreciated. I am used to using other prop hiring systems such as farleys so making them similar could help. While some of my team are younger, I think a majority of my colleagues would like the system to be as straightforward as possible.

Survey Questions:

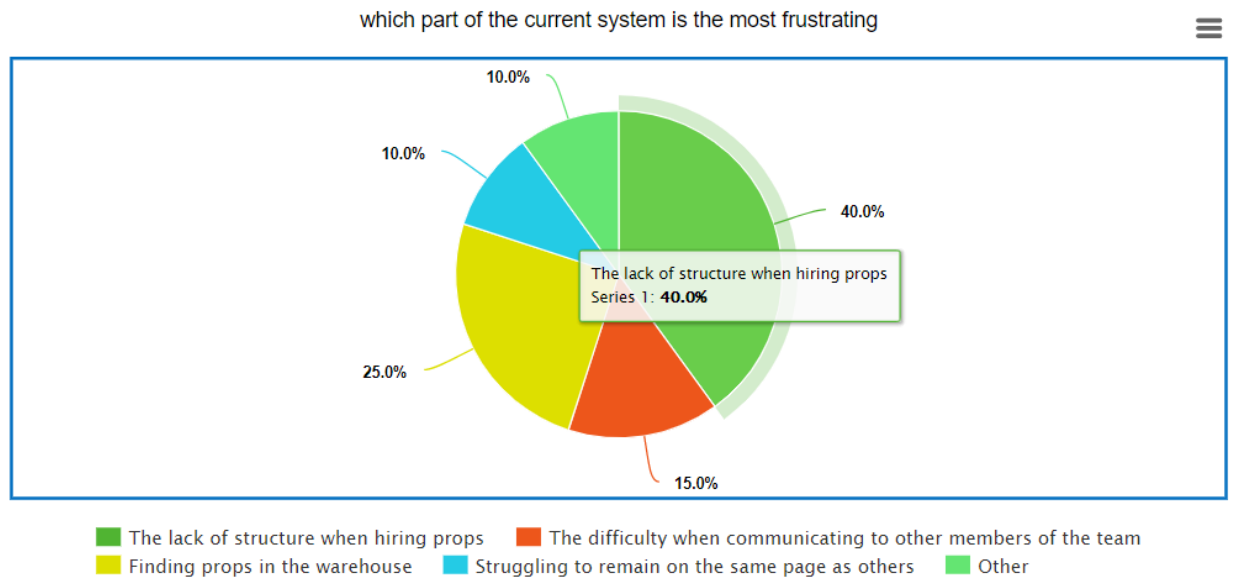
1. Which part of the current system is the most frustrating?



The screenshot shows a survey question interface. At the top, there is a question: "Which part of the current system is the most frustrating?". To the right of the question is a "Multiple choice" dropdown menu. Below the question, there are six radio button options, each followed by a delete icon (X):

- ☐ The lack of structure when hiring props
- ☐ The difficulty when communicating to other members of the team
- ☐ Finding props in the warehouse
- ☐ Struggling to remain on the same page as others
- ☐ Other...
- ☐ Add option

At the bottom of the interface, there are icons for a document, a trash can, and a "Required" toggle switch which is currently turned on.

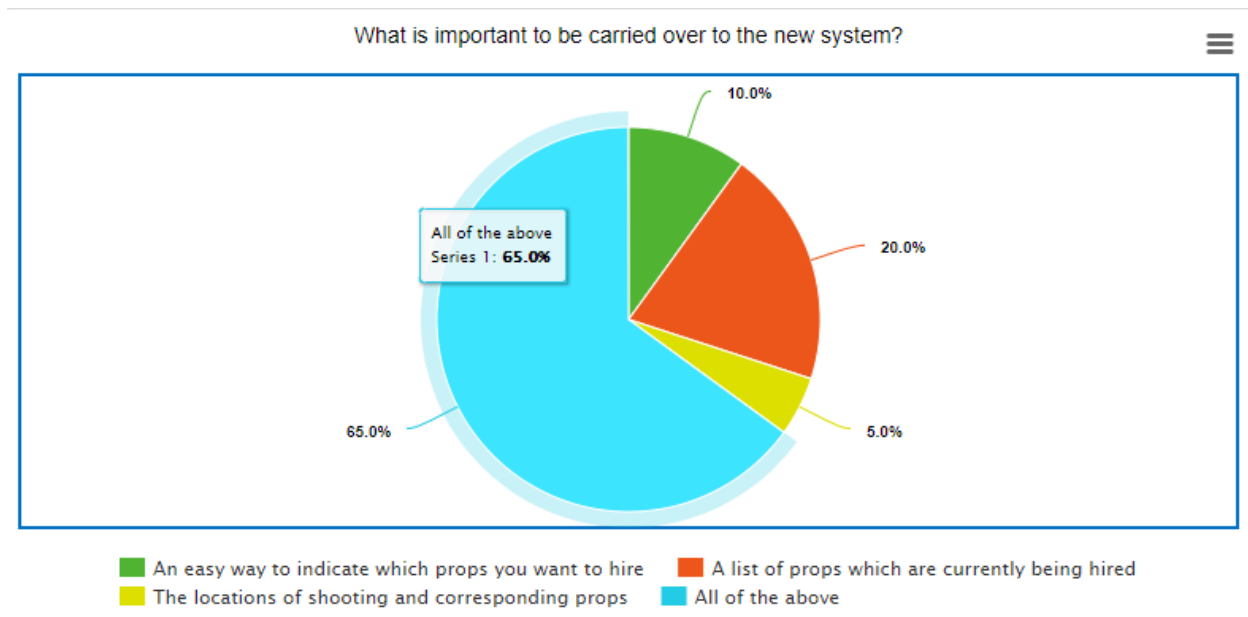


From this question I was able to learn that the problems which are faced by my client are also impacting those she works with which means this approach will help a majority of those working in the prop house. The main objective of this project will therefore to be streamlining the rental process for props and the returning of them to storage after use.

2. What is important to be carried over to the new system?

What is important to be carried over to the new system?

- ☐ An easy way to indicate which props you want to hire
- ☐ A list of props which are currently being hired
- ☐ The locations of shooting and corresponding props
- ☐ All of the above

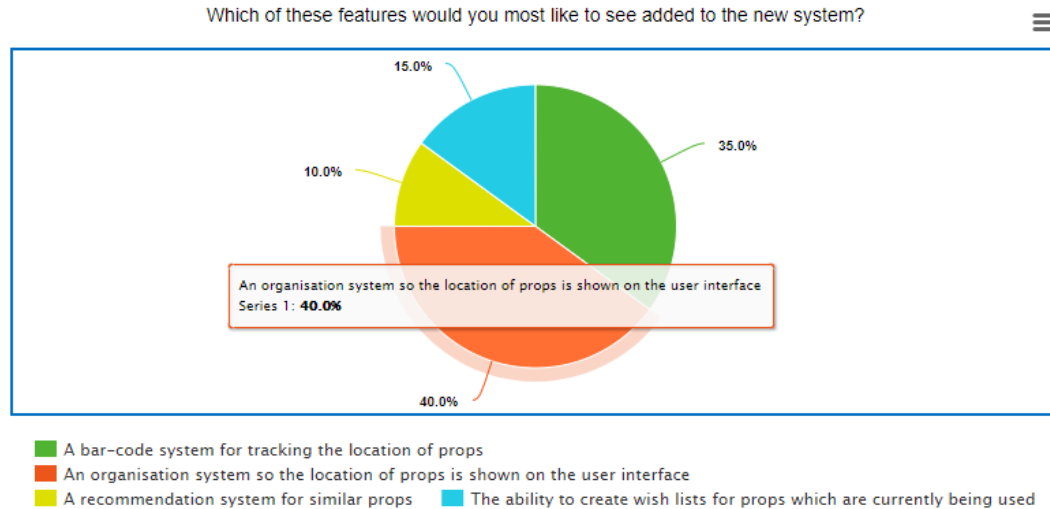


The results from this survey question show that there is an overwhelming majority in support of including all these features. I believe that this is because these are the main problems which exist in the prop house currently so a system which solves all of them is what the employees are looking for. While I will strive to complete all of these, I think that an understanding of which props are currently available within the database is the most important change which needs to be made.

3. Which of these features would you most like to see added to the new system?

Which of these features would you most like to see added to the new system?

- ☐ A bar-code system for tracking the location of props
- ☐ An organisation system so the location of props is shown on the user interface
- ☐ A recommendation system for similar props
- ☐ The ability to create wish lists for props which are currently being used

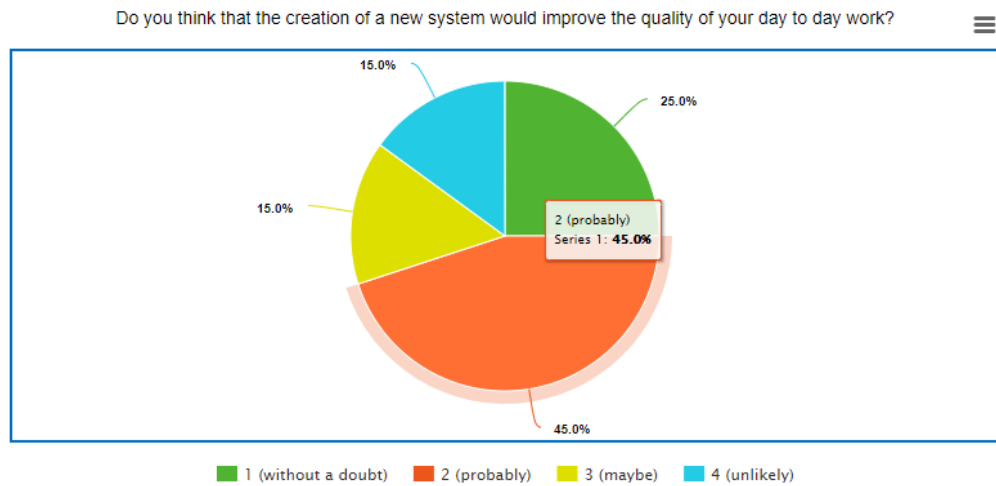


From this question there shows a clear divide between an organisation system and a barcode system. The fact that there is a majority vote for the organisation system suggests that employees have a difficult job remaining on the same page and browsing props. While there is also a large vote for barcodes, it seems a decisive issue with some people finding it could make the process too complicated.

4. Do you think that the creation of a new system would improve the quality of your day to day work?

Do you think that the creation of a new system would improve the quality of your day to day work?

- ☐ 1 (without a doubt)
- ☐ 2 (probably)
- ☐ 3 (maybe)
- ☐ 4 (unlikely)

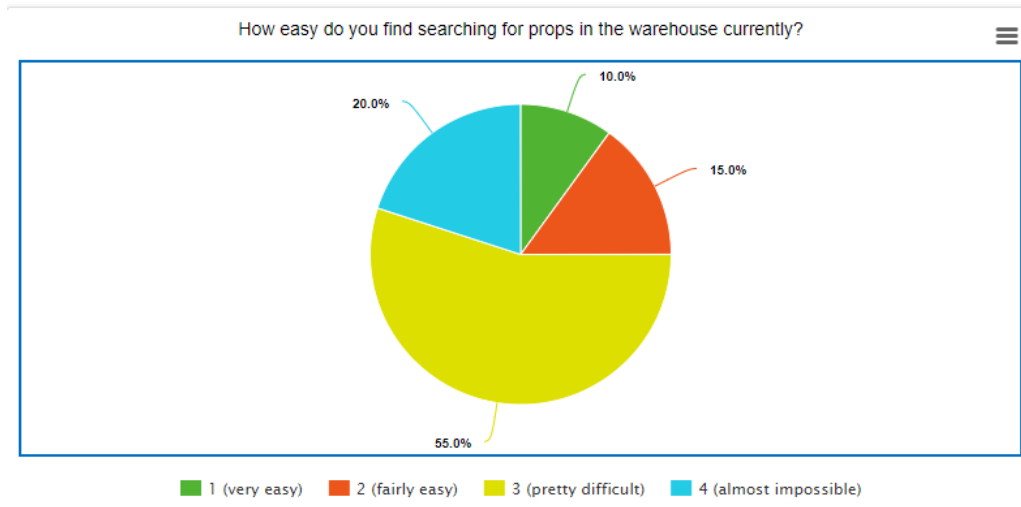


I believe that this response is expected with many agreeing that the new system will most likely help their work. However, there are some who seem more uncertain with 3 even selecting unlikely, I believe that this may be because a new technology can be daunting for some people. To overcome this hurdle, I think that a clear user interface and available instruction will help them become more confident with using the technology.

5. How easy do you find searching for props in the warehouse currently?

How easy do you find searching for props in the warehouse currently?

- ☐ 1 (very easy)
- ☐ 2 (fairly easy)
- ☐ 3 (pretty difficult)
- ☐ 4 (almost impossible)



This question was intended to determine how important an organisation system would be to the project. I think from the fact that a majority of people struggle to an extent shows how including a form of organisation will improve their quality of working life. However, this question may not accurately reflect the how they work, after speaking to my client she stated that her colleagues do not search for a specific prop and instead look around and use what is most suitable.

Current System:

Currently there is almost no system since it is common for props to be removed from the storage based on when they are needed. This is because the props are owned by Netflix which means that there are no hire forms which are required such as how long the prop will be used for. When they are placed back into the storage there tends to be a loose form of organisation where they are grouped based on which Netflix show is currently using them. They may also be grouped on the type of prop such as the chairs being placed together.

The problems with the current system are that it is hard to find the prop you are looking for, many props often go missing since it is hard to track them between sets and it is unclear which props are available for use since there is no way of checking.

There are 2 main groups of users for the system, the set decorators such as my client who decides which props should be used to decorate the sets. And the prop men who are responsible for managing the storage of the props, they are also responsible for placing the props back into storage after use. While there are other roles within the company which forms a hierarchy, most roles can be placed into one of these two groups.

Matrix of User Roles and Responsibilities:

User Role	Activities and Description
Set Decorator	Determines which props are required for each set
Prop Master	Manages the inventory of props through a team

Matrix of Actions Performed by Job Roles:

Activity	Prop Man	Set Decorator
Finding props in store	x	x
Putting props back	x	
Checking which are currently available		x
Organising props	x	

In general, for the new system the process would need to be made digital by placing it in a database system. In addition, a recommendation system could be required which would allow for the easier selection of related props. Finally, a check out system for each item would be required in order to ensure that the items are tracked.

Data Sources and Destinations:

Input	Current System Inputs
1	Send list of required props to prop master
2	Props leaving warehouse entered into a form
3	Determining whether the props have returned safely and entering this information into a form

Output	Current System Outputs
--------	------------------------

1	Prop master collects props for location shoot or responds that some of the props are not available
2	If props are not returned in correct condition, then further action is taken

Current reports are not organised in a formal manner. If a set decorator such as my client wants to send a list to the prop master, there would just be an email sent but not a form of any kind. This can make tracking simultaneous requests difficult. This is also the case for when taking the props from the warehouse, there may be some paper form, but this would be a note rather than an organised system. While there are email communications between workers in the prop house, there is no centralised digital system where decisions can be tracked.

For the new digital system, the hope is that this process would be similar except the requests and forms would be organised into a digital database. There may be more inputs and outputs such as when entering the prop code for the item leaving the warehouse.

Conversion to New System

Analysis Data Dictionary:

Prop Entity Table

Variables	Description	Validation
Prop Type	The type of prop	Must be from a list of options such as: chair, light, bed...
Prop Date	The numerical year for the item such as 19XX	
Era / period	The time period from which the prop would be found	
Colour	The most prominent colour of the prop	
Set type	Where the item would typically be used	
Programme	The show which currently owns it	
Size	The dimensions of the item	A pair of 2 integers to create

		dimensions in cm
Image	Photo of the item	Stored as a valid file path and non null
Prop ID	Identification code	Auto increment and unique
Quantity	The number of this prop which are available	Must be integer
Hired Status	Whether or not the prop is currently being used	Boolean value (true if already hired)

User Entity Table:

Variable	Description	Validation
User ID	Unique identifier of employees	Auto increment and unique
First Name	The first name of the user based on the details entered in the new user form	
Surname	The second name of the user	
Email	The main contact detail for the user	Must contain @
Phone	The secondary contact method, could be used for validation	Must be a certain length such as 11 for UK numbers
Job Role	Determines their seniority and role in the production	Must be one of the drop down options such as runner

Job	The production which the employee is currently working on	Must be entered in the format of being a single word
-----	---	--

Orders Entity Table:

Variable	Description	Validation
Order ID	The primary key used for orders	Unique and auto increment
Prop ID	Identifies the prop used in the order	
User ID	The ID of the user who filed the order	
Shoot ID	The identifier of the shoot location and information	
Order Detail ID	Connects the order to additional information about the order	

Shoot Location Entity Table:

Variable	Description	Validation
Shoot ID	Used to identify information about the shoot	
Location Postcode	The post code of the location	Should follow the format of UK postcodes

Duration	How long the production will be filming there	Should be calculated based on the number of days
Production	The name of the production	One of the options and one word

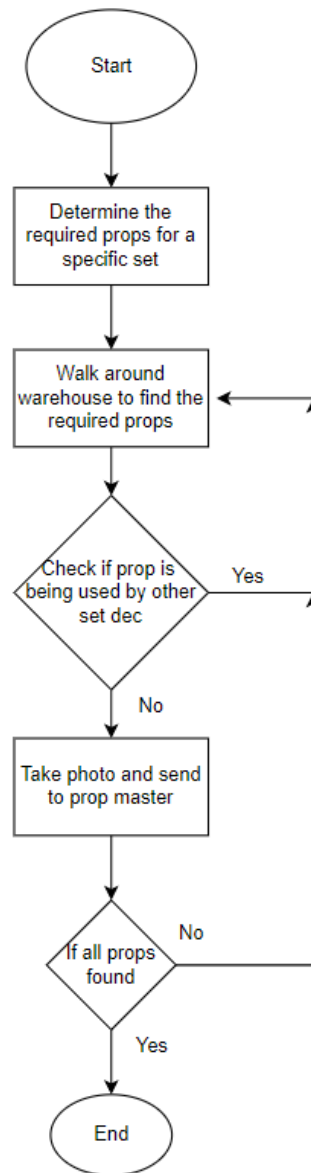
Order Details Entity Table:

Variable	Description	Validation
Order detail ID	Used to identify within this table	
Hire Date	The date when the order is initially submitted	Date in the format of YYYY/MM/DD
Return Date	A number of days after the hire date based on the length of filming	
Quantity	Number of props in the order	Integer
Order ID	The initial order id	
Shoot ID	The identifier for the shooting location and information	

Current System Flow Charts:

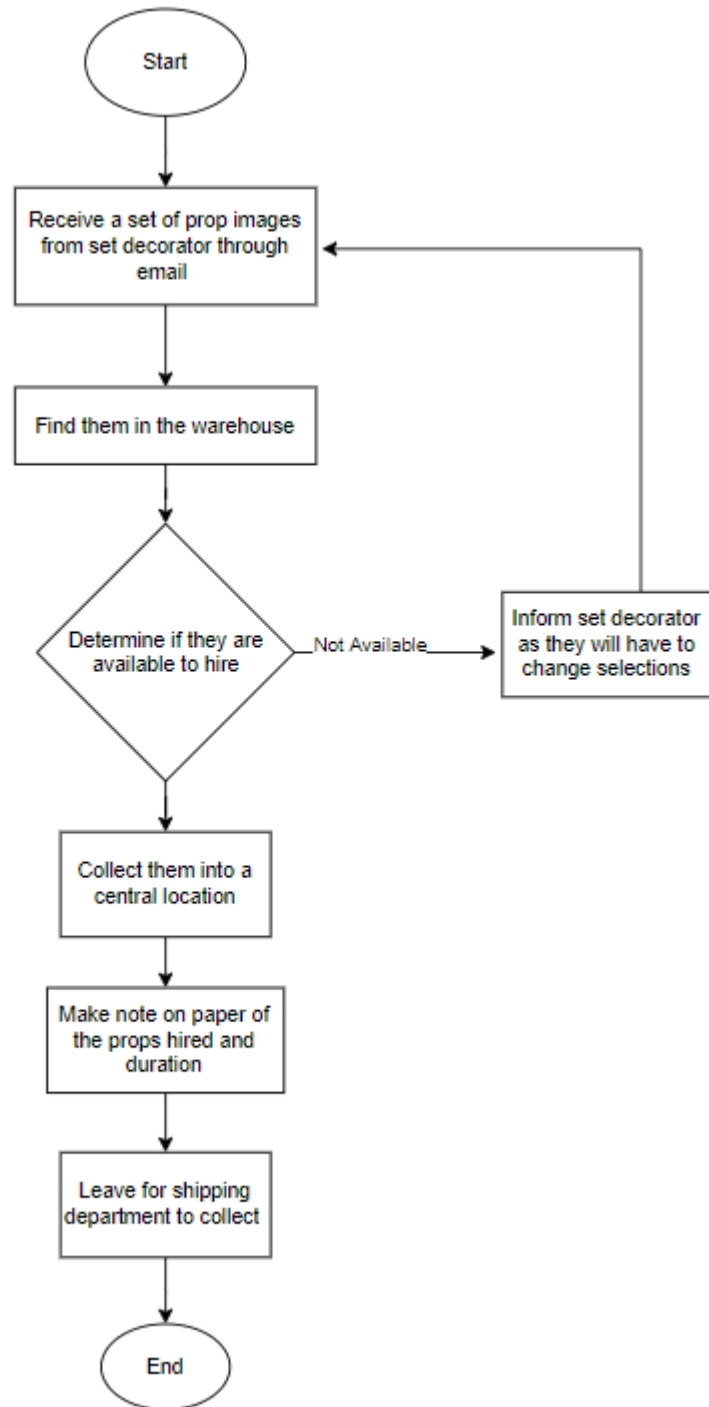
Hire Process flow Chart:

This flow chart shows the process for hiring a prop under the current system based on the perspective of the client as a set decorator. Due to the size of the warehouse the set decorator can find themselves wasting time as they search for an item. In addition, having to check the availability of the prop means that hiring the item is a back-and-forth process of checking with different departments which is inefficient.



Prop Transfer Flow Chart:

This flow chart shows the process for the prop master to remove items from the warehouse currently. This is inefficient due to the lack of formal manner to receive the list since it currently is sent through email which makes tracking hires more difficult. In addition, the note on paper made once they are collected, is not officially filed.



Algorithms:

Item Hire Pseudocode:

```
1 bool userLogin;  
2 bool hireStatus;  
3 bool continueHire  
4 string itemID;  
5 array[][] itemCharacteristics;  
6  
7 if userLogin == true and hireStatus == false then  
8     output("this item is available for hire");  
9     output("these are the characteristics: " + array[itemID][COLOUR] + array[itemID][PERIOD] ... );  
10    output("do you want to hire this item");  
11    continueHire = userInput;  
12    if continueHire == true then  
13        output("you have successfully hired this item, collect from " + itemCharacteristics[itemID][LOCATION]);  
14    else then  
15        output("you have decided not to hire, press back to continue browsing");  
16 else if userLogin == true and hireStatus == true then  
17     output("this item is already hired");  
18  
19 else if userLogin == false and hireStatus == false then  
20     output("incorrect login details please try again");  
21  
22 else then  
23     output("there has been an error please try again")  
24  
25
```

This is the pseudocode for hire of a prop which would be essential for the new system. The current system uses a similar process however without the use of an algorithm.

The code starts by declaring all the relevant variables, the array `itemCharacteristics` is a 2D array which is used to store all the relevant characteristics of the items which can be found in the Analysis Data Dictionary. While this currently takes the form of a 2D array for the ease of use in pseudocode, this will most likely become a database at a later point in development.

The algorithm checks if the user has the correct details and if the item is available, if this is true then it displays characteristics and checks if the user still wants the item. When this process is complete the user is provided with the location of the prop so they can find the item.

There is also the option to incorrectly input some value which would provide an error message and direct the user back to the original screen to try again.

Recommendation System Pseudocode:

```

1 array [][] itemCharacteristics; #a 2D array for the prop itemCharacteristics
2 string currentItemID           # stores the ID of the item you are currently looking itemRelatedness
3 array [][] itemRelatedness     #an array which stores how similar an item is compared to current item
4 array orderedRelatedness       #an array which orders the list in terms of similarity
5
6 for i = 0 , array[0].len() , i++ then
7     for x = 0 , array[1].len() , x++ then
8         if itemCharacteristics[currentItemID][i] == itemCharacteristics[x][i] then
9             itemRelatedness[x][i] += 1;
10
11 orderedRelatedness = itemRelatedness.sort();
12 output ("this is the recommended item " + orderedRelatedness[1])
13
14

```

This algorithm is designed to recommend other props based on the current item you are looking at. It does this by looking at which props share the most common attributes. The for loops in this pseudocode are used to loop through all values in the itemCharacteristics array and compare them to the current item. If the characteristic is the same then the item gets +1 in relatedness. Once all items have been compared the item with the greatest relatedness is recommended.

Since the top value in relatedness would be the item itself you recommend the second value in the list which is at position 1.

Another way this problem could be tackled could be using machine learning such as scikit which is a python library, this would allow the values to be represented as vectors and compared in terms of similarity. However I believe that integrating this approach into the project may be too ambitious and using a more basic algorithm is better suited.

Objectives:

Objective 1:

Due to the different workers and positions in the warehouse system, my client believes that a login system would allow the tracking of orders to be easier. In addition, it would improve security by preventing those without an account from accessing and impacting the prop storage.

Target: Create a secure login system while considering how to ensure the encryption of sensitive data

Success Criteria:

- A clear GUI which allows the user to log in with ease
- Prompts the user to attempt again if incorrect
- Directs the user to the appropriate page if login is correct
- Stores the data of the user in a database securely

Sub Tasks:

- 1) A function which stores the password and details of a user in a secure process such as with a form of encryption. The use of a custom hash function stores the password value in ciphertext which means that the data is not vulnerable in the event of a database security issue.
- 2) If a user attempts to login using incorrect details, then a page will be shown which indicates that the details should be re-entered. This will prevent confusion for the user and will lead to an easier user experience.
- 3) If the user is not yet registered in the system, then they should be able to create an account through the use of a button on the login page. This will redirect them to a different page which has fields where data can be entered. This would include the name, role, email and phone number so that the account is clearly associated with them.
- 4) In order to prevent the use of erroneous data, all fields where data is entered should have some form of verification to check that the data fits what is expected. An example of this could be when entering the email address and knowing that '@' must be used. This would prevent mistakes from being made by the user and would also improve the quality of information stored about the user in the database.
- 5) While there are several pages associated with the login system, there should always be either a back button or a home button available for the user which provides for easy navigation and a better user experience.

Technical Requirements:

- Single table SQL query (parameterised)
- Use of regular expressions to validate input of username and password
- Research and use of java encryption function

Objective 2:

An essential element of this project is searching and finding the available props in the database. My client finds the easiest method of searching for items in the database would be the application of a search system which is based on the key characteristics of the props. This would allow the available props to be easily found which is an issue with the current system.

Target: Prop Search and Browse
<p>Success Criteria:</p> <ul style="list-style-type: none">-Allows user to search for items and returns results-Can have criteria applied to the search such as specific colours-Returns error if no props are found-Shows images of corresponding props-Redirects to hire page
<p>Sub Tasks:</p> <ol style="list-style-type: none">1) When the user is searching for props using the text search box the program needs to be able to recognise the type of item a user is looking for. This may include the ability to accept several keywords which it can match items to rather than just taking one such as recognising the number of values entered.2) If the user's search does not yield any results from the database then a message should be shown which shows that their terms have not matched. This could prompt the user to try using a different set of terms or reducing the number of filters applied to the search. There would not need to be a button to redirect, it would just need to indicate to the user that while the search is functioning as intended, it has still not found any results.3) Upon selecting a prop, the program should provide the user the option to view more details on the item. This allows the user to view the characteristics of the prop such as the era, colour and type. The detail page should also give the user the option to add to a basket or hire the item immediately.4) On the search page there should be a clear way to display the information about the props found in the search. This could include information about the props and an image of the prop which would allow the user to select the appropriate item to hire. The user will also be able to easily interact with the results such as scroll system which makes it easy to view all the results.
<p>Technical Requirements:</p> <ul style="list-style-type: none">-Aggregate SQL functions-Merge sort or similarly efficient sort-Hashing-Writing and reading from files

Objective 3:

A feature which would improve the workflow of my client is the implementation of a recommendation system, this is because it would allow the user to easily find similar props. My client finds that when dressing sets that many of the items are connected by similar characteristics such as colour or type which is why I believe that recommending similar props would be effective.

Target: Prop Recommendation System
Success Criteria: <ul style="list-style-type: none">-Create a button which redirects to props that are similar-Have a page which presents the recommended props-Highlight which elements are similar to the original prop
Sub Tasks: <ol style="list-style-type: none">1) Having a button on the search page would be used to direct the user to a specific page where the recommendations could be browsed in one location. Placing this button near the search page would make switching between specific searching and browsing recommendations an easy process.2) Create a page which shows the recommendations based on props which were previously hired by the user. To manage store these details, a database of previous hires would have to be kept for each user with this data then being used to recommend.3) An algorithm would also have to be created which would generate the recommendations. This could then be used to determine which items would be shown on the recommendation page. The pseudocode which has already been made could be used as a starting point for the algorithm.
Technical Requirements: <ul style="list-style-type: none">-Complex user-defined algorithms (e.g. optimisation, minimisation, scheduling, pattern matching) or equivalent difficulty-Merge sort or similarly efficient sort-Recursive algorithms

Objective 4:

When looking for props my user finds that she looks for related props in one session which means that a basket system would improve the user experience. This would allow the set decorators to quickly and efficiently hire props.

Target: Create a basket system

Success Criteria:

- To be able to browse props and add items to the basket
- To not allow the same item to be placed in the basket to avoid duplicates
- Only allow items which are available to hire to be added to the basket so there is no confusion when basket is checked out
- To allow the basket to be viewed and allow the user to remove items
- The ability to hire all items in the basket simultaneously and generate the corresponding order forms for it

Sub Tasks:

- 1) Create a button on the item details page which allows the item to be added to the basket
- 2) Ensure that the correct checks are made before the item is added such as the existing items in the basket and the availability of the item
- 3) Store the item basket based on the user currently logged in and delete the basket if there user logs out so that the other users are not impacted
- 4) Provide one button which generates all the order forms and hires the correct props for the user
- 5) Allow the user to view all the items in the basket through the press of a button on the search page
- 6) The ability to edit and remove items from the basket through a simple UI

Objective 5:

Once the props have been selected the user should be able to find and hire props automatically. This would generate an order for this item in the database, allowing for the hiring of props to be tracked. This is an important feature for my user because currently the method of hiring props does not have a clear record or process which can be improved through this system. It should also be possible to return props through the use of the prop ID which would be found with the search function.

Target: Hire, return and add system for props
Success Criteria: <ul style="list-style-type: none">-Hire of props through a clear UI with automatically generated records about the orders-The ability to return items through a dedicated page based on the item id-A page which allows new props to be entered into the database with the characteristics and images
Sub Tasks: <ol style="list-style-type: none">1) Using the existing page which shows item characteristics, buttons will be added which add the item as either a hire or add to basket.2) A page which can be used to return items.3) A page which shows a form to enter information about the new prop. This would include the characteristics and an image; the image would be located through the use of a file explorer which would improve the user experience.
Technical Requirements: <ul style="list-style-type: none">-Aggregate SQL statements-Lists / hashing-OOP

Project Limitations & Constraints:

Data Volumes:

The current system does not have a formal way of storing data, there is a form which can be filled when hiring an item however this is rarely done. This will be altered for the new system where a database will be created to store all the values, this can be accessed through a login system and allows for manipulation using SQL.

For the storage of prop information such as characteristics there would be approximately 10 characteristics x 1000 props. When stored in excel this comes to roughly 41kb. However this does not consider the use of images which may be needed in the database. If we consider the average JPG to be 400kb when compressed then multiplied by 1000 props this would lead to around 400mb worth of data.

This amount of data seems manageable however could lead to longer access times if the images have to be transmitted from the server.

An additional set of values which must be stored would be the users details such as logins and access rights, this would also take up space however would be an insignificant amount in comparison to the size of the images.

System Considerations:

The system would not be taxing in terms of computational power, therefore it should be capable of running on most computers. Since the database would not be stored locally on the computer it means that any client who is logged in should be able to access it. I think that having the application accessible on many computers is essential to allow different departments to communicate and work together efficiently. However a consideration is the use of barcodes, if these are to be scanned then the use of a mobile device or scanner may be required which could cause difficulties if the application is designed for a desktop interface.

There would be security requirements, the clearest one would be the data protection act since the database would be storing sensitive subject data such as name, password, email and phone number. Another consideration in terms of safety is maintaining the security of the prop house, if there was an attack on the network then someone may be able to obtain the right to remove items which would cause financial loss for the company. To circumvent these possibilities, the use of a secure encryption system would be essential, for example the hashing of sensitive data which is stored in the database. This could be applied to the password when submitted and when verifying the login details. Another attempt to maintain security could be to educate users on how to keep their devices safe since the human element is often the weakest point.

While the application may take time for users to become comfortable with, there should be almost no training required since the user interface should hide the complexities of managing the database. If developed to an appropriate standard, the application should be easy to understand for most users.

There could be some technical constraints if the size of server required grows too large. This could happen if there are more props than estimated or the number of variables required increases.

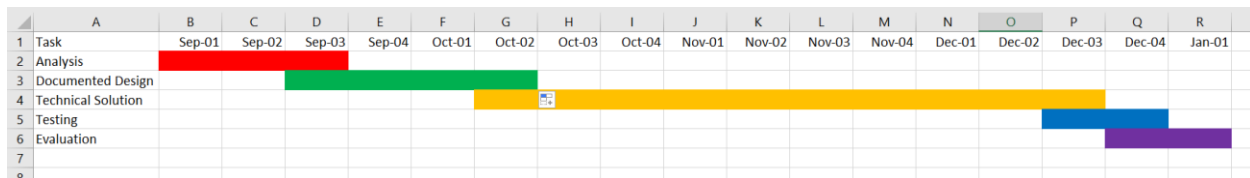
Overall, I think this suggests that a majority of machines would be able to run the software for the programme however a final consideration is that the software would have to be kept up to date. Since the software would need to communicate with other computers, I think that keeping the programme up to date after changes is important. This prevents issues where one user has a different set of data to another user which leads to them attempting to hire the same prop.

Limitations:

An additional functionality could be the addition of a path finding system which would tell the user the most efficient route to the prop in the warehouse. This would allow the easy navigation of the warehouse for both set decorators and prop men, it would also lead to a more organised prop house. The implementation of this would provide me with the opportunity to improve my understanding of graph traversal algorithms. However, I think this will not be possible to implement for this project since it would require the organisation of the prop house into clear marked areas and may be too much to include for this project.

Critical Path:

Gantt Chart:



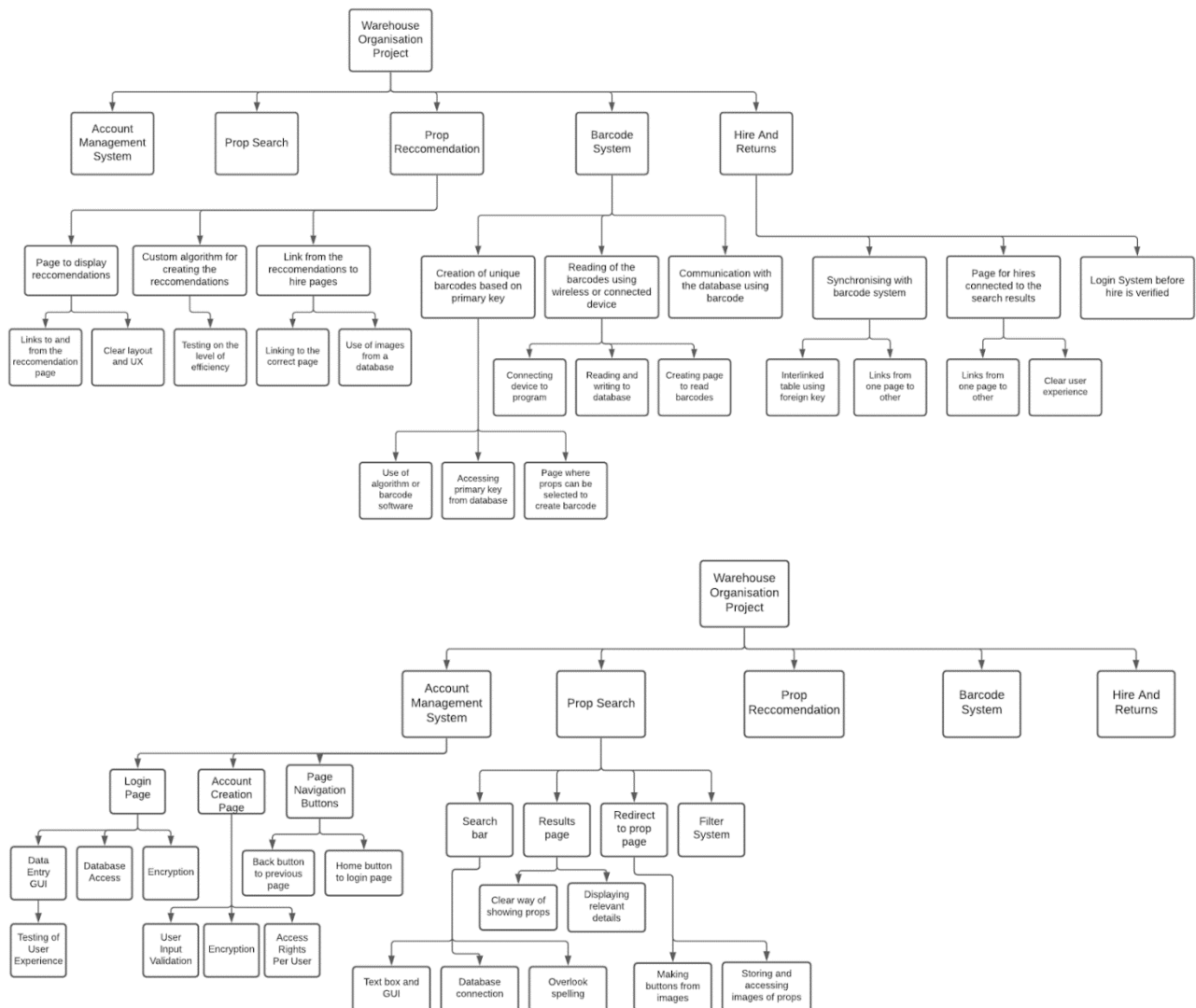
This Gantt chart shows the progress I would like to make with the project leading up to the deadline. It provides some overlap between elements which provides time for starting early or missing deadlines. The most important dates in this chart are the start of the technical solution which I believe will be the most challenging part of the project. I hope to be fully starting by the October half term which lands in the 3rd week of the month. This will provide me with time until December to get the technical solution completed. By completing a majority of the project moving into the Christmas holidays, it gives me time to test and reflect while also checking through the project for any errors.

Design:

Project Overview and Problem Deconstruction:

Structure / Hierarchy chart:

This diagram shows the hierarchy of problems which need to be solved to tackle the project. This starts with the main areas of the project and decomposes them into small problems which makes them easier to tackle. Once on the bottom-most layer of the nodes, the problems should be atomic and can no longer be broken down any further. This single diagram has been broken into 2 pieces, so it is easier to view and understand. While I believe this is a clear overview of the project at the moment, there may be additional elements which I have not considered. By creating this plan, I think it will prevent me from creating redundant code and allow me to connect separate areas in a clearer manner.



Hiring a Prop Flow Chart:

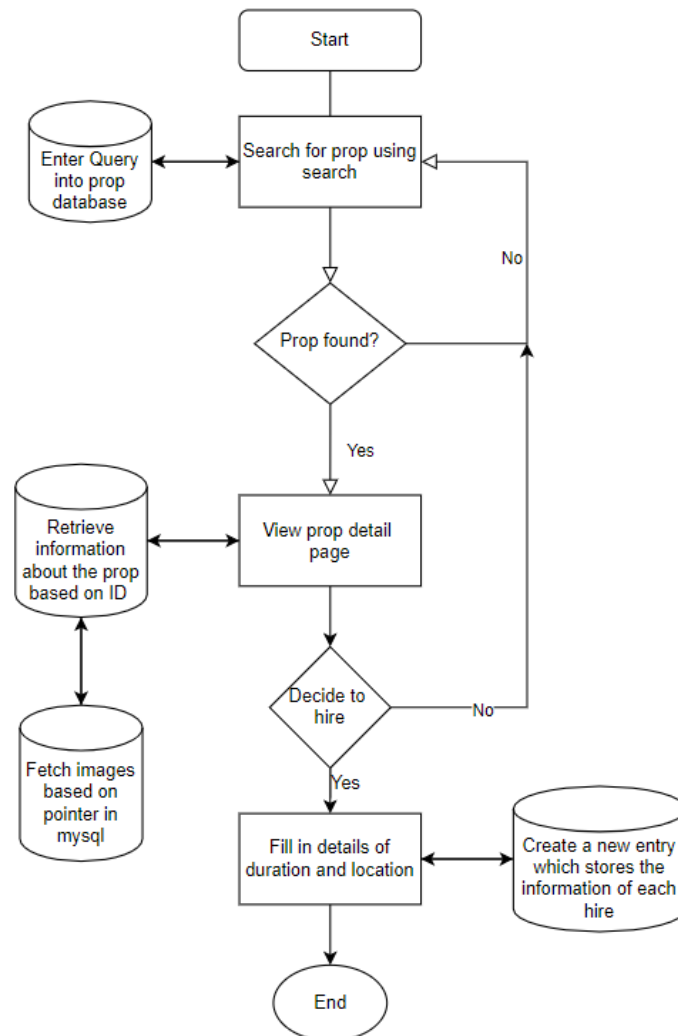


Figure 1

This flowchart shows the process for hiring a prop which the user would experience. It also considers the points where the database would need to be accessed such as when searching or storing information. This is denoted using the cylinder. The problem of storing and retrieving images using MySQL could be solved by storing the images in an external database and storing pointers to these locations in the MySQL database. An example of how these pointers could be stored is using a file path stored in the database which then references the image files stored locally on the user's PC. This is because it is not recommended that you store images directly in the MySQL database.

Login Flow Chart:

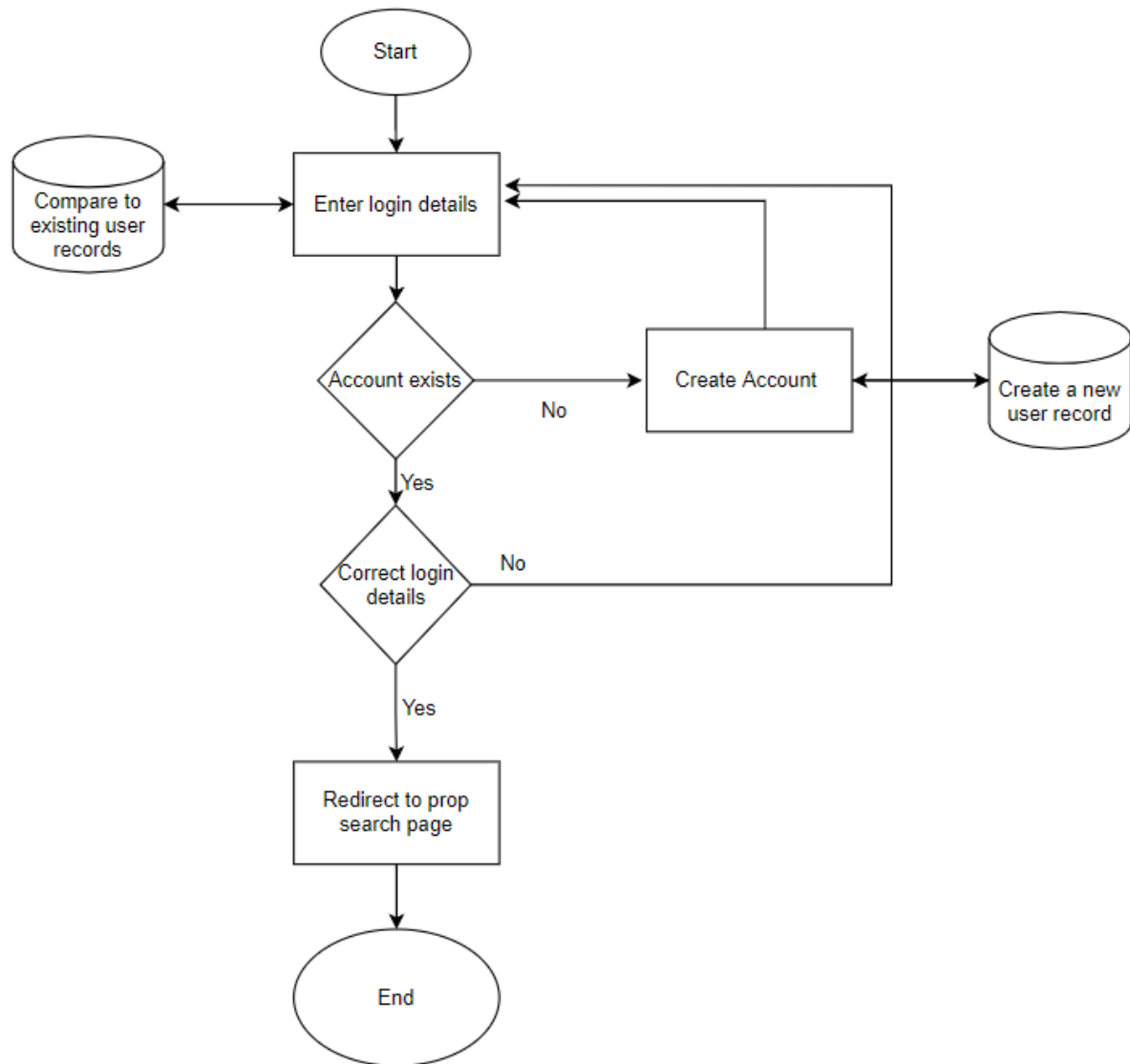


Figure 2

Figure 2 shows the process for either logging in or creating a new user profile. The user can either login using existing credentials or create an account which they will use from that point forward. When the user is creating an account, it is currently unclear as to which characteristics should be stored because there may be details which do not seem important currently but could be required if the project wants to scale. Finally, there should be some form of verification of the details entered so that there are not any issues, an example could be that the phone number entered must be of a valid format for the UK.

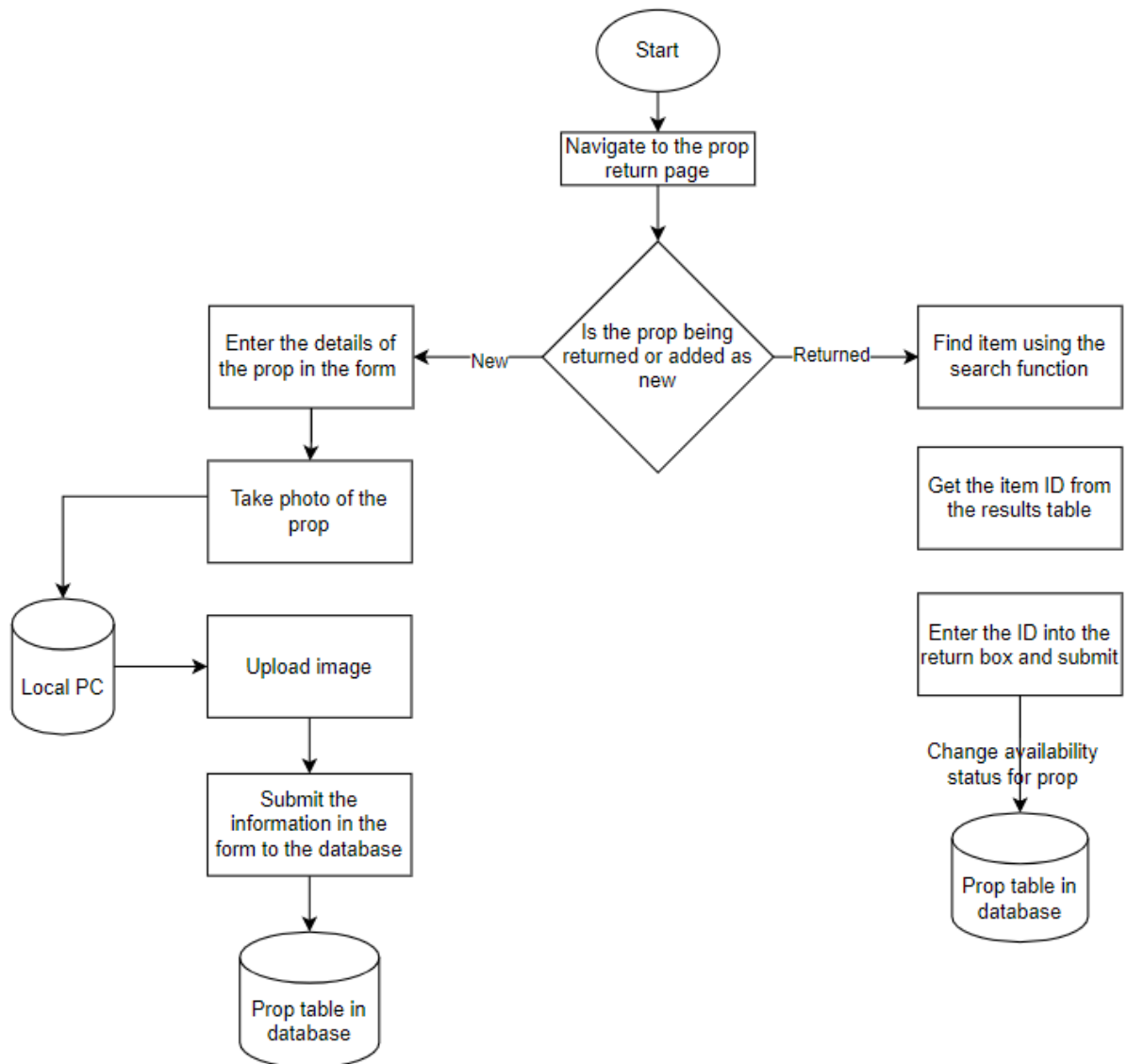
Login Pseudocode:

```
1 String username
2 String password
3 Boolean loginPass
4 Boolean createAccount
5 String newUsername = "null"
6 String newPassword = "null"
7
8 output("enter login details")
9 username = userInput
10 password = userInput
11
12 IF password == database.getPassword where username == database.getUsername
13     output("the login has been sucessful")
14     loginPass = True
15
16 Else
17     output("the login details are incorrect")
18     loginPass = False
19     output("do you want to create an account")
20     createAccount = userInput
21
22 IF createAccount == True
23     output("enter your new details")
24     newUsername = userInput
25     newPassword = userInput
26 Endif
27
28 Endif
29
```

This pseudocode shows the process of logging in which is shown in figure 2. This is the logical process which would take place so this would have to be converted to allow the user to interact using a user interface.

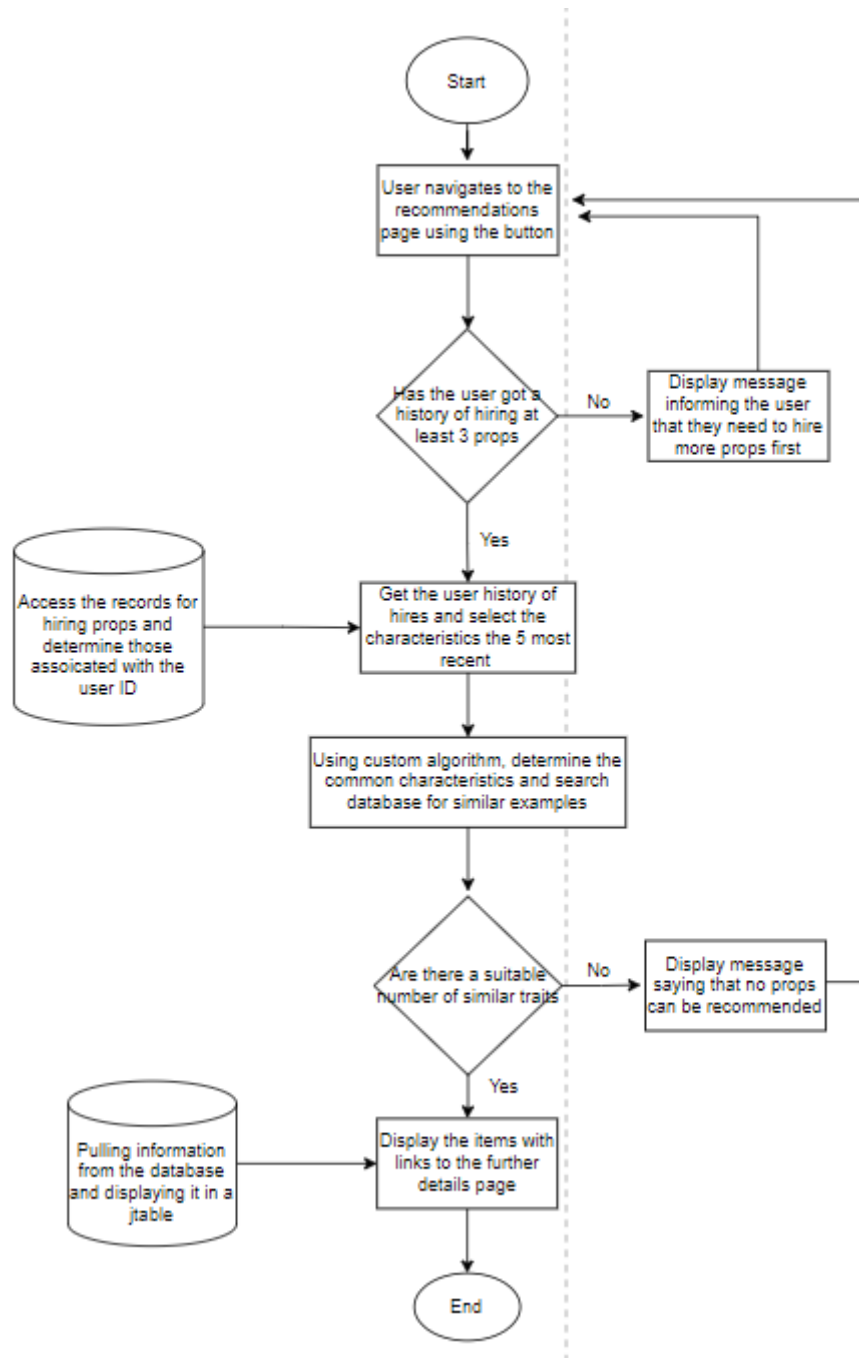
Prop Return Flow Chart:

This flow chart shows the process prop men would go through when using the system to return an item into the warehouse after it has been hired. It also provides the option of adding a new item to the database through the use of the same page.



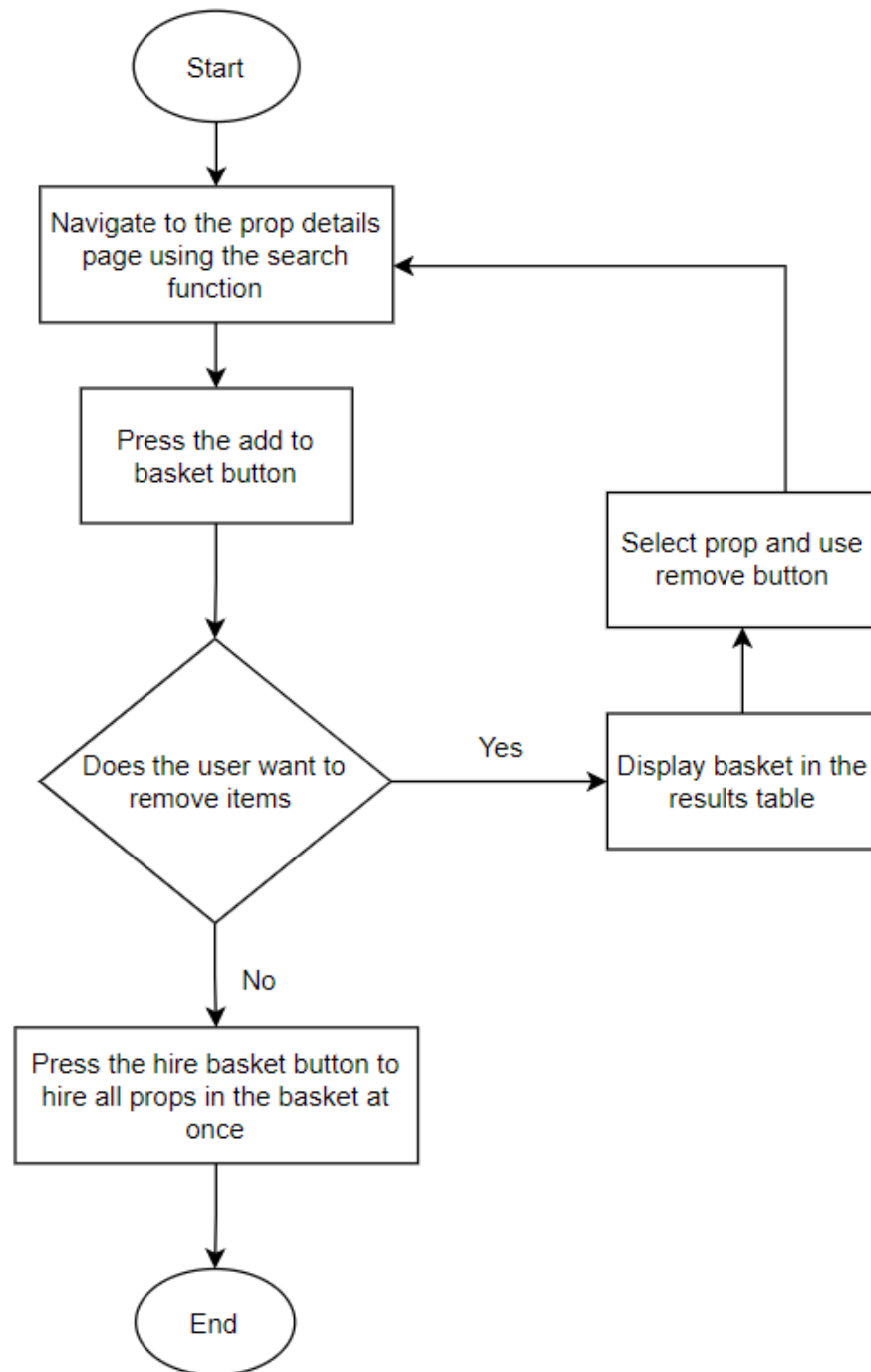
Recommendation Flow Chart:

The recommendations system can be accessed through a button found on the search page. Once the button is pressed the algorithm will calculate the recommendations based on what has been hired previously. These are then displayed through the use of the same results table used when searching for items.

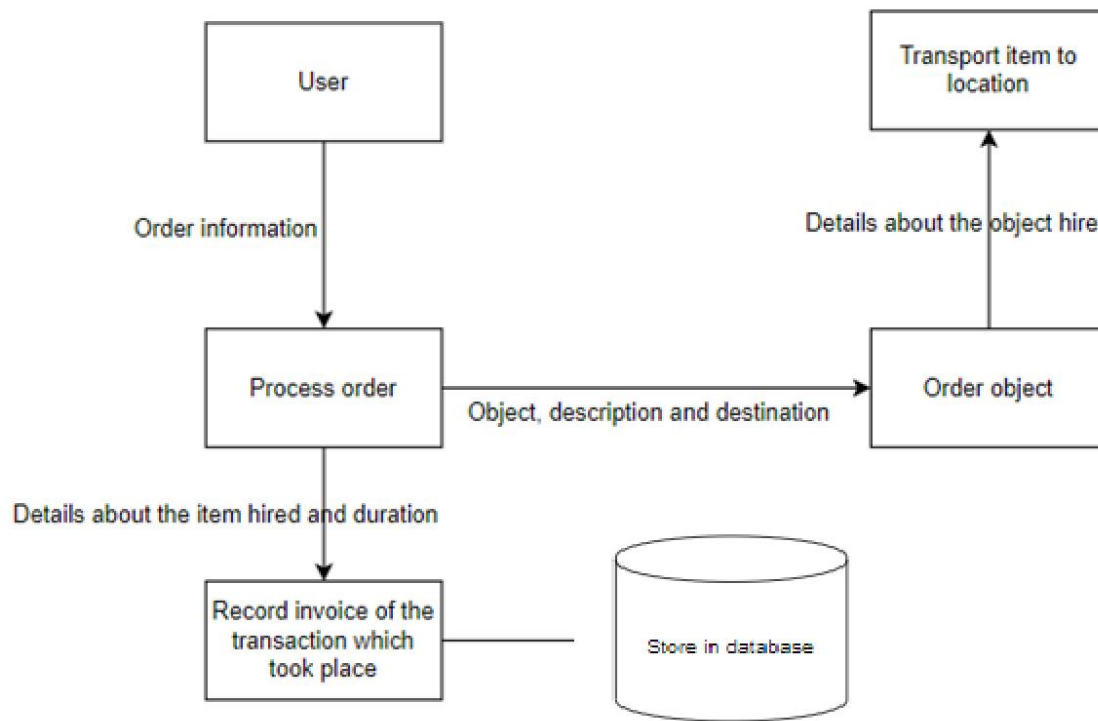


Basket Flow Chart:

This flow chart shows how the user would interact with the basket function to go from viewing an item to adding it to the basket and hiring it. This also displays how the user would have an option in how to remove items and view the basket.



Information Transfer Diagram:



This diagram is similar to an entity relationship diagram except it shows how elements in the project would be communicating with each other by transferring information. This can be seen with the initial user who would be communicating with the order process by entering the props they wish to hire.

Database Normalisation:

ONF:

<u>orderID</u>	<u>user</u>	<u>job</u>	<u>location</u>	<u>duration</u>	<u>Email</u>	<u>Phone</u>	<u>Role</u>	<u>propID</u>
AJ45	Alison Havey	Crown	AL9 5HX	2 weeks	ali.harvey@me.com	7493515110	SetDec	<u>233</u>
								<u>3243</u>
								<u>435</u>
								<u>33</u>
KW23	Alice Davison	Crown	WD3 6ER	1 week	aliDavison@outlook.com	7458235546	SetDec	<u>644</u>
								<u>434</u>
								<u>72</u>
								<u>104</u>

This shows the initial database for an order and includes all the values which are associated when hiring a prop from the database. However, this is an example of ONF since the data has not been normalised, this is because a single row can contain more than one value which is seen with the propID column. The primary keys have been identified by the underlines.

1NF:

<u>orderID</u>	firstName	surname	job	location	duration	<u>propID</u>	Email	Phone	Role
AJ45	Alison	Havey	Crown	AL9 5HX	2 weeks	233	ali.harvey@me.com	7493515110	SetDec
AJ45	Alison	Havey	Crown	AL9 5HX	2 weeks	3243	ali.harvey@me.com	7493515110	SetDec
AJ45	Alison	Havey	Crown	AL9 5HX	2 weeks	435	ali.harvey@me.com	7493515110	SetDec
AJ45	Alison	Havey	Crown	AL9 5HX	2 weeks	33	ali.harvey@me.com	7493515110	SetDec
KW23	Alice	Davison	Crown	WD3 6ER	1 week	644	aliDavison@outlook.com	7458235546	SetDec
KW23	Alice	Davison	Crown	WD3 6ER	1 week	434	aliDavison@outlook.com	7458235546	SetDec
KW23	Alice	Davison	Crown	WD3 6ER	1 week	72	aliDavison@outlook.com	7458235546	SetDec
KW23	Alice	Davison	Crown	WD3 6ER	1 week	104	aliDavison@outlook.com	7458235546	SetDec

This table has been normalised into 1NF because the data is now atomic which can be seen since there is only one value per row. This has been achieved by storing the data in many rows. However, this has caused a large amount of data redundancy since there are a lot of repeats in terms of all other columns such as email and phone of the users.

2NF:

<u>orderID</u>	<u>userID</u>	job	<u>propID</u>	<u>shootID</u>
AJ45	1	Crown	233	4
AJ45	1	Crown	3243	4
AJ45	1	Crown	435	4
AJ45	1	Crown	33	4
KW23	3	Crown	644	1
KW23	3	Crown	434	1
KW23	3	Crown	72	1
KW23	3	Crown	104	1

<u>userID</u>	FirstName	Surname	Email	Phone	Role
1	Alison	Harvey	ali.harvey@me.com	7493515110	SetDec
2	James	Miner	jminer@gmail.com	7779275978	PropMan
3	Alice	Davison	aliDavison@outlook.com	7458235546	SetDec
4	Ryan	Clifford	ryanCliff@gmail.com	7841738959	PropMan

<u>shootID</u>	location	duration
1	AL9 5HX	10 Days
2	NE 2GW	14 days
3	W5 1NX	7 days
4	WD3 6ER	14 days

Now the database is in 2NF since the partial key dependencies have been eliminated which means that both elements of the primary key has to be used to uniquely identify a row. This process has also eliminated most of the data redundancy which was found in the previous stage of normalisation.

3NF:

<u>orderID</u>	<u>userID</u>	<u>propID</u>	<u>shootID</u>
AJ45	1	233	4
AJ45	1	3243	4
AJ45	1	435	4
AJ45	1	33	4
KW23	3	644	1
KW23	3	434	1
KW23	3	72	1
KW23	3	104	1

<u>shootID</u>	<u>location</u>	<u>duration</u>
1	AL9 5HX	10 Days
2	NE 2GW	14 days
3	W5 1NX	7 days
4	WD3 6ER	14 days

<u>userID</u>	<u>FirstName</u>	<u>Surname</u>	<u>Email</u>	<u>Phone</u>	<u>Role</u>	<u>job</u>
1	Alison	Harvey	ali.harvey@me.com	7493515110	SetDec	Crown
2	James	Miner	jminer@gmail.com	7779275978	PropMan	Sherlock
3	Alice	Davison	aliDavison@outlook.com	7458235546	SetDec	Crown
4	Ryan	Clifford	ryanCliff@gmail.com	7841738959	PropMan	Hellboy

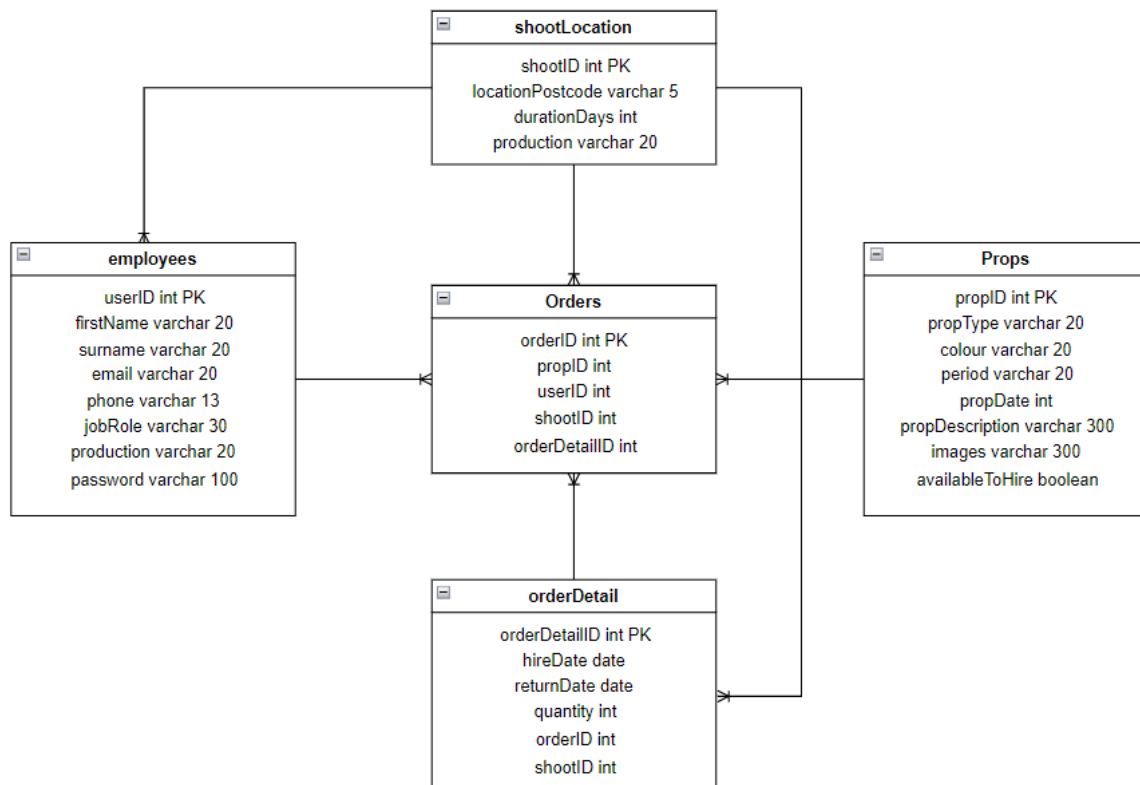
<u>propID</u>	<u>type</u>	<u>colour</u>	<u>period</u>	<u>date</u>
1	Chair	Red	Modern	70s
2	Table	Yellow	Victorian	1880s
3	Lamp	Green	Edwardian	10s
4	Chair	Brown	Modern	60s
5	Bed	Red	Art Deco	30s

<u>orderDate</u>	<u>orderID</u>	<u>historyID</u>
10.09.20	AR23	1
27.03.21	LE82	2
03.07.20	EW45	3

There was previously a non-key dependency since the job could be determined from the userID but that has been changed so that the job is now associated with the userID in a separate table. While in this example the userID can be determined from the shootID, that can change which means this database is now fully normalised.

Entity Relationship Diagram:

By using the normalisation stage of the database, we are able to create clear ER diagram which shows the relationship between each other elements and whether it is many to one or not. It is clear that each type of data is stored in a separate table which is brought together in the order table. This leads to each order only having a few characteristics which can be referenced in other tables.



Creating database tables:

Through the use of these DDL scripts I will create the SQL tables which were created and normalised in the previous normalisation stage.

Prop Table:

This table is used to store the characteristics of the props and information about them. The use of the auto increment key word allows the primary id to automatically increase when each row is added. The available to hire column is a binary value which allows it to be either true or false with the value defaulting to true since when props are first added they are assumed to be available.

```
CREATE TABLE `props` (  
  `propID` int NOT NULL AUTO_INCREMENT,  
  `propType` varchar(10) DEFAULT NULL,  
  `colour` varchar(10) DEFAULT NULL,  
  `period` varchar(10) DEFAULT NULL,  
  `propDate` int DEFAULT NULL,  
  `propDescription` varchar(300) DEFAULT NULL,  
  `images` varchar(300) DEFAULT NULL,  
  `availableToHire` bit(1) DEFAULT b'1',  
  PRIMARY KEY (`propID`)  
)
```

Employee Table:

The telephone number is only going to include digits however is being treated as a string so that the leading 0's which may be found at the start of the number are not removed as the number is automatically truncated.

```
CREATE TABLE `employees` (  
  `userID` int NOT NULL AUTO_INCREMENT,  
  `firstName` varchar(20) DEFAULT NULL,  
  `surname` varchar(20) DEFAULT NULL,  
  `email` varchar(30) DEFAULT NULL,  
  `phone` varchar(20) DEFAULT NULL,  
  `jobRole` varchar(10) DEFAULT NULL,  
  `job` varchar(20) DEFAULT NULL,  
  `password` varchar(100) DEFAULT NULL,  
  PRIMARY KEY (`userID`)  
)
```

Orders Table:

This orders table is able to combine the different primary keys which are found in the separate tables. To uniquely identify each row a composite key is used since it will always be unique. While there is an auto increment order id column, since a single order can contain many props there may be several rows with the same order id which is why the composite key is necessary.

```
CREATE TABLE `orders` (  
  `orderID` int NOT NULL AUTO_INCREMENT,  
  `propID` int NOT NULL,  
  `userID` int NOT NULL,  
  `shootID` varchar(100) NOT NULL,  
  `orderDetailID` int DEFAULT NULL,  
  PRIMARY KEY (`orderID`, `propID`))
```

Shoot Location Table:

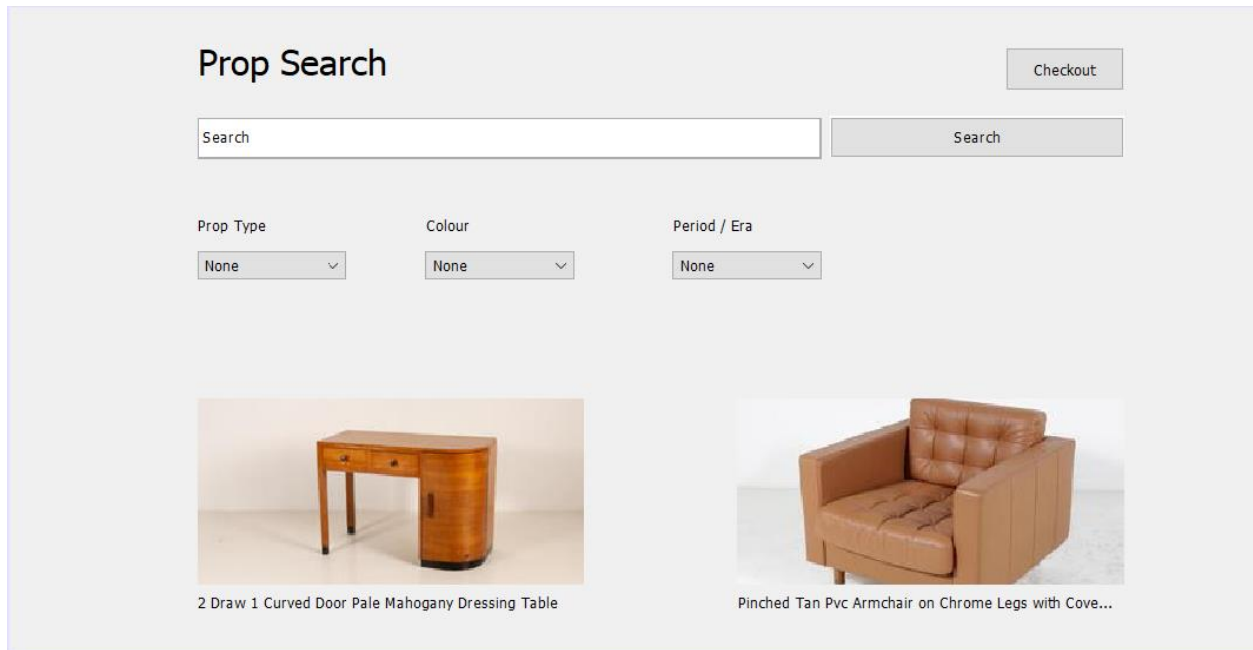
```
CREATE TABLE `shootlocation` (  
  `shootID` int NOT NULL AUTO_INCREMENT,  
  `locationPostcode` varchar(10) DEFAULT NULL,  
  `durationDays` int DEFAULT NULL,  
  `production` varchar(45) DEFAULT NULL,  
  PRIMARY KEY (`shootID`)  
)
```

Order Details Table:

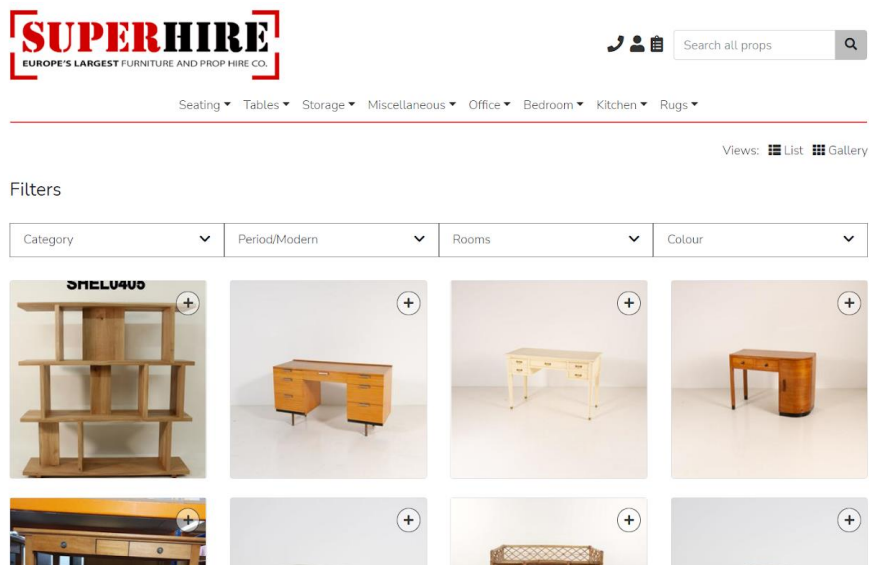
```
CREATE TABLE `orderdetail` (  
  `orderDetailID` int NOT NULL AUTO_INCREMENT,  
  `hireDate` date DEFAULT NULL,  
  `returnDate` date DEFAULT NULL,  
  `quantity` int DEFAULT NULL,  
  `orderID` int DEFAULT NULL,  
  `shootID` int DEFAULT NULL,  
  PRIMARY KEY (`orderDetailID`)  
)
```

UI Design:

Prop Search Page – Phase 1

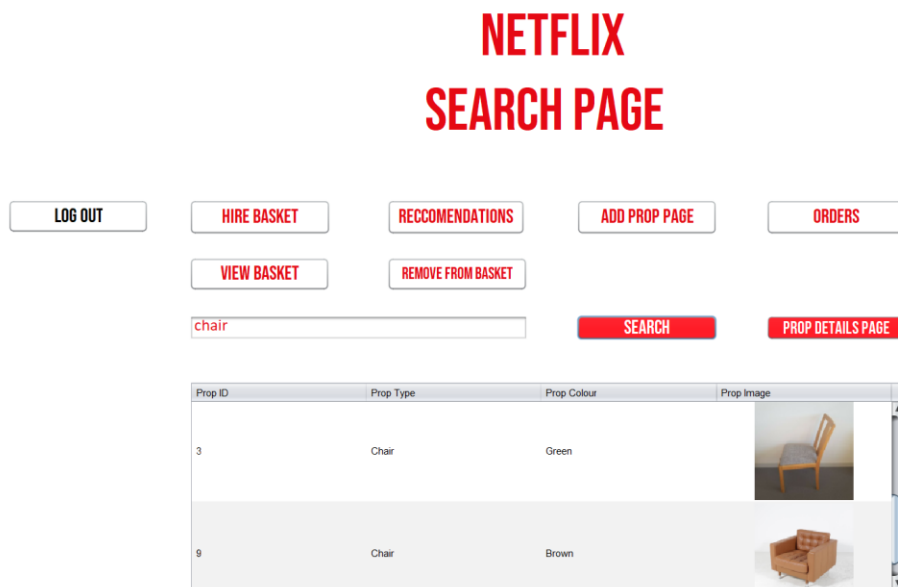


The initial design for the search page which I showed to my client was this one, it was created in NetBeans and shows how the search page would function. It would include a search bar and a set of variables which can be adjusted. Once the search criteria has been applied it would show a set of results from the database. These results would be links which could be clicked and direct to more information about the prop and allow it to be hired.



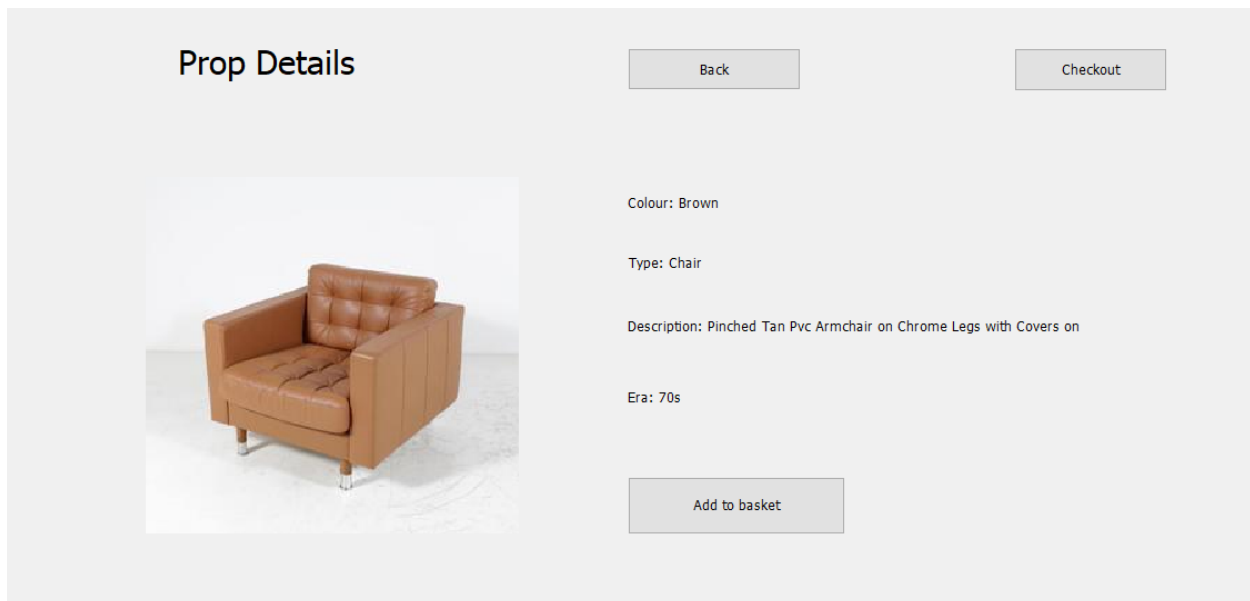
After showing my client the initial design, she showed me a reference in the form of SuperHire which is a site that fulfils a similar function. This design differs from mine in several ways such as the colours used and the fact there is a logo. The props can also be added to basket from just the thumbnail which would improve the user experience however I believe that implementing this may be beyond the scope of this project. By using this example of a page, I was able to improve upon the design.

Prop Search Page – Phase 2

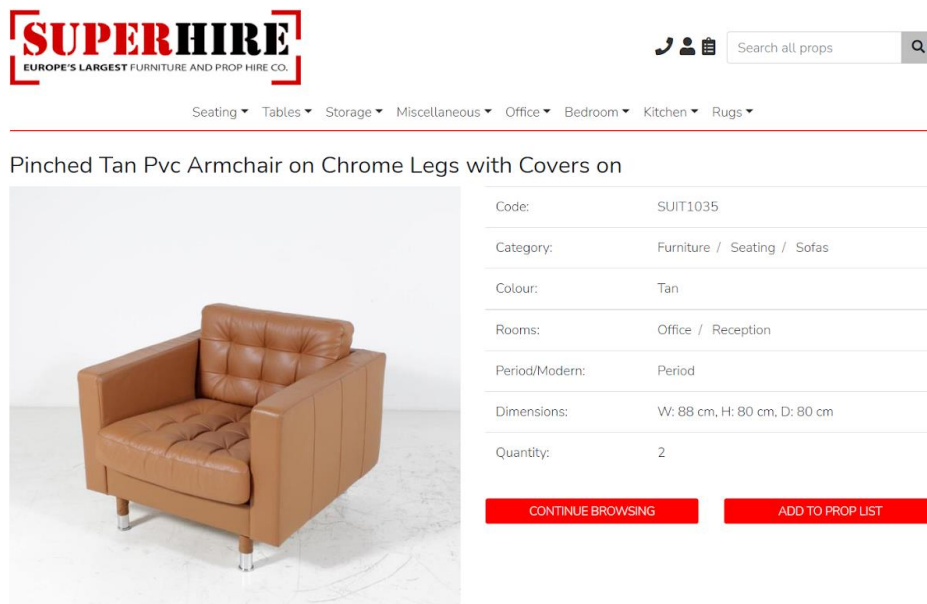


After reviewing the preferences of my client, I was able to make the design more in line with their vision for the search page. To use the super hire page, I used the logo of the company so that it is closer tied to those using the product. In addition, the new version of the UI design includes many of the functions which are found when using Super Hire such as the basket buttons. Finally, I decided to include more information about the props so that the user can clearly read essential information about the props without having to select. I think this table layout is good for this project since it clearly displays the essential information.

Prop Details Page Design:



The same process was applied for the prop detail page. This page would be displayed based on the prop clicked on. It would be used to display further details about the prop and provide the button to add to basket. From this point the user can either add to basket, return to browsing or checkout. They can also see a larger image of the item and a description.



From comparing the expectations of the client, the 2 pages are fairly similar in function. While the appearance of the pages is slightly different, the buttons included are essentially the same. This version has the search bar constantly present would be included, however may make searching more confusing. I believe that the placement of the return to browsing button is better for the SuperHire page since it is more intuitive. The super hire logo as functions as a button to return the user back to the search page which could be included in my design.

PROP DETAILS



PROP INDEX: 9

TYPE: CHAIR

COLOUR: BROWN

PERIOD: PERIOD

DATE: 1970

DESCRIPTION: NULL

HIRE PROP

ADD TO BASKET

RETURN TO SEARCH

This is the slightly changed version of the prop detail page, I have changed the back button to instead read “return to search” which makes it clearer where the button will lead to. By having all the buttons in a similar area it makes them easier to find and understand. While there could be additional functions such as a home page or a search bar, my client said that “it could feel more cluttered and confusing” which suggests that keeping the page as simple as possible would be beneficial. I have also added colour since it improves makes the page more thematically in line with the other pages.

Login Page Design:

NETFLIX LOGIN PAGE

USERNAME / EMAIL:

PASSWORD:

LOGIN

CREATE ACCOUNT

This is the design used for the login page, it would be the first page the user would see when first opening the program. The login button would redirect to the search page if the login details are entered correctly. The create account button would lead to an account creation page. After researching other options for a login page, my client and I agreed that this would work as an opening page since it presents the essential information in a clear manner. She did enquire about the forgotten password function however I believe that is outside the scope of this project since it would require sending emails to the user and confirming details with a web page and link. Therefore, this is all the information which is needed for the login page.

UI and UX Interview With Client:

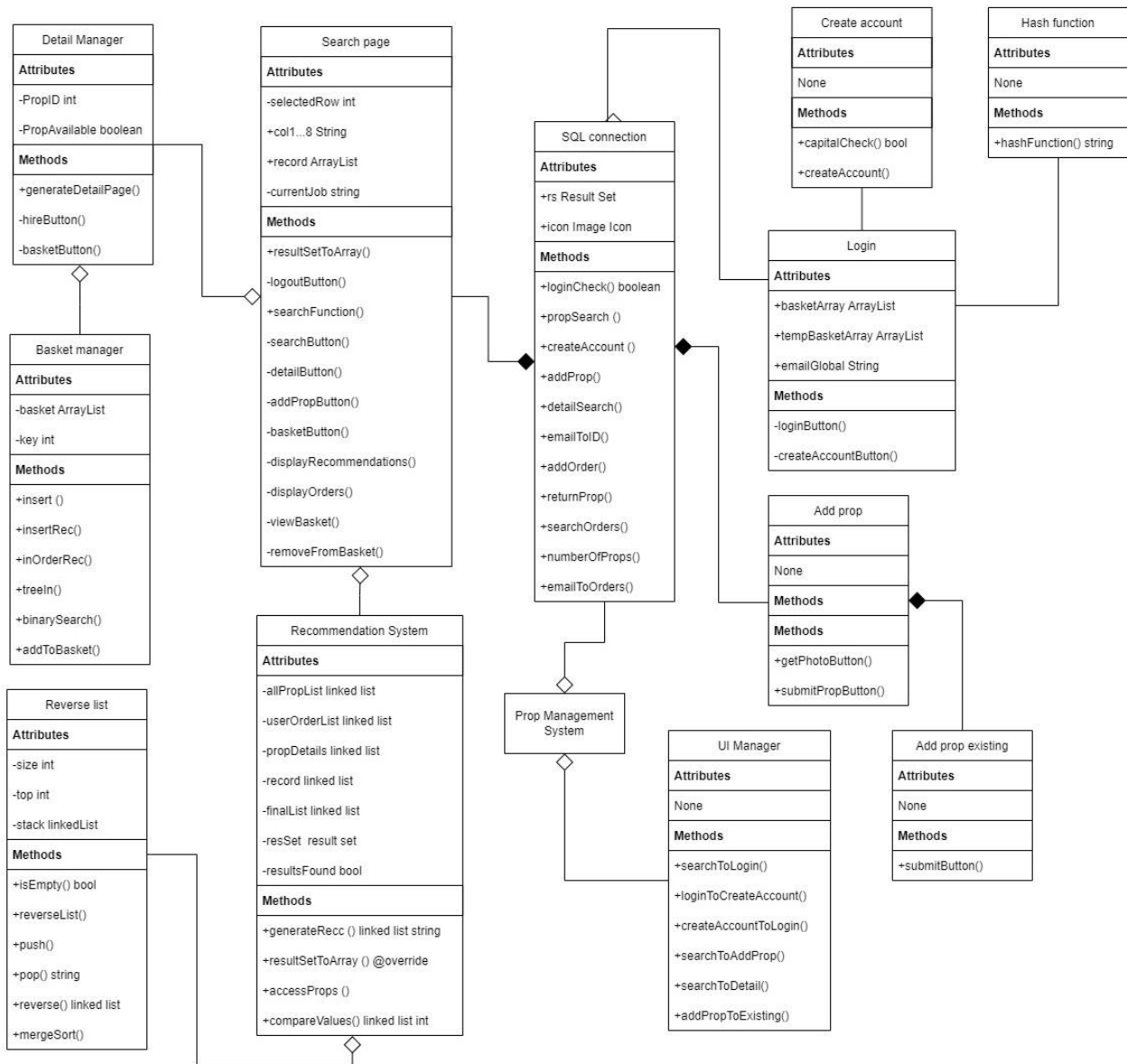
After the first round of designing the user interface I decided to interview my client to see what she thought about the layout. This allowed me to refine the designs and make changes based on what she thought would improve the usability.

- 1. What are the first impressions on the UI such as the layout and colours used?**
From first impressions the layout seems clear with it including the essential elements. While there is somewhat a lack of colour or logos, I believe that would not be an issue since it could make it easier to use. The lack of colour with the surrounding elements also draws the eye to the props which is helpful especially on the search page.
- 2. Does the UX such as interacting with the software seem intuitive?**
Since I am yet to have hands-on experience with the software it is hard to determine how easy it will be to use. However, it does seem to have fairly clear buttons and connections between pages, also the unnecessary functions which are often found on sites have been removed which makes it easier to navigate. One thing which I did notice is that more props could be included on the search page per row since it would make browsing them quicker.
- 3. With the search page, do you think that there should be more ways to search for props?**
I believe that the number currently included should be enough since they are the ones which I typically use. While there could be more options included such as the “size” or “typical room” I believe that they would not be frequently used by me or my team.
- 4. Are there any other parts of the user interface or experience which could be tweaked?**
It sometimes feels as though there is no main page or home button where you can overview what you have been viewing. This could be improved by possibly keeping many of the functions connected to the search page, this would make navigating to the different functions quick and easy since it is all routed through that single main page. An alternative could be to have a more dedicated home page or main page.

Following this interview, the client also recommended I compared the current design to the website “Superhire” which is another platform her team typically uses. I was able to take these critiques into account to change elements of the pages.

Class Diagram:

This class diagram highlights how elements of the project are connected with them all being built around a central prop management system. This allows the connections between the elements to be clear such as which functions and variables are being inherited from the parent object. Some elements cannot exist without the existence of the parent object such as the add existing prop page which is a child of the larger add prop class, this is denoted through the use of the filled diamonds..



Testing Table:

This testing table is a set of possible errors and expected outcomes which can be tested on the programme during and after development. This will reduce the number of bugs and improve the stability of the program, the results of the completed testing table are found in the tuesting section.

Function Name	Data Type	Input Value	Reason for test	Expected outcome
Search	Normal	"red"	To see if entering a characteristic returns the appropriate props	The expected values which the user should be entering are characteristics of props such as year, period and colour. When one of these keywords are entered in the search box all the props with that characteristic should be returned.
Search	Erroneous	"hotel"	To determine if the search function handles an incorrect input by displaying a message to the user	When using the search function the user will typically be entering characteristics of props, however if the user decides to enter an incorrect query the program should inform them which is the expected result.
Search	Erroneous	"Red table"	To see if the results table shows all props which match the characteristics when multiple terms are entered in the search box	When the user attempts to enter multiple search terms the program should consider each of the values and return the appropriate search results.
Login	Normal	username: "admin" password: "pass"	To determine if the login system works as expected when correct login details are entered	When the user logs in using their correct email and password the expected outcome is to alert them that the login was successful through a pop up message and transition the current window to the search page.
Login	Boundary	(Incorrect login details) username:	To determine if the login system	The expected outcome was providing the user

		"admin" password: "pas"	would alert the user to the use of incorrect login details	with an error message which would allow them to remain on the login page and enter correct details
Account creation	Normal	Appropriate login details	To determine if the program would allow the account to be created when all the appropriate login details were entered	When all the valid information is entered to create the account then the user should be informed that it has been created and their details should be added to the database.
Account creation	Erroneous	Invalid account details	To ensure that the database has accurate values when storing the details of users. This test aims to increase the likelihood of accurate employee details	The user is informed that the login details are incorrect and the characteristics which have to be met. Once the popup is closed the window should remain on the login page with all the details still filled in so only the incorrect details have to be remedied.
Hire	Normal	Hire button pressed	To ensure that when all characteristics are correct there are no errors	When an available prop has been selected to hire by a valid account the user should be alerted, the prop should be made unavailable to hire by anyone else and an order should be created.
Hire	Normal	Hire button pressed on an unavailable prop	If the check is not made then an unavailable prop may be hired.	Once a prop has been selected and the hire button has been pressed a check should be made to see if the prop is available to hire. An error message should be displayed to the user to show that the prop is not available to hire.
Basket	Normal data	Add to basket button press	To test if the basket function will generate duplicates in hires and the	When adding a prop to the users basket the program should check if the prop is available to hire and if the prop is

			hiring of unavailable items	not already in the users basket.
Basket	Erroneous data	Add to basket button pressed	To stop invalid items from being added to the basket and hired	When the user attempts to hire a prop which is not available to hire the user should be informed that the prop cannot currently be hired.
Basket	Normal data	Hire button pressed	To ensure that the collective hiring of items through the use of the basket is functioning as expected with the correct number of orders being executed.	Once the basket has been created and there are items to hire, the basket button should be used to automatically hire all these items. This includes changing the status of these props to 0 instead of 1 and creating a record of the hire for each prop. However, if there are no props in the basket the user should be informed and the process should stop.
Basket	Erroneous data	Adding a prop which is already present to the basket	To check whether adding duplicate props to the basket will be detected	When the prop is being added to the basket the program adds the value to the basket it should first check to see if the item has already been added to the basket. If the prop is already in the basket then the item should not be added and the user should be informed through a pop up message.
Prop Detail	Normal input	Detail button pressed	To ensure that when all the correct characteristics are entered that the detail function works as intended	When a prop is selected the prop details page can then be pressed to see further information about the prop, this page also allows the prop to be hired or added to the basket.
Prop detail	Normal data	Prop details button pressed	To make sure the user experience is clear when	When the prop details button is pressed but there is no prop selected the expected outcome is

			interacting with the search function and details button	that the user is informed through a pop up.
Recommendation	Normal data	Recommendation button pressed	To determine whether the recommendation system is working as expected when all the variables are correct	When an account with previous hires uses the recommendation button the user should be shown a message and the recommendations are displayed in the results table. These props have been determined by the similarity of the users previous hires
Recommendation	Boundary data	User pressing recommendation button without a valid history of hires	To ensure that the props recommended are accurate instead of recommending props based on only a single previous hire	When there is no valid hire history for the current user there should be an alert to explain why there are no results
Add Prop	Erroneous data	Image file not selected	To ensure a level of quality in the submissions of new props to the database	When there is no image path selected for the prop photo the prop submission form should not complete. While the other sections of the form are optional, by ensuring an image is submitted it allows the other information to be inferred by the user. The expected outcome is that the user is informed and the process stops.
Basket view	Normal data	View basket button pressed	To test whether the view basket button correctly displays all the items in the basket when the user has created a basket	The results table shows the items which the user has added to their basket while they have been logged

Pseudocode:

After working through the design stage, I was able to work on developing the pseudocode for some of the user-based algorithms I would be including in the project such as the process for hiring and recommending a prop.

Item Hire Pseudocode:

```
1 bool userLogin;
2 bool hireStatus;
3 bool continueHire
4 string itemID;
5 array[][] itemCharacteristics;
6
7 if userLogin == true and hireStatus == false then
8     output("this item is available for hire");
9     output("these are the characteristics: " + array[itemID][COLOUR] + array[itemID][PERIOD] ... );
10    output("do you want to hire this item");
11    continueHire = userInput;
12    if continueHire == true then
13        output("you have successfully hired this item, collect from " + itemCharacteristics[itemID][LOCATION]);
14    else then
15        output("you have decided not to hire, press back to continue browsing");
16 else if userLogin == true and hireStatus == true then
17     output("this item is already hired");
18
19 else if userLogin == false and hireStatus == false then
20     output("incorrect login details please try again");
21
22 else then
23     output("there has been an error please try again")
24
25
```

This is the pseudocode for hire of a prop which would be essential for the new system. The current system uses a similar process however without the use of an algorithm.

The code starts by declaring all the relevant variables, the array `itemCharacteristics` is a 2D array which is used to store all the relevant characteristics of the items which can be found in the Analysis Data Dictionary. While this currently takes the form of a 2D array for the ease of use in pseudocode, this will most likely become a database at a later point in development.

The algorithm checks if the user has the correct details and if the item is available, if this is true then it displays characteristics and checks if the user still wants the item. When this process is complete the user is provided with the location of the prop so they can find the item.

There is also the option to incorrectly input some value which would provide an error message and direct the user back to the original screen to try again.

Recommendation System Pseudocode:

```
1 array [][] itemCharacteristics; #a 2D array for the prop itemCharacteristics
2 string currentItemID           # stores the ID of the item you are currently looking itemRelatedness
3 array [][] itemRelatedness     #an array which stores how similar an item is compared to current item
4 array orderedRelatedness       #an array which orders the list in terms of similarity
5
6 for i = 0 , array[][].len() , i++ then
7     for x = 0 , array[][].len() , x++ then
8         if itemCharacteristics[currentItemID][i] == itemCharacteristics[x][i] then
9             itemRelatedness[x][i] += 1;
10
11 orderedRelatedness = itemRelatedness.sort();
12 output ("this is the recommended item " + orderedRelatedness[1])
13
14
```

This algorithm is designed to recommend other props based on the current item you are looking at. It does this by looking at which props share the most common attributes. For loops in this pseudocode are used to loop through all values in the itemCharacteristics array and compare them to the current item. If the characteristic is the same, then the item gets +1 in relatedness. Once all items have been compared the item with the greatest relatedness is recommended.

Since the top value in relatedness would be the item itself you recommend the second value in the list which is at position 1.

Another way this problem could be tackled could be using machine learning such as SciKit which is a python library, this would allow the values to be represented as vectors and compared in terms of similarity. However, I believe that integrating this approach into the project may be too ambitious and using a more basic algorithm is better suited.

Technical Solution:

Objective 1 Login Page:

NETFLIX LOGIN PAGE

USERNAME / EMAIL:

PASSWORD:

LOGIN

CREATE ACCOUNT

ACCOUNT CREATION:

FIRST NAME

SURNAME

EMAIL

PHONE NUMBER

JOB ROLE LABEL

CURRENT PRODUCTION

ACCOUNT PASSWORD

CREATE ACCOUNT

Message

The login details were correct

OK

LOGIN

CREATE ACCOUNT

EMAIL

PHONE NUMBER

JOB ROLE LABEL

CURRENT PRODUCTION

ACCOUNT PASSWORD

Message

1. Email must contain @
2. Phone number of 11 characters
3. Number and capital in password

OK

CREATE ACCOUNT

Objective 1 Login Page:

The user interface takes the login details and applies the checks using the

```
1. public class loginPage extends javax.swing.JFrame {
2.
3.     //used to temporarily hold the prop ID when applying the tree sort in basket manager
4.     public static ArrayList < Integer > tempBasketArray = new ArrayList < Integer > ();
5.
6.     //the array which stores the ID of all props added to basket during the login session
7.     public static ArrayList < Integer > basketArray = new ArrayList < Integer > ();
8.
9.     //identifies which user is currently logged in
10.    public static String emailGlobal;
11.
12.    public loginPage() {
13.
14.        getContentPane().setBackground(Color.WHITE); //sets the frame background to white
15.        initComponents(); //adds components
16.        this.setExtendedState(this.MAXIMIZED_BOTH); //sets the frame to the size of the window
17.
18.    }
19.
20.    //function applied when the login button is pressed by the user
21.    //used to determine if the login was successful
22.    private void loginActionPerformed(java.awt.event.ActionEvent evt) {
23.
24.        emailGlobal = username.getText(); //stores the email of the user currently logged in
25.        sqlConnection sC = new sqlConnection();
26.
27.        //compares the values entered to those found in the database
28.        if (sC.sqlLoginCheck(username.getText(), password.getText()) == true) {
29.            UImanager uiMan = new UImanager();
30.            uiMan.loginToSearch(); //shows the search page when login successful
31.            this.setVisible(false); //hides this window and shows the search when logged in
32.        }
33.    }
34.
35.    //shows the user the account creation page when the button is pressed
36.    private void createAccountActionPerformed(java.awt.event.ActionEvent evt) {
37.        UImanager UIman = new UImanager();
38.        UIman.loginToCreateAccount();
39.    }
40.
41.
42.    // Variables declaration - do not modify
43.    private javax.swing.JButton createAccount;
44.    private javax.swing.JLabel jLabel1;
45.    private javax.swing.JButton login;
46. }
```

Objective 1 Account Creation Page:

Techniques used:

-Regex

```
1. package netflixpropstorage;
2.
3. import java.awt.Color;
4. import java.util.regex.Matcher;
5. import java.util.regex.Pattern;
6. import javax.swing.JOptionPane;
7.
8. public class createAccount extends javax.swing.JFrame {
9.
10.     public createAccount() {
11.         getContentPane().setBackground(Color.WHITE);
12.         initComponents();
13.         setDefaultCloseOperation(this.DISPOSE_ON_CLOSE);
14.     }
15.
16.     //used to determine the presence of a lower case, capital and number
17.     private static boolean capitalCheck(String password) {
18.         char ch; //variable which temporarily stores the current character
19.         boolean capitalFlag = false; //flag to check for presence of capital
20.         boolean lowerCaseFlag = false; //flag for lower case
21.         boolean numberFlag = false; //flag for numbers
22.
23.         //loops based on the length of the password
24.         for (int i = 0; i < password.length(); i++) {
25.             ch = password.charAt(i);
26.             if (Character.isDigit(ch)) { //if the character is a number then set number flag
27.                 numberFlag = true;
28.             } else if (Character.isUpperCase(ch)) { //if character is a capital then set flag
29.                 capitalFlag = true;
30.             } else if (Character.isLowerCase(ch)) { //if the character is lower case then set flag
31.                 lowerCaseFlag = true;
32.             }
33.             if (numberFlag && capitalFlag && lowerCaseFlag) //check if all flags are true
34.                 return true;
35.         }
36.         return false;
37.     }
38.
39.     private void createAccountButtonActionPerformed(java.awt.event.ActionEvent evt) {
40.         hashFunction hF = new hashFunction();
41.         sqlConnection sC = new sqlConnection();
42.         UIManager UIman = new UIManager();
43.
44.         //regex pattern which checks for the presence of @ in the string
45.         Pattern pattern = Pattern.compile(".*@.*", Pattern.CASE_INSENSITIVE);
46.
47.         //applies regex library using previous regex and the email field string
48.         Matcher matcher = pattern.matcher(emailField.getText());
49.         boolean emailCheck = matcher.find(); //true if @ is found
50.
51.         //hash the password for database storage
52.         String passwordHashed = hF.hashFunction(passwordField.getText());
53.
54.         //checks all fields contain the correct characteristics and only runs if all are true
```

```

55.         if (emailCheck == true && phoneNumberField.getText().length() == 11 &&
passwordField.getText().matches(".*\\d.*") == true && capitalCheck(passwordField.getText())) {
56.             //generates a new user profile using the characteristics entered as parameters
57.             sC.accountCreation(firstNameField.getText(), surnameField.getText(), emailField.getText(),
phoneNumberField.getText(), (String) jobRoleCombo.getSelectedItem(), currentProductionField.getText(),
passwordHashed);
58.             JOptionPane.showMessageDialog(null, "Account Created Successfully"); //output to user
59.             UIman.createAccountToLogin(); //switch window
60.         }
61.
62.         //executed if the previous if check fails
63.         else {
64.             JOptionPane.showMessageDialog(null, "1. Email must contain @ \n 2. Phone number of 11 characters \n 3.
Number and capital in password");
65.         }
66.
67.
68.
69.     }
70.
71.     // Variables declaration - do not modify
72.     private javax.swing.JButton createAccountButton;
73.     private javax.swing.JTextField currentProductionField;
74.     private javax.swing.JTextField emailField;
75.     private javax.swing.JLabel emailLabel;
76.     private javax.swing.JTextField firstNameField;
77.     private javax.swing.JLabel firstNameLabel;
78.     private javax.swing.JLabel jLabel1;
79.     private javax.swing.JComboBox < String > jobRoleCombo;
80.     private javax.swing.JLabel jobRoleLabel;
81.     private javax.swing.JTextField passwordField;
82.     private javax.swing.JTextField phoneNumberField;
83.     private javax.swing.JLabel phoneNumberLabel;
84.     private javax.swing.JLabel phoneNumberLabel1;
85.     private javax.swing.JLabel phoneNumberLabel2;
86.     private javax.swing.JTextField surnameField;
87.     private javax.swing.JLabel surnameLabel;
88.     // End of variables declaration

```

Objective 1 Hash Function:

The hashFunction class is applied when entering and storing the password

Technique used:

-Hashing

```

1. package netflixpropstorage;
2.
3. import java.math.BigInteger;
4. import java.security.MessageDigest;
5. import java.security.NoSuchAlgorithmException;
6.
7. public class hashFunction {
8.
9.     public static String hashFunction(String password) {

```

```
10.     try {
11.
12.         //creates message digest object from library using MD5 encryption
13.         MessageDigest md = MessageDigest.getInstance("MD5");
14.
15.         //stores an array of bytes based on the input password
16.         byte[] messageDigest = md.digest(password.getBytes());
17.
18.         //convert byte array into big integer which stores all the digits
19.         BigInteger no = new BigInteger(1, messageDigest);
20.
21.         //convert message digest into hex value
22.         String hashtext = no.toString(16);
23.         while (hashtext.length() < 32) {
24.             hashtext = "0" + hashtext;
25.         }
26.         return hashtext;
27.     } catch (NoSuchAlgorithmException e) {
28.         throw new RuntimeException(e);
29.     }
30. }
```

Objective 1: SQL Account Creation

By using the details entered on the account creation form page, this SQL function is able to generate a new row in the employees table.

```
1. //creates a new row in the account table using the characteristics entered as parameters
2. public static void accountCreation(String firstName, String surname, String email, String phone, String jobRole,
   String job, String password) {
3.
4.     try {
5.         Class.forName("com.mysql.cj.jdbc.Driver");
6.         Connection connection = DriverManager.getConnection("jdbc:mysql://localhost:3306/propdb", "root",
   "rocket08");
7.
8.         Statement statement = connection.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
   ResultSet.CONCUR_READ_ONLY);
9.
10.        String customQuery = "INSERT INTO employees(firstName , surname, email, phone, jobRole, job, password)
   VALUES('" + firstName + "', '" + surname + "', '" + email + "', '" + phone + "', '" + jobRole + "', '" + job + "', '"
   + password + "')";
11.        statement.executeUpdate(customQuery);
12.    } catch (SQLException e) {
13.        System.out.println("SQL Exception: " + e.toString());
14.    } catch (ClassNotFoundException cE) {
15.        System.out.println("Class Not Found Exception: " + cE.toString());
16.    }
17.
18. }
```

This project used the agile methodology for development of this project since it allowed the client to be closely integrated within the project such as in providing feedback. By doing this it

allowed the project to constantly be moving towards the goals of the end user since the feedback provided time to plan changes for the project.

Agile Methodology for Objective 1:

Sprint 1:

Achieved:

During the first sprint of objective 1 the goal was to create a clear working model of how the account management system would work. This was achieved through the use of a minimal login page and functioning account creation page. This provided a proof of concept for the login system and showed that the fundamentals were working as expected. By the end of the first sprint the account management system looked similar to the planned design which was found in the design section.

Feedback:

From the client I received several points of feedback which I was able to implement and work on for the next sprint. The first point was use of colours, logo and design to make the login system feel closer to the Netflix brand image, through the first sprint, only the default buttons and background were used which made the end product feel generic and bland. Another point was that while the user interface was simple, it could be expanded upon by maximising space since the buttons were fairly small in comparison to the size of the window. Finally with the account creation page the client was concerned that it would be possible to enter incorrect values which they found after testing the initial plan themselves.

Goals for the next sprint:

Moving forward into sprint 2, the main goal is to improve both the UI and the UX by improving the size and colour of elements on the page, this will make the user feel more confident in the software and corresponds to the client feedback received. Once this objective has been achieved, I plan to improve the user detail form so that it is less susceptible to mistakes or issues from the user entering values.

Sprint 2:

Achieved:

The goals laid out following the initial sprint were primarily pursued which is why I believe this sprint phase was successful. The colours were set to match those used for all Netflix branding such as the dark red with a contrasting background. The layout of the login and create account buttons was also altered to fill more of the screen. Finally, all fields in the create account section had some form of check applied to them such as the email field including '@' and the title areas being non null. The user interface can be found in the final documentation of the login page found in the technical solution.

Feedback:

The client was impressed with the login system following the changes through sprint 2. Notably the change in colours and the inclusion of the logo allowed the user to feel confident and professional when using the programme. Also, the user experience was slightly improved through the use of the changes made to the buttons, despite using more colours it did not make the interface overwhelming or confusing. However, one concern the client had was that all fields for the account creation page had some form of check on them, this could cause issues in the future since it may not always be relevant for employees to include job title.

Goals for the next sprint:

Due to the client being overall pleased with the login page there are few changes which need to be made. The only noticeable change which will have to be made to ensure the quality of the account management system is to remove some of the checks on the fields found in the account creation page. In addition, thorough testing of all elements of the program is necessary to avoid bugs and issues which may have not been found yet.



Objective 2 Search functions:

NETFLIX SEARCH PAGE

LOG OUT HIRE BASKET RECOMMENDATIONS ADD PROP PAGE ORDERS

VIEW BASKET REMOVE FROM BASKET

chair SEARCH PROP DETAILS PAGE


Prop ID	Prop Type	Prop Colour	Prop Image
1	Chair	Orange	
2	Chair	Red	

NETFLIX SEARCH PAGE

LOG OUT HIRE BASKET RECOMMENDATIONS ADD PROP PAGE ORDERS

VIEW BASKET REMOVE FROM BASKET

red chair SEARCH PROP DETAILS PAGE

Prop ID	Prop Type	Prop Colour	Prop Image
2	Chair	Red	

PROP DETAILS



PROP INDEX: 9
TYPE: CHAIR
COLOUR: BROWN
PERIOD: PERIOD
DATE: 1970
DESCRIPTION: NULL

HIRE PROP
ADD TO BASKET
RETURN TO SEARCH

HIRE BASKET RECOMMENDATIONS ADD PROP PAGE ORDERS

VIEW BASKET REMOVE FROM BASKET

hotel SEARCH PROP DETAILS PAGE

Message
this search returned no results
OK

Prop ID	Prop Type	Prop Colour	Prop Image
---------	-----------	-------------	------------

Objective 2 Search Functions:

Techniques used:

-Use of OOP (inheritance / polymorphism)

```

1. public class searchPage extends javax.swing.JFrame {
2.
3.     int selectedRow;
4.     public String searchBoxValue = "null";
5.
6.     //set of public variables which store the characteristics of prop. used for generating the detail page
7.     String col1,col2,col3,col4,col5,col6,col7,col8;
8.
9.     public searchPage() {
10.         getContentPane().setBackground(Color.WHITE);
11.         initComponents();
12.         this.setExtendedState(this.MAXIMIZED_BOTH);
13.     }
14.
15.
16.     ArrayList < String > record = new ArrayList < String > (); //the arraylist which will store the results from the
rs
17.
18.     //this func converts a result set into an array and then returns the arraylist
19.     public void resultSetToArray(ResultSet resSet) {
20.         record.clear(); //empties the arraylist
21.         try {
22.             ResultSetMetaData metaData = (ResultSetMetaData) resSet.getMetaData();
23.             int columns = metaData.getColumnCount(); //get meta data and store the number of columns
24.             while (resSet.next()) { //loops based on the number of results
25.
26.                 for (int i = 1; i < columns + 1; i++) {
27.                     String value = resSet.getString(i);
28.                     record.add(value);
29.                 }
30.             }
31.         } catch (SQLException ex) {
32.             Logger.getLogger(searchPage.class.getName()).log(Level.SEVERE, null, ex);
33.             record.add("null");
34.
35.         }
36.     }
37.
38.     public void searchFunction(String searchText) {
39.         sqlConnection sC = new sqlConnection();
40.         sC.sqlSearchProp(searchText); //applies prop search when button is pressed using the textbox value
41.         searchBoxValue = searchText;
42.
43.         ArrayList < String > result1 = new ArrayList < String > (); //used to store the values from search
44.         resultSetToArray(sC.getRs()); //populates result1 with the results
45.         result1 = record; //record is an array list which is declared at the start it stores the results
46.
47.         //this block overrides the getColumnClass so that the 3rd column is read as an image rather than file path
48.         DefaultTableModel tM = new DefaultTableModel() {
49.             @Override
50.             public Class getColumnClass(int column) {
51.                 if (column == 3) return ImageIcon.class;
52.                 else return Object.class;
53.             }
54.         };
55.

```

```
56.         //generates the columns for the table
57.         tM.addColumn("Prop ID");
58.         tM.addColumn("Prop Type");
59.         tM.addColumn("Prop Colour");
60.         tM.addColumn("Prop Image");
61.
62.         //checks if there were no results to return error message
63.         if (result1.isEmpty()) {
64.             JOptionPane.showMessageDialog(null, "this search returned no results");
65.         }
66.
67.         //block used to generate the table model which fill the results table
68.         for (int i = 0; i <= result1.size() / 8 - 1; i++) { //loops based on the number of results
69.             ImageIcon img = new ImageIcon(result1.get(i * 8 + 6)); // load the image to a ImageIcon
70.             Image image = img.getImage(); // transform it
71.             Image newimg = image.getScaledInstance(120, 120, java.awt.Image.SCALE_SMOOTH); // scale it the smooth
way
72.             tM.addRow(new Object[] {
73.                 result1.get(i * 8), result1.get(i * 8 + 1), result1.get(i * 8 + 2), new ImageIcon(newimg)
74.             });
75.             // ^^ creates a new row by drawing values from the result1 array which contains results
76.         }
77.         resultsTable.setModel(tM); //set the model to the results table
78.
79.
80.         //iterates through the rows and sets the row height so that the images are not cropped when rendered
81.         for (int row = 0; row < resultsTable.getRowCount(); row++) {
82.             int rowHeight = resultsTable.getRowHeight();
83.
84.             for (int column = 0; column < resultsTable.getColumnCount(); column++) {
85.                 Component comp = resultsTable.prepareRenderer(resultsTable.getCellRenderer(row, column), row,
column);
86.                 rowHeight = Math.max(rowHeight, comp.getPreferredSize().height);
87.             }
88.             resultsTable.setRowHeight(row, rowHeight);
89.
90.         }
91.     }
92.
93.     //used to search for props and create the results table
94.     private void searchButtonActionPerformed(java.awt.event.ActionEvent evt) {
95.
96.         searchFunction(searchBox.getText());
97.
98.     }
99.
100.    //used to create a page which shows the details of the selected prop
101.    private void detailButtonActionPerformed(java.awt.event.ActionEvent evt) {
102.
103.        if (resultsTable.getSelectionModel().isSelectionEmpty()) {
104.            JOptionPane.showMessageDialog(null, "There is no prop selected");
105.        } else {
106.
107.            sqlConnection sC = new sqlConnection();
108.            detailManager dM = new detailManager();
109.
110.            try {
111.
112.                String search = (String) resultsTable.getValueAt(resultsTable.getSelectedRow(), 0);
113.                //^^ gets the prop id from the generated table
114.
```

```

115.         ResultSet resSet = sC.detailSearch(search); // uses the prop id from above to sql search
116.         resSet.first(); //moves the result set current row to the first line
117.
118.         //sets the values of each characteristic which can then be passed into the detail page generator
119.         col1 = resSet.getString("propID");
120.         col2 = resSet.getString("propType");
121.         col3 = resSet.getString("colour");
122.         col4 = resSet.getString("period");
123.         col5 = resSet.getString("propDate");
124.         col6 = resSet.getString("propDescription");
125.         col7 = resSet.getString("images");
126.         col8 = resSet.getString("availableToHire");
127.
128.         dM.generateDetailPage(col1, col2, col3, col4, col5, col6, col7, col8); //creates new page with
        details
129.
130.         } catch (SQLException ex) {
131.             Logger.getLogger(searchPage.class.getName()).log(Level.SEVERE, null, ex);
132.         }
133.     }
134. }
135.
136.

```

Objective 2 SQL Functions:

When the user enters values into the search box an SQL function has to be applied which returns the appropriate prop characteristics

```

1. //returns all the characteristics of the selected props based on a characteristic
2. public void sqlSearchProp(String prop) {
3.
4.     //checks if multiple words have been entered
5.     if (prop.contains(" ")) {
6.
7.         String[] words = prop.split(" "); //creates a list of words from the search values
8.
9.         try {
10.             Class.forName("com.mysql.cj.jdbc.Driver");
11.             Connection connection = DriverManager.getConnection("jdbc:mysql://localhost:3306/propdb", "root",
                "rocket08");
12.             //Statement statement = connection.createStatement();
13.             Statement statement = connection.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                ResultSet.CONCUR_READ_ONLY);
14.             String customQuery = "SELECT * FROM props WHERE'" + words[0] + "'IN(propID, propType, colour, period,
                propDate, propDescription)";
15.
16.
17.             //runs a for loop based on the number of words entered which creates a statement with all the terms
18.             for (int i = 0; i < words.length; i++) {
19.                 customQuery = customQuery.concat("OR'" + words[i] + "'IN(propID, propType, colour, period, propDate,
                propDescription)");
20.             }
21.
22.             //the result set which stores the result from the search
23.             rs = statement.executeQuery(customQuery);
24.
25.

```

```

26.         } catch (SQLException e) {
27.             System.out.println("SQL Exception: " + e.toString());
28.         } catch (ClassNotFoundException cE) {
29.             System.out.println("Class Not Found Exception: " + cE.toString());
30.         }
31.     }
32.
33.     //else block runs when there is only one word entered
34.     else {
35.         try {
36.             Class.forName("com.mysql.cj.jdbc.Driver");
37.             Connection connection = DriverManager.getConnection("jdbc:mysql://localhost:3306/propdb", "root",
"rocket08");
38.             //Statement statement = connection.createStatement();
39.             Statement statement = connection.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_READ_ONLY);
40.             String customQuery = "SELECT * FROM props WHERE'" + prop + "'IN(propID, propType, colour, period,
propDate, propDescription)";
41.             rs = statement.executeQuery(customQuery);
42.
43.
44.         } catch (SQLException e) {
45.             System.out.println("SQL Exception: " + e.toString());
46.         } catch (ClassNotFoundException cE) {
47.             System.out.println("Class Not Found Exception: " + cE.toString());
48.         }
49.     }
50. }

```

Objective 2 Detail Page:

To display the details of the prop a new window is generated to display all the required elements, these are generating using the detail manager class.

```

1.  package netflixpropstorage;
2.
3.  import java.awt.Color;
4.  import java.awt.Font;
5.  import java.awt.Image;
6.  import java.awt.event.ActionEvent;
7.  import java.awt.event.ActionListener;
8.  import java.sql.ResultSet;
9.  import java.sql.SQLException;
10. import java.text.ParseException;
11. import java.util.logging.Level;
12. import java.util.logging.Logger;
13. import javax.swing.*;
14.
15. //a class used to generate the page showing a selected prop
16. public class detailManager {
17.
18.     String propID; //stores the id of the prop being displayed
19.     String propAvailable; //determines whether the props is currently on hire
20.
21.     public void generateDetailPage(String col1, String col2, String col3, String col4, String col5, String col6,
String col7, String col8) {
22.         JFrame f = new JFrame(); //creating instance of JFrame

```

```
23.     propID = col1; //sets the id of the prop
24.     propAvailable = col8; //stores the availability of the prop
25.     f.getContentPane().setBackground(Color.WHITE); //set background to white
26.
27.     //sets the labels to the characteristics of the prop found in the parameters
28.     JLabel detail1 = new JLabel("Prop index: " + col1); //sets the label to the prop id
29.     detail1.setBounds(700, 130, 1000, 40); //determines the size and position of the label
30.     detail1.setFont(new Font("Bebas Neue", Font.PLAIN, 35)); //sets the font and size
31.     detail1.setForeground(new Color(229, 9, 20)); //colour of the label using the rgb value from netflix
32.
33.     JLabel detail2 = new JLabel("Type: " + col2);
34.     detail2.setBounds(700, 190, 1000, 40);
35.     detail2.setFont(new Font("Bebas Neue", Font.PLAIN, 35));
36.     detail2.setForeground(new Color(229, 9, 20));
37.
38.     JLabel detail3 = new JLabel("Colour: " + col3);
39.     detail3.setBounds(700, 250, 1000, 40);
40.     detail3.setFont(new Font("Bebas Neue", Font.PLAIN, 35));
41.     detail3.setForeground(new Color(229, 9, 20));
42.
43.     JLabel detail4 = new JLabel("Period: " + col4);
44.     detail4.setBounds(700, 310, 1000, 40);
45.     detail4.setFont(new Font("Bebas Neue", Font.PLAIN, 35));
46.     detail4.setForeground(new Color(229, 9, 20));
47.
48.     JLabel detail5 = new JLabel("Date: " + col5);
49.     detail5.setBounds(700, 370, 1000, 40);
50.     detail5.setFont(new Font("Bebas Neue", Font.PLAIN, 35));
51.     detail5.setForeground(new Color(229, 9, 20));
52.
53.     JLabel detail6 = new JLabel("Description: " + col6);
54.     detail6.setBounds(700, 430, 1000, 40);
55.     detail6.setFont(new Font("Bebas Neue", Font.PLAIN, 35));
56.     detail6.setForeground(new Color(229, 9, 20));
57.
58.     //creates the image of the prop
59.     ImageIcon icon = new ImageIcon(col7); //stores the file in a icon object
60.     Image scaleImage = icon.getImage().getScaledInstance(450, 450, Image.SCALE_DEFAULT); //scales the image to
the right size
61.     icon = new ImageIcon(scaleImage); //sets the image icon to the the icon so it can be assigned to a label
62.     JLabel detail7 = new JLabel(icon); //sets the label to the value of the icon
63.     detail7.setBounds(110, 110, 450, 450); //size and position of the image
64.
65.     //title text
66.     JLabel detail8 = new JLabel("Prop Details");
67.     detail8.setBounds(440, 30, 1000, 40); //x axis, y axis, width, height
68.     detail8.setFont(new Font("Bebas Neue", Font.PLAIN, 50));
69.     detail8.setForeground(new Color(229, 9, 20));
70.
71.     f.add(detail1); //adding button in JFrame
72.     f.add(detail2);
73.     f.add(detail3);
74.     f.add(detail4);
75.     f.add(detail5);
76.     f.add(detail6);
77.     f.add(detail7);
78.     f.add(detail8);
79.
80.
81.     JButton b = new JButton("Hire Prop");
82.     b.setBounds(700, 500, 160, 40);
```

```
83.         b.setFont(new Font("Bebas Neue", Font.PLAIN, 20));
84.         b.setForeground(new Color(229, 9, 20));
85.         b.setBackground(Color.white);
86.
87.         JButton c = new JButton("Add to basket");
88.         c.setBounds(700, 550, 160, 40);
89.         c.setFont(new Font("Bebas Neue", Font.PLAIN, 20));
90.         c.setForeground(new Color(229, 9, 20));
91.         c.setBackground(Color.white);
92.
93.         JButton d = new JButton("Return to search");
94.         d.setBounds(700, 600, 160, 40);
95.         d.setFont(new Font("Bebas Neue", Font.PLAIN, 20));
96.         d.setForeground(new Color(229, 9, 20));
97.         d.setBackground(Color.white);
98.
99.         b.addActionListener(new ActionListener() {
100.             public void actionPerformed(ActionEvent e) { //executed when the hire button is pressed
101.
102.                 //confirms with the user
103.                 int dialogButton = JOptionPane.YES_NO_OPTION;
104.                 int dialogResult = JOptionPane.showConfirmDialog(null, "Are you sure you want to hire this prop",
"Warning", dialogButton);
105.
106.                 if (dialogResult == JOptionPane.YES_OPTION) {} else {
107.                     JOptionPane.showMessageDialog(null, "Prop has not been hired");
108.                 }
109.                 sqlConnection sC = new sqlConnection();
110.                 loginPage lP = new loginPage();
111.                 ResultSet resSet = sC.emailToID(loginPage.emailGlobal); //stores the id of the user
112.                 try {
113.                     if (propAvailable.equals("1")) { //checks if the prop is available to hire
114.                         resSet.first(); //sets the current row to the first row
115.                         String currentJob = resSet.getString("job"); //retrieves the current job
116.                         String userID = resSet.getString("userID"); //retrieves the user id
117.                         sC.addOrder(propID, userID, currentJob); //generates an order
118.                         JOptionPane.showMessageDialog(null, "This prop has been hired by user with email: " +
lP.emailGlobal);
119.                     } else {
120.                         JOptionPane.showMessageDialog(null, "This prop is not available to hire currently");
121.                     }
122.
123.
124.                 } catch (SQLException ex) {
125.                     Logger.getLogger(detailManager.class.getName()).log(Level.SEVERE, null, ex);
126.                 } catch (ParseException ex) {
127.                     Logger.getLogger(detailManager.class.getName()).log(Level.SEVERE, null, ex);
128.                 }
129.             }
130.
131.
132.         });
133.
134.         c.addActionListener(new ActionListener() {
135.             public void actionPerformed(ActionEvent e) { //executed when the add to basket button is pressed
136.
137.                 int dialogButton = JOptionPane.YES_NO_OPTION;
138.                 int dialogResult = JOptionPane.showConfirmDialog(null, "Are you sure you want to add this prop to
basket", "Warning", dialogButton);
139.                 if (dialogResult == JOptionPane.YES_OPTION) {
140.                     basketManager bM = new basketManager();
```

```

141.         loginPage LP = new loginPage();
142.         int tempInt = Integer.parseInt(col1); //used to hold the ID of the prop
143.         if (propAvailable.equals("1")) {
144.             if (bM.addToBasket(LP.basketArray, tempInt) == -1) { //used as a check to see if the item
has already been added
145.                 //returns -1 if the item is not found
146.                 JOptionPane.showMessageDialog(null, "This prop has been added to basket");
147.                 LP.basketArray.add(tempInt);
148.             } else {
149.                 JOptionPane.showMessageDialog(null, "This prop is already in the basket so has not been
added");
150.             }
151.         } else {
152.             JOptionPane.showMessageDialog(null, "This prop is not available to hire currently");
153.         }
154.
155.
156.     } else {
157.         JOptionPane.showMessageDialog(null, "This prop has not been added to basket");
158.     }
159.
160.
161. }
162.
163. });
164.
165. //runs when the return search to button pressed
166. d.addActionListener(new ActionListener() {
167.     public void actionPerformed(ActionEvent e) {
168.         f.setVisible(false); //hides the current window to show the search page
169.     }
170. });
171.
172.
173. f.add(b); //adds the buttons to the frame
174. f.add(c);
175. f.add(d);
176.
177. f.setSize(1080, 720);
178. f.setLayout(null); //using no layout managers
179. f.setVisible(true); //making the frame visible
180. f.setDefaultCloseOperation(f.DISPOSE_ON_CLOSE);
181. }
182. }

```

Objective 2 Add order:

When the hire button is pressed by the user, an order is created which many of the tables through the use of primary keys.

Technique used: Complex data model in database (several interlinked tables)

```

1. //adds a new row to the order table
2. public static void addOrder(String propID, String userID, String shootID) throws ParseException {
3.
4.     try {
5.         Class.forName("com.mysql.cj.jdbc.Driver");

```

```

6.      Connection connection = DriverManager.getConnection("jdbc:mysql://localhost:3306/propdb", "root",
"rocket08");
7.      Statement statement = connection.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_READ_ONLY);
8.
9.      //used to add a new row to the orders table using the appropriate values
10.     String customQuery = "INSERT INTO orders(propID, userID, shootID) VALUES('" + propID + "' , '" + userID +
"'," + shootID + "');";
11.     statement.executeUpdate(customQuery);
12.
13.     //changes the availability of the prop so that it cannot be hired again until it is returned
14.     String customQuery2 = "UPDATE props SET availableToHire = FALSE WHERE propID = '" + propID + "';";
15.     statement.executeUpdate(customQuery2);
16.
17.     //stores the maximum orderID and stores it in an sql variable
18.     String customQuery3 = "set @orderId =(SELECT orderID FROM propdb.orders WHERE orderID=(SELECT max(orderID)
FROM propdb.orders)); ";
19.     statement.executeUpdate(customQuery3);
20.
21.     //stores the current job of the user hiring the props
22.     String customQuery5 = "set @job =(SELECT job FROM propdb.employees WHERE employees.userID =' " + userID +
"');";
23.     statement.executeUpdate(customQuery5);
24.
25.     //retrieves and stores the shoot id from the shoot location based on the users current job
26.     String customQuery6 = "set @shootId =(SELECT shootID FROM propdb.shootlocation WHERE
shootlocation.production = @job);";
27.     statement.executeUpdate(customQuery6);
28.
29.     //gets the number of days based on the shoot id
30.     String customQuery7 = "select durationDays from shootlocation where shootId = @shootId";
31.
32.     //stores the duration of the shoot for the current user
33.     ResultSet durationDays;
34.     durationDays = statement.executeQuery(customQuery7);
35.     durationDays.first();
36.
37.     //sets the format of the dates and creates a new date object
38.     DateTimeFormatter dtf = DateTimeFormatter.ofPattern("yyyy/MM/dd");
39.
40.     //gets the local date
41.     LocalDateTime now = LocalDateTime.now();
42.
43.     //stores the current date as a string based on the previously set format
44.     String hireDate = dtf.format(now);
45.
46.     String returnDate = hireDate; //stores the date which the prop will be returned on
47.     SimpleDateFormat sdf = new SimpleDateFormat("yyyy/MM/dd");
48.     Calendar c = Calendar.getInstance();
49.     c.setTime(sdf.parse(returnDate)); //formats and sets the return date
50.
51.     //adds to the return date based on the duration of days for the shoot
52.     c.add(Calendar.DATE, Integer.parseInt(durationDays.getString("durationDays")));
53.     returnDate = sdf.format(c.getTime()); // dt is now the new date
54.
55.     //stores the information about this prop hire in the order detail table
56.     String customQuery4 = "INSERT INTO orderDetail(hireDate, returnDate, quantity, orderID, shootID) VALUES('" +
hireDate + "' , '" + returnDate + "' , '1' , " + @orderId + " , @shootId);";
57.     statement.executeUpdate(customQuery4);
58.
59.     } catch (SQLException e) {

```



```

60.     System.out.println("SQL Exception: " + e.toString());
61. } catch (ClassNotFoundException cE) {
62.     System.out.println("Class Not Found Exception: " + cE.toString());
63. }
64.
65. }
66.

```

orderID	propID	userID	shootID	orderDetailID
1	2	1	Crown	1
				NULL

The change made to the order table through the execution of the SQL statement

Objective 2 Convert Email to Orders:

Using an SQL function the orders of a user are retrieved to then be stored in an array. Allows the comparisons to be made.

Techniques used:

-Cross table parameterised SQL (INNER JOIN)

```

1.  //takes the user email as a parameter and returns their orders
2.  public ResultSet emailToOrders(String email) {
3.
4.      try {
5.          Class.forName("com.mysql.cj.jdbc.Driver");
6.          Connection connection = DriverManager.getConnection("jdbc:mysql://localhost:3306/propdb", "root",
            "rocket08");
7.          Statement statement = connection.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
            ResultSet.CONCUR_READ_ONLY);
8.
9.          //stores the employee id by converting from the email to the id
10.         String customUpdate = "set @EmpId = (select userID from propdb.employees where employees.email = '" + email
            + "' );";
11.         statement.executeUpdate(customUpdate);
12.
13.         //selects all the orders based on the previously stored employee id
14.         String customQuery = "SELECT Orders.*, props.propID, props.propType, props.colour, props.period,
            props.propDate FROM Orders INNER JOIN props ON orders.propID=props.propID where orders.userID = @EmpID order by
            orderID DESC;";
15.         statement.executeQuery(customQuery);
16.         rs = statement.executeQuery(customQuery);
17.
18.     } catch (SQLException e) {
19.         System.out.println("SQL Exception: " + e.toString());
20.     } catch (ClassNotFoundException cE) {
21.         System.out.println("Class Not Found Exception: " + cE.toString());
22.     }
23.     return rs;
24. }

```

Agile Methodology for Objective 2:

Sprint 1:

Achieved:

During the first stage of development of the search system we were able to create a search page which would display images of props accompanied by a small amount of information about the prop such as the colour. In addition the user would be able to select a prop and use the prop detail page to find more detailed information about the prop such as a description and the era.

Feedback:

While many of the functions of the search system worked as intended there was feedback from the client surrounding the precise details. The first point which the client noticed was the inability to enter multiple terms in the search box as it would only return searches based on the first term, the client felt that this was an important addition as she often searches using several terms. Another point was the ability to determine more information about the prop from the search page rather than having to click the details page frequently since the client found this time consuming.

Since the project is also to be used by other stake holders such as the prop master, I felt that it was important to incorporate other users into the feedback. By including the prop master in the testing, it provides another perspective on the project objectives. An example of feedback was that he felt the user should be able to search based and see the prop ID since it would make the job of those managing the props easier as that would be frequently queried.

Goals for the next sprint:

To improve the ability to search using multiple terms I plan to implement some form of aggregate search which can incorporate the search process multiple times. In addition, I need to find a manner to display the information in a more efficient manner such as through the use of a pop up or hover box which shows a greater level of detail surrounding the prop. Finally, the inclusion of the ID with the process is an important change such as displaying it in the prop detail page.

Sprint 2:

Achieved:

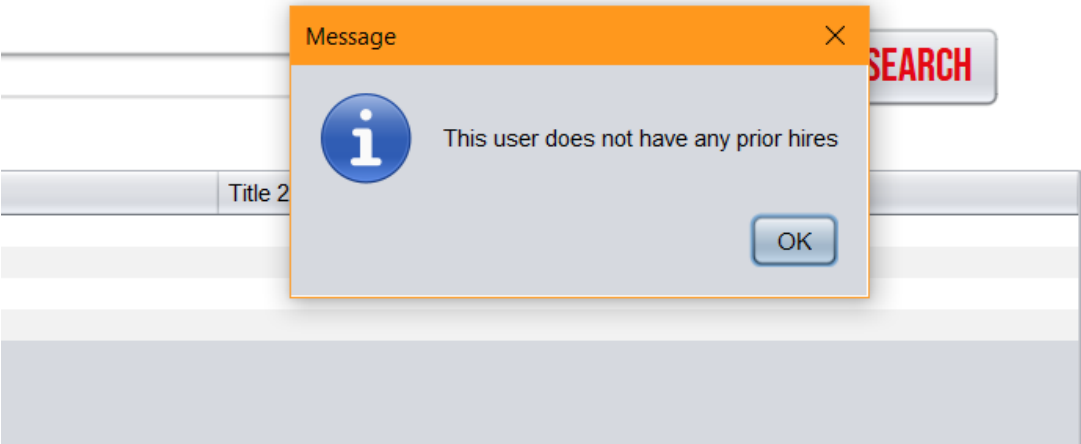
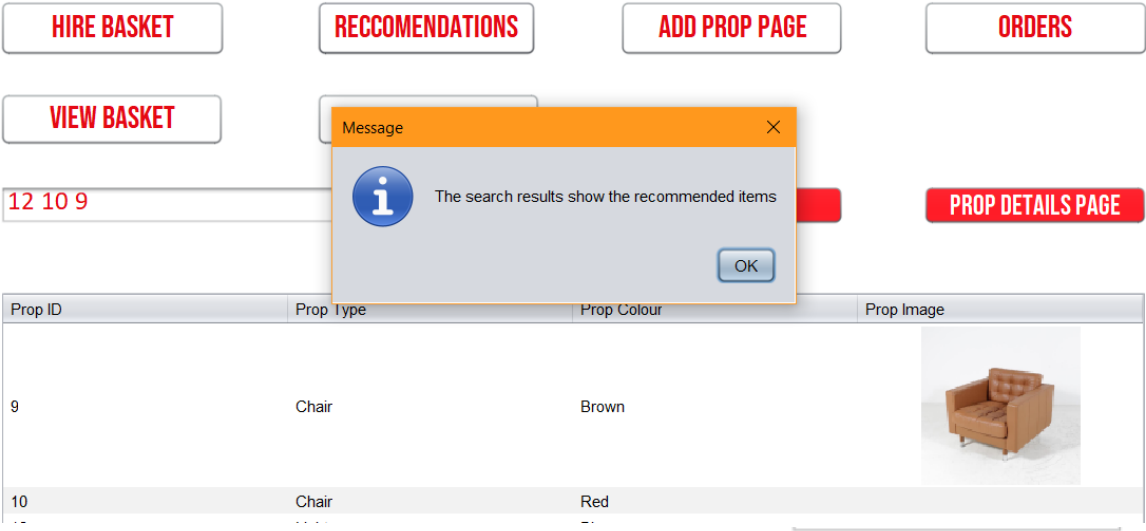
During this sprint multiple search terms were made possible through determining the number of words entered in the search box and repeating the search for each word which increases the quality of the results. To improve the amount of information displayed to the user when searching for items I decided to switch the display method from a set of large images to a table system since it still includes the image of the prop while including the amount of information

which can be included. Finally, I was able to improve the use of the prop ID through including the information in both the results table and the prop information page which makes it easier to use for those managing the inventory.

Feedback:

The client was pleased with the changes made to the search system, since it is used every day by many members of the organisation, they felt that it was important to have it as refined as possible. The user preferred the display of information through the use of the table since it shows all the necessary information in an efficient way. They also felt that the multiple search terms greatly improved the useability and efficiency of the program. Finally, the prop master was surprised about how the prop ID was quickly integrated into almost all parts of the search process.

Objective 3 Recommendation System:



Objective 3 Display Recommendations:

This generates the recommendations and then displays them using the search system which can be found in objective 2 or in the appendix.

```

1. //displays the automatic recommendations to the user
2. private void recommendationButtonActionPerformed(java.awt.event.ActionEvent evt)
   {
3.
4.     recommendationPage rP = new recommendationPage();
5.     loginPage lP = new loginPage();
6.
7.     //generates a list of recommendations of prop IDs using the users email
8.     String listString = String.join(" ", rP.generateRecc(lP.emailGlobal));
9.
10.    searchBox.setText(listString);    //sets the value of the search box to the recommendations IDs
11.
12.    //this checks if there are any recommendations and informs the user
13.    //if the generated list starts with a null value then no recommendations were found
14.    if(rP.finnalList.get(0).charAt(0) == '0'){
15.        searchBox.setText("");
16.        JOptionPane.showMessageDialog(null, "There were no recommendations found\nCheck your email and orders");
17.    }
18.    }
19.    else{
20.        searchFunction(searchBox.getText());    //displays the recommended props with the search function
21.        JOptionPane.showMessageDialog(null, "The search results show the recommended items");
22.    }
23. }

```

Objective 3 Generate Recommendations:

Element of the search page is mostly for outputting the results of the recommendations. The processing and generating is done with the recommendation system class:

Techniques used:

- Complex user defined algorithm (pattern matching)
- User defined use of OOP (polymorphism / override)

```

1. import com.mysql.cj.jdbc.result.ResultSetMetaData;
2. import java.sql.ResultSet;
3. import java.sql.SQLException;
4. import java.util.LinkedList;
5. import java.util.logging.Level;
6. import java.util.logging.Logger;
7. import javax.swing.JOptionPane;
8.
9. public class recommendationSystem extends searchPage {
10.
11.     LinkedList < String > allPropList = new LinkedList < > ();
12.     LinkedList < String > userOrdersList = new LinkedList < > ();
13.     LinkedList < String > userOrdersList2 = new LinkedList < > ();
14.     LinkedList < String > propDetails1 = new LinkedList < > ();
15.     LinkedList < String > propDetails2 = new LinkedList < > ();
16.     LinkedList < String > propDetails3 = new LinkedList < > ();
17.     LinkedList < String > record = new LinkedList < > (); //the linked list which will store the results from the rs

```

```

18.    LinkedList < String > publicRecommendationList = new LinkedList < > (); //stores the results of the function in
    a linked list
19.    LinkedList < String > finallist = new LinkedList < > (); //stores the final set of values
20.    ResultSet resSet;
21.    boolean resultsFound = true;
22.    int temporary = 0;
23.
24.    //used to create a linked list which stores the ID of props which are most similar
25.    public LinkedList < String > generateRecc(String emailField) {
26.
27.        sqlConnection sC = new sqlConnection();
28.
29.        accessProps(emailField); //calls the access props function using the email found in parameter
30.        LinkedList < Integer > similarScores = new LinkedList < > (); //stores the similarity score of each prop
31.
32.        for (int i = 0; i < sC.numberOfProps() + 1; i++) { //this creates a linked list with the size equal to the
    number of total props
33.            similarScores.add(0); //fills the list with blank values which can be replaced
34.        }
35.
36.        compareValues(propDetails1, similarScores); //runs the programme for prop1
37.        compareValues(propDetails2, similarScores); //same but for prop 2
38.        compareValues(propDetails3, similarScores); //same but for prop 3
39.
40.        for (int i = 0; i < similarScores.size(); i++) {
41.            finallist.add(similarScores.get(i).toString()); //adds the values from similar scores to the final list
42.            String temp = finallist.get(i) + String.format("%03d", i); //adds the index to the end and uses 3 digits
43.            finallist.set(i, temp); //sets each index to the value on the previous line which is the similarity +
    the index
44.        }
45.
46.        reverseList rL = new reverseList(finallist.size()); //creates an instance of the reverse list class
47.        rL.mergeSort(finallist, 0, finallist.size() - 1); //orders the final list using a merge sort based on the
    similarity values
48.        rL.reverse(finallist, finallist.size()); //the previous sort is smallest to largest so this reverses using
    stack
49.
50.        // finallist.get() means get that value from final list such as 2003
51.        // substring selects only the last 3 digits
52.        // replace first removes all leading zeros using regex
53.
54.        publicRecommendationList.add(finallist.get(0).substring(finallist.get(0).length() -
    3).replaceFirst("^0+(?!$)", ""));
55.
56.        publicRecommendationList.add(finallist.get(1).substring(finallist.get(0).length() -
    3).replaceFirst("^0+(?!$)", ""));
57.
58.        publicRecommendationList.add(finallist.get(2).substring(finallist.get(0).length() -
    3).replaceFirst("^0+(?!$)", ""));
59.
60.        this.setVisible(false);
61.        return publicRecommendationList;
62.    }
63.
64.    @Override //polymorphs the similar function found in searchPage but works for linked lists
65.    public void resultSetToArray(ResultSet resSet) {
66.        record.clear(); //empties the linked list
67.
68.        try {
69.
70.            ResultSetMetaData metaData = (ResultSetMetaData) resSet.getMetaData();

```

```
71.         int columns = metaData.getColumnCount(); //get meta data and store the number of columns
72.         resSet.first(); //moves the result set to the first line
73.
74.         //a loop which runs based on the number of columns to add the results to record
75.         for (int i = 1; i < columns + 1; i++) {
76.             String value = resSet.getString(i);
77.             record.add(value);
78.         }
79.
80.         while (resSet.next()) { //loops based on the number of results
81.
82.             for (int i = 1; i < columns + 1; i++) {
83.                 String value = resSet.getString(i);
84.                 record.add(value);
85.             }
86.         }
87.     } catch (SQLException ex) {
88.         Logger.getLogger(searchPage.class.getName()).log(Level.SEVERE, null, ex);
89.         record.add("null");
90.
91.     }
92.
93. }
94.
95. //stores the characteristics of the props which the user most recently hired
96. public void accessProps(String email) {
97.
98.     try {
99.         sqlConnection sC = new sqlConnection();
100.        resSet = sC.emailToID(email); //stores the id of the user based on their email
101.        resSet.first(); //moves the current row of the result set to the first row
102.
103.        ResultSet orderSet = sC.searchOrders(resSet.getString("userID")); //contains the orders from userid
104.        orderSet.first(); //moves the current row of the result set to the first row
105.
106.        ResultSet rs = sC.searchAllProps();
107.        resultSetToArray(rs); //converts the result set to a linked list using the over ride function
108.        allPropList.addAll(0, record); //sets the allPropList to the value of record
109.
110.        resultSetToArray(sC.emailToOrders(email)); //converts the result set to a linked list using the over
ride function
111.        userOrdersList2.addAll(0, record); //sets the allPropList to the value of record
112.
113.        if (userOrdersList2.get(0).equals("null")) {
114.
115.            JOptionPane.showMessageDialog(null, "This user does not have any prior hires");
116.        }
117.
118.        propDetails1.clear(); //creates a list of characteristics by getting the value from the list of props
119.        propDetails1.add(userOrdersList2.get(4));
120.        propDetails1.add(userOrdersList2.get(5));
121.        propDetails1.add(userOrdersList2.get(6));
122.        propDetails1.add(userOrdersList2.get(7));
123.        propDetails1.add(userOrdersList2.get(8));
124.
125.        propDetails2.clear();
126.        propDetails2.add(userOrdersList2.get(13));
127.        propDetails2.add(userOrdersList2.get(14));
128.        propDetails2.add(userOrdersList2.get(15));
129.        propDetails2.add(userOrdersList2.get(16));
130.        propDetails2.add(userOrdersList2.get(17));
```

```

131.
132.         propDetails3.clear();
133.         propDetails3.add(userOrdersList2.get(22));
134.         propDetails3.add(userOrdersList2.get(23));
135.         propDetails3.add(userOrdersList2.get(24));
136.         propDetails3.add(userOrdersList2.get(25));
137.         propDetails3.add(userOrdersList2.get(26));
138.
139.
140.     } catch (SQLException ex) {
141.         resultsFound = false;
142.
143.         Logger.getLogger(recommendationPage.class.getName()).log(Level.SEVERE, null, ex);
144.     }
145.
146. }
147.
148. //used to compare the similarities of the users hires compared to other props
149. public LinkedList < Integer > compareValues(LinkedList < String > propDetails, LinkedList < Integer >
    similarScores) {
150.     for (int i = 0; i < propDetails.size(); i++) {
151.
152.         for (int j = 0; j < allPropList.size(); j++) { //loops based on the number of props in the database
153.
154.             if (allPropList.get(j).equals(propDetails.get(0))) {
155.                 similarScores.set(Math.floorDiv(j, 5) + 1, -100); //makes similarity negative if comparing the
    same prop
156.             }
157.
158.             if (allPropList.get(j).equals(propDetails.get(i))) { //if the prop has the same characteristic it
    gains a point
159.                 temporary++;
160.                 similarScores.set(Math.floorDiv(j, 5) + 1, similarScores.get(Math.floorDiv(j, 5) + 1) + 1);
161.             }
162.         }
163.     }
164.
165.     if (temporary < 1) {
166.         resultsFound = false;
167.     }
168.     return similarScores;
169.
170. }

```

Objective 3 Number of Props:

To retrieve the number of props currently in the warehouse and database an SQL function has to be applied which determines the amount.

Technique used: aggregate SQL functions through the use of COUNT()

```

1. //determines the number of props currently available in the database
2. int numberOfResults;
3. public int numberOfProps() {
4.
5.     try {
6.         Class.forName("com.mysql.cj.jdbc.Driver");

```



```
7.      Connection connection = DriverManager.getConnection("jdbc:mysql://localhost:3306/propdb", "root",
"rocket08");
8.      Statement statement = connection.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_READ_ONLY);
9.
10.     //returns the single value of how many props there are currently
11.     String customQuery = "SELECT COUNT(*) FROM props";
12.     statement.executeQuery(customQuery);
13.     rs = statement.executeQuery(customQuery);
14.
15.     rs.first();
16.     numberOfResults = rs.getInt(1); //sets the number of results to the value calculated
17. } catch (SQLException e) {
18.     System.out.println("SQL Exception: " + e.toString());
19. } catch (ClassNotFoundException cE) {
20.     System.out.println("Class Not Found Exception: " + cE.toString());
21. }
22. return numberOfResults;
23. }
```

Objective 3 List Functions:

To determine which props are the most similar, they are ordered using a merge sort which splits the array and reorders them. This is applied using the reverseList class. The items are then reversed to be ordered in descending order through the implementation of a stack.

Techniques used:

- Stack / queue operations (reverse)
- Merge sort of similarly efficient sort
- Recursion (merge sort)
- Linked list maintenance (a)

```
1. import java.util.*;
2.
3. public class reverseList {
4.
5.     //stores maximum count of elements stored in a stack
6.     int size;
7.     //stores index of top element of a stack
8.     int top;
9.     //stores address of array element
10.    LinkedList < String > a = new LinkedList < > ();
11.
12.    //function to check if the stack is empty or not
13.    boolean isEmpty() {
14.        return (top < 0);
15.    }
16.
17.    //function to create a stack of given capacity.
18.    reverseList(int n) {
19.        top = -1;
20.        size = n;
```

```
21.     for (int i = 0; i < size; i++) {
22.         a.add(" ");
23.     }
24.
25. }
26.
27. //pushes an element onto the stack
28. boolean push(String x) {
29.
30.     // If Stack is full
31.     if (top >= size) {
32.         System.out.println(
33.             "Stack Overflow");
34.         return false;
35.     } else {
36.
37.         // Insert element into stack
38.         a.set(++top, x);
39.         return true;
40.     }
41. }
42.
43. //function to remove an element from stack.
44. String pop() {
45.     //check if stack is empty
46.     if (top < 0) {
47.         System.out.println(
48.             "Stack Underflow");
49.         return null;
50.     }
51.
52.     //pop element from stack
53.     else {
54.         String x = a.get(top--);
55.         return x;
56.     }
57. }
58.
59. //takes the linked list and returns it reversed
60. public static LinkedList < String > reverse(LinkedList < String > arr, int n) {
61.     LinkedList < String > temp = new LinkedList < > ();
62.
63.     reverseList obj = new reverseList(n); //initialize a stack of capacity n
64.
65.     for (int i = 0; i < n; i++) {
66.
67.         //add arr to the stack
68.         obj.push(arr.get(i));
69.     }
70.
71.     //reverse the array elements
72.     for (int i = 0; i < n; i++) {
73.
74.         // Update arr[i]
75.         //arr[i] = obj.pop();
76.         arr.set(i, obj.pop());
77.     }
78.
79.     for (int i = 0; i < n; i++) {
80.         temp.add(arr.get(i));
81.     }
```

```
82.     return temp;
83. }
84.
85. //used to apply the sort
86. public static void mergeSort(LinkedList < String > list, int from, int to) {
87.     if (from == to) {
88.         return;
89.     }
90.     int mid = (from + to) / 2;
91.     // sort the first and the second half
92.     mergeSort(list, from, mid);
93.     mergeSort(list, mid + 1, to);
94.     merge(list, from, mid, to);
95. }
96.
97.
98.
99. public static void merge(LinkedList < String > list, int from, int mid, int to) {
100.     int n = to - from + 1; // size of the range to be merged
101.     String[] b = new String[n]; // merge both halves into a temporary array b
102.     int i1 = from; // next element to consider in the first range
103.     int i2 = mid + 1; // next element to consider in the second range
104.     int j = 0; // next open position in b
105.
106.     // as long as neither i1 nor i2 past the end, move the smaller into b
107.     while (i1 <= mid && i2 <= to) {
108.         if (list.get(i1).compareTo(list.get(i2)) < 0) {
109.             b[j] = list.get(i1);
110.             i1++;
111.         } else {
112.             b[j] = list.get(i2);
113.             i2++;
114.         }
115.         j++;
116.     }
117.
118.     //only one of the two while loops below is executed copy any remaining entries of the first half
119.     while (i1 <= mid) {
120.         b[j] = list.get(i1);
121.         i1++;
122.         j++;
123.     }
124.
125.     // copy any remaining entries of the second half
126.     while (i2 <= to) {
127.         b[j] = list.get(i2);
128.         i2++;
129.         j++;
130.     }
131.
132.     // copy back from the temporary array
133.     for (j = 0; j < n; j++) {
134.         list.set(from + j, b[j]);
135.     }
```

Objective 3 Retrieve Prop Details:

To retrieve the orders and prop details from the database, several SQL functions are applied

```
1. //selects all the orders from a specific user
2. public ResultSet searchOrders(String userID) {
3.
4.     try {
5.         Class.forName("com.mysql.cj.jdbc.Driver");
6.         Connection connection = DriverManager.getConnection("jdbc:mysql://localhost:3306/propdb", "root",
"rocket08");
7.         //Statement statement = connection.createStatement();
8.         Statement statement = connection.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_READ_ONLY);
9.
10.        String customQuery = "SELECT * FROM orders WHERE userID = '" + userID + "'";
11.        statement.executeQuery(customQuery);
12.        rs = statement.executeQuery(customQuery);
13.
14.    } catch (SQLException e) {
15.        System.out.println("SQL Exception: " + e.toString());
16.    } catch (ClassNotFoundException cE) {
17.        System.out.println("Class Not Found Exception: " + cE.toString());
18.    }
19.    return rs;
20. }
```

```
1. //returns the userID based on the email entered
2. public ResultSet emailToID(String email) {
3.
4.     try {
5.         Class.forName("com.mysql.cj.jdbc.Driver");
6.         Connection connection = DriverManager.getConnection("jdbc:mysql://localhost:3306/propdb", "root",
"rocket08");
7.         //Statement statement = connection.createStatement();
8.         Statement statement = connection.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_READ_ONLY);
9.        String customQuery = "SELECT userID, job FROM propdb.employees where email = '" + email + "'";
10.        rs = statement.executeQuery(customQuery);
11.
12.
13.    } catch (SQLException e) {
14.        System.out.println("SQL Exception: " + e.toString());
15.    } catch (ClassNotFoundException cE) {
16.        System.out.println("Class Not Found Exception: " + cE.toString());
17.    }
18.    return rs;
19. }
```

Agile Methodology for Objective 3:

Sprint 1:

Achieved:

During the initial stage of development, I was able to develop a recommendation system which would use the previous 5 hires from the user and show a page of similar props which they could then use to hire. The user would press the recommendation button which would open a separate window to enter their email, the page would then run the algorithm to generate a list of the 3 most similar props.

Feedback:

Once the initial stage of development was finished the client felt it was easy to use and intuitive but still had some areas which they felt could be improved upon. The first point they noticed was the use of a separate page which felt unnecessary since it made keeping track of information more difficult and led to the UX being more confusing. Another point was the fact that the user had to enter the email each time which felt unnecessary, it also could lead to issues with security if other users can enter emails which aren't their own. Finally, the use of 5 most recent hires could be changed to 3 in order to make the recommendations relevant to the current project instead of recommending based on props hired for previous sets.

Goals for the next sprint:

To remedy the use of another window I plan to use the same page which currently displays the searches, by doing this it keeps the user using a familiar interface. In addition, it allows the user to hire directly from the recommendations. The use of the email field can be changed by storing the email of the current user when they initially login which can be used while they are logged in which improves both the user experience and security.

Sprint 2:

Achieved:

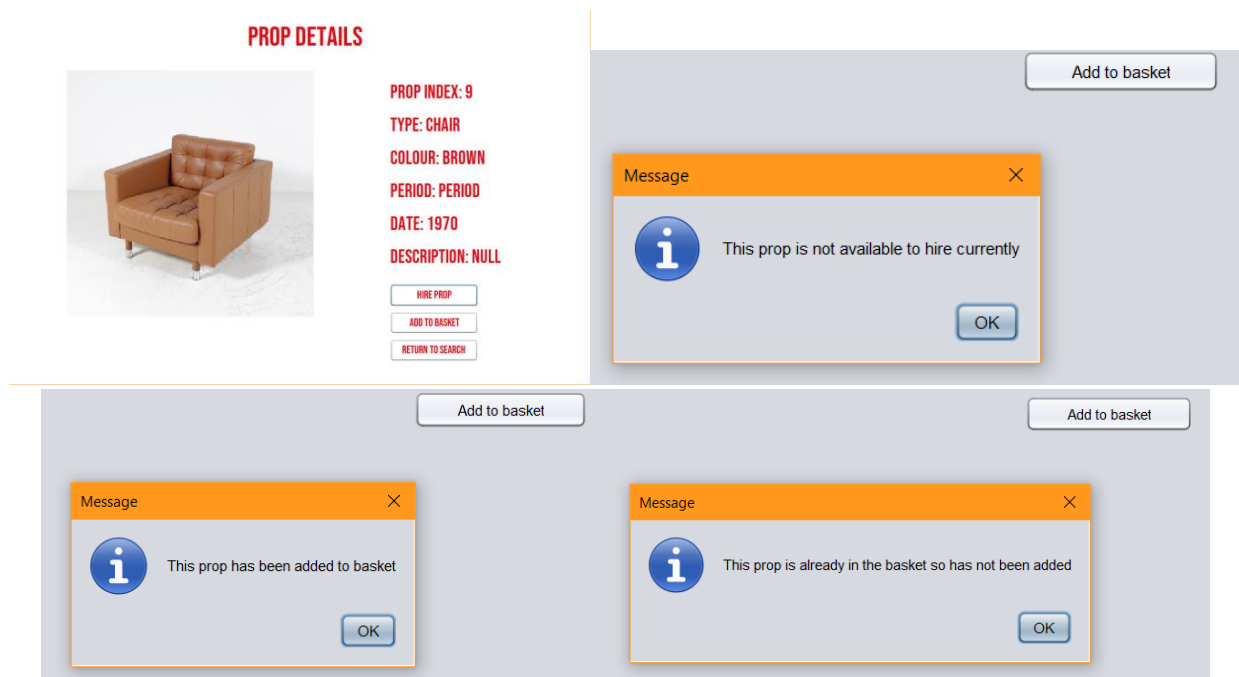
I was able to change where the recommendations were displayed so that the search function is used instead, this used the prop ID to display all the recommendations within the results table. In addition, the user no longer has to enter their email to see their recommendations, instead the program stores the email of the user logged in and uses that. Finally, the recommendations are instead based on the last 3 props instead of the last 5 to match the client's preferences.

Feedback:

After these changes were implemented, the client felt that the user interface was much clearer and easier to use because there were fewer inputs required. It also remained close to the interface used for searching and hiring so there was no need to relearn how the buttons worked. Finally, the client felt that the recommendations matched what they wanted when the algorithm used the 3 most recent hires instead of the last 5. Overall, they were very impressed by how easy and accurate the system was, comparing it to the Amazon feature.

Goals for the next sprint:

While the main client found the interface easy to use and clear, while testing with other stakeholders I found that they sometimes struggled to understand that the recommendations were being displaying in the results table. To remedy this I intend to add a pop up box that informs the user once the button has been pressed that the props shown in the table correspond to the most similar props. This communication with the user is important to make the work flow as clear as possible for new users.

Objective 4 basket system:

Objective 4 Checking Duplicates:

To manage the basket such as checking for duplicates the basket manager class is used. This check is applied by first using a tree sort and then a binary search.

Techniques used:

-Graph / tree traversal

-List operations

-Linked list maintenance (array list)

-Recursive algorithm

```
1. public class basketManager {
2.     ArrayList < Integer > basket = new ArrayList < Integer > (); //an array list of prop ID which will be sorted and
       searched
3.
4.     //declares the node object which will populate the binary search tree
5.     class Node {
6.         int key; //declares the key as an int
7.         Node left, right; //declares the left and right node to be generated
8.
9.         public Node(int item) {
10.             key = item; //sets the value of key to the item from parameter
11.             left = right = null; //null values to the left and right nodes
12.         }
13.     }
14.
15.     //the root node of the binary search tree
16.     Node root;
17.
18.     //constuctor for the class
19.     basketManager() {
20.         root = null;
21.     }
22.
23.     //insert a new value into the tree through insert rec
24.     void insert(int key) {
25.         root = insertRec(root, key);
26.     }
27.
28.     //A recursive function to insert a new key into binary search tree
29.     Node insertRec(Node root, int key) {
30.
31.         //check if the tree is empty and return a new node if so
32.         if (root == null) {
33.             root = new Node(key);
34.             return root; //returns the new root node
35.         }
36.
37.         //if the tree isnt empty then recursively loop down the tree
38.         if (key < root.key)
39.             root.left = insertRec(root.left, key); //insert left if less
40.         else if (key > root.key)
41.             root.right = insertRec(root.right, key); //insert right if more
42.
43.
44.         return root;
45.     }
```

```

46.
47. //traverse the binary tree in order based on the initial node
48. void inorderRec(Node root) {
49.     if (root != null) {
50.         inorderRec(root.left);
51.         loginPage LP = new loginPage();
52.         LP.tempBasketArray.add(root.key); //calls the function and adds the value to the array
53.         inorderRec(root.right);
54.
55.
56.     }
57. }
58.
59. //inserts new values into the tree based on the size of the array
60. void treeins(ArrayList < Integer > arr) {
61.     for (int i = 0; i < arr.size(); i++) {
62.         insert(arr.get(i)); //function used to create new node
63.     }
64.
65. }
66.
67. //used to apply a binary search to the now ordered list
68. //takes the sorted list , search value , start and end values
69. public int binarySearch(ArrayList < Integer > sortedArray, int value, int first, int last) {
70.     int middle = first + ((last - first) / 2); //determines the middle point of the array
71.
72.     if (last < first) {
73.         return -1; //return -1 if the values entered are not valid or search value not found
74.     }
75.
76.     if (value == sortedArray.get(middle)) {
77.         return middle; //return the middle index if the value is found
78.     } else if (value < sortedArray.get(middle)) {
79.         return binarySearch(sortedArray, value, first, middle - 1); //applies search again if not found
80.     } else {
81.         return binarySearch(sortedArray, value, middle + 1, last); //-1 or +1 if greater or less than middle
82.     }
83. }
84.
85. //applied each time an item is added to basket
86. public int addToBasket(ArrayList < Integer > arr, int searchValue) {
87.     loginPage LP = new loginPage();
88.     basketManager bM = new basketManager();
89.     bM.treeins(arr); //generates the binary tree
90.     bM.inorderRec(bM.root); //orders the list through an in order traversal
91.
92.     return binarySearch(LP.tempBasketArray, searchValue, 0, arr.size() - 1);
93.     //returns the position of the item in the array
94. }

```

Objective 4 Hire Basket Button:

When the hire basket button is pressed the program creates individual orders and changes the status of the props

```

1. //hires the props currently in the users basket
2. private void basketButtonActionPerformed(java.awt.event.ActionEvent evt) {

```



```
3.
4.     sqlConnection sC = new sqlConnection();
5.     String currentJob; //stores the users current production
6.     loginPage lP = new loginPage();
7.
8.     //prevents the user from using the hire function if there are no items in the basket
9.     if (lP.basketArray.isEmpty()) {
10.         JOptionPane.showMessageDialog(null, "The basket is empty");
11.     } else {
12.         try {
13.             ResultSet resSet = sC.emailToID(loginPage.emailGlobal); //stores the ID based on user email
14.             resSet.first(); //moves the result set to the first row
15.             currentJob = resSet.getString("job"); //stores the current production of the user
16.             String userID = resSet.getString("userID"); //stores their ID
17.
18.             String propID = Integer.toString(lP.basketArray.get(0)); //retrieves the propID and stores as string
19.             sC.addOrder(propID, userID, currentJob); //generates a new order using the information retrieved
20.
21.             //if the basket has more than one item in it the generate order process is repeated
22.             if (lP.basketArray.size() > 1) {
23.                 for (int i = 1; i < lP.basketArray.size(); i++) {
24.                     propID = Integer.toString(lP.basketArray.get(i));
25.                     sC.addOrder(propID, userID, currentJob);
26.                 }
27.             }
28.         } catch (SQLException ex) {
29.             Logger.getLogger(basketCheck.class.getName()).log(Level.SEVERE, null, ex);
30.         } catch (ParseException ex) {
31.             Logger.getLogger(basketCheck.class.getName()).log(Level.SEVERE, null, ex);
32.         }
33.         //informs the user once the hire process is complete
34.         JOptionPane.showMessageDialog(null, "The props in the basket have been hired");
35.     }
36. }
```

Objective 4 View and Edit Basket:

To view and remove the props from the basket the results table is used again with some modifications

```
1. private void viewBasketButtonActionPerformed(java.awt.event.ActionEvent evt) {
2.     loginPage lP = new loginPage();
3.     //checks the length of the basket array
4.     if (lP.basketArray.isEmpty()) {
5.         JOptionPane.showMessageDialog(null, "There are currently no items in your basket");
6.     } else {
7.         //converts from arraylist to string so that the values can be used in the search box
8.         String listString = lP.basketArray.stream().map(Object::toString).collect(Collectors.joining(", "));
9.
10.        searchBox.setText(listString); //sets the value of the search box to the recommendations IDs
11.        searchFunction(searchBox.getText()); //displays the recommended props with the search function
12.        JOptionPane.showMessageDialog(null, "The results table shows your current basket");
13.    }
14. }
15.
16. private void removeFromBasketButtonActionPerformed(java.awt.event.ActionEvent evt) {
17.     loginPage lP = new loginPage();
```

```
18. //checks if the user has selected a prop to remove from the basket
19. if (resultsTable.getSelectionModel().isSelectionEmpty() == true) {
20.     JOptionPane.showMessageDialog(null, "There is no prop selected");
21. } else {
22.     //stores the value which the user has selected in the results table
23.     String selection = (String) resultsTable.getValueAt(resultsTable.getSelectedRow(), 0);
24.
25.     //checks if the item selected is in the basket
26.     if (IP.basketArray.indexOf(Integer.parseInt(selection)) == -1) {
27.         JOptionPane.showMessageDialog(null, "This item cannot be found in your basket");
28.     } else {
29.         //if all inputs are correct the item is removed from the basket
30.         IP.basketArray.remove(IP.basketArray.indexOf(Integer.parseInt(selection)));
31.
32.         JOptionPane.showMessageDialog(null, "This prop has been removed");
33.     }
34. }
35. }
```

Agile Methodology for Objective 4:

Sprint 1

Achieved:

During the first stage of development for the basket I created a button and an array which would store the ID of props hired. These props would then be hired through the use of a button on the search page. The add to basket button would be found on the prop detail page and would alert the user when added.

Feedback:

While the basic functionality was created, the user felt that it was lacking some elements which are typically found when using other basket systems. For example, the ability to see and edit the items in the basket. By not having this the user felt that it was hard to keep track of which elements were added to the basket and meant that if they changed their mind, they couldn't edit the items they wanted to hire. To prevent this the user found themselves only adding a few props to the basket before hiring which defeats the point of the basket. In addition, since there was no way of tracking the items in the basket the client says that they found themselves adding duplicates to the basket. This could have the impact of duplicating hires on the same item.

Goals for the next sprint:

To remedy these issues, I decided I will add a button to the search page which can be used to display the items in the basket through the results table. This will allow the user to check the items which they have already hired. I also plan to use the selected row function to allow the user to select a row which will then remove the selected item from the basket. Finally, while the previous changes should reduce the number of duplicates, I will still implement a check which will determine if the item being added will be a duplicate. This will hopefully reduce the potential for errors and bugs.

Sprint 2

Achieved:

Based on the objectives of the previous sprint I was able to work on improving the quality of the basket for the client and stakeholders. The first change which was made was the ability to view the basket through the use of a button. This was implemented with the same system as found in the recommendations by entering the prop IDs and applying a search based on those. Next the ability to remove items was added which functions in a similar way to the prop detail page except it gets the prop ID and removes it from the basket array. Finally, through the use of a tree sort and binary search I was able to check the presence of an item before adding another item to the basket.

Feedback:

The client was widely impressed by how much the user experience improved when compared to the first sprint. The ability to view and remove the items from the basket made it much easier to manage the items they wanted to hire and reduced the amount of guessing they had to make. By allowing the basket system to work better, the user found that they switched from using the normal hire function to instead preferring the basket workflow. Finally, they rarely noticed the duplicate item message but when they did, it was clear and could understand why the item wasn't added which made the process clear.

Objective 5 Prop Management:

GET PHOTO FILE PATH THIS IS THE FILE PATH: NONE

PROP TYPE:

PROP COLOUR:

PROP PERIOD:

PROP ERA:

PROP DESCRIPTION:

Message

There is no prop image selected

OK

SUBMIT PROP



TITLE: CHAIR

COLOUR: RED

PERIOD: VICTORIAN

Message

This prop has been hired by user with email: ali.harvey@me.com

OK

HIRE PROP

ADD TO BASKET

RETURN TO SEARCH

COLOUR: ORANGE

PERIOD: PERIOD

Message

This prop is not available to hire currently

OK

HIRE PROP

ADD TO BASKET

RETURN TO SEARCH

Objective 5 Hire Prop Button:

The hire button has a function attached which hires the prop instantly based on the current prop detail page.

```

1.  b.addActionListener(new ActionListener() {
2.      public void actionPerformed(ActionEvent e) { //executed when the hire button is pressed
3.          //checks the validity of the string entered by the user
4.
5.          //confirms with the user
6.          int dialogButton = JOptionPane.YES_NO_OPTION;
7.          int dialogResult = JOptionPane.showConfirmDialog(null, "Are you sure you want to hire this prop", "Warning",
            dialogButton);
8.
9.          if (dialogResult == JOptionPane.YES_OPTION) {} else {
10.             JOptionPane.showMessageDialog(null, "Prop has not been hired");
11.         }
12.         sqlConnection sC = new sqlConnection();
13.         loginPage lP = new loginPage();
14.         ResultSet resSet = sC.emailToID(loginPage.emailGlobal); //stores the id of the user
15.         try {
16.             if (propAvailable.equals("1")) { //checks if the prop is available to hire
17.                 resSet.first(); //sets the current row to the first row
18.                 String currentJob = resSet.getString("job"); //retrieves the current job
19.                 String userID = resSet.getString("userID"); //retrieves the user id
20.                 sC.addOrder(propID, userID, currentJob); //generates an order
21.                 JOptionPane.showMessageDialog(null, "This prop has been hired by user with email: " +
                    lP.emailGlobal);
22.             } else {
23.                 JOptionPane.showMessageDialog(null, "This prop is not available to hire currently");
24.             }
25.
26.         } catch (SQLException ex) {
27.             Logger.getLogger(detailManager.class.getName()).log(Level.SEVERE, null, ex);
28.         } catch (ParseException ex) {
29.             Logger.getLogger(detailManager.class.getName()).log(Level.SEVERE, null, ex);
30.         }
31.     }
32. });

```

Objective 5 Returning Props:

When returning or entering a new prop a separate page is used

```

1.  import java.awt.Color;
2.  import javax.swing.JFileChooser;
3.  import javax.swing.JOptionPane;
4.
5.
6.  public class addProp extends javax.swing.JFrame {
7.
8.      /**
9.       * Creates new form addProp
10.      */
11.      public addProp() {
12.          getContentPane().setBackground(Color.WHITE);
13.          this.setExtendedState(this.MAXIMIZED_BOTH);
14.          initComponents();

```

```

15.         setDefaultCloseOperation(this.DISPOSE_ON_CLOSE);
16.     }
17.
18.     //run when the get photo button is pressed
19.     private void getPhotoButtonActionPerformed(java.awt.event.ActionEvent evt) {
20.         //creates a file chooser button
21.         JFileChooser file = new JFileChooser("D:\\Homework A Levels\\Computing\\NEA\\projectImages");
22.
23.         file.setMultiSelectionEnabled(false); //allows the user to only select one file
24.
25.         //allows both files and directories to be displayed
26.         file.setSelectionMode(JFileChooser.FILES_AND_DIRECTORIES);
27.
28.         file.setFileHidingEnabled(false); //shows file
29.         if (file.showOpenDialog(null) == JFileChooser.APPROVE_OPTION) {
30.             //gets the file path of the selected file from the file chooser
31.             java.io.File f = file.getSelectedFile();
32.
33.             filePathLabel.setText(f.getPath()); //sets the label to the path
34.         }
35.
36.     }
37.
38.     private void submitPropButtonActionPerformed(java.awt.event.ActionEvent evt) {
39.         String imagePath = filePathLabel.getText(); //gets the file path from the value which is found in the
        label
40.         if (imagePath.equals("This is the file path: none")) { //determines whether a file has been selected
41.             JOptionPane.showMessageDialog(null, "There is no prop image selected");
42.         } else {
43.             imagePath = imagePath.replace("\\", "\\"); //adds the escape characters to allow the file path to
        function when stored
44.             sqlConnection sC = new sqlConnection();
45.             sC.addProp((String) typeCombo.getSelectedItem(), (String) colourCombo.getSelectedItem(), (String)
        periodCombo.getSelectedItem(), (String) eraCombo.getSelectedItem(), descriptionField.getText(), imagePath);
46.             //creates a new prop using the characteristics entered
47.         }
48.     }
49.
50.     private void returnHiredActionPerformed(java.awt.event.ActionEvent evt) {
51.         // TODO add your handling code here:
52.         UIManager uiMan = new UIManager();
53.         uiMan.addPropToExisting();
54.     }

```

Agile methodology for objective 5

Sprint 1

Achieved:

During the initial sprint I was able to generate prop hire system which functioned through pressing the hire button on the prop detail page. This would lead to a message informing the user, setting the prop to unavailable and generating an order which documents the hire of the item. In addition, a prop add page was created which would allow those managing the inventory to add new props to the database through entering values into the fields and uploading images.

Feedback:

The client felt that a majority of the functionality was met through the initial stage however there were a few changes which they want to make. One thing which they noticed was with the adding of new props which did not check whether the user had added an image, this meant that a prop could be added without including an image. The client felt that this would be an issue since all items would need images.

Goals for the next sprint:

Since the client felt that only a small change needs to be made only the addition of a check on the use of an image will need to be added. This will ensure that when submitting a prop, the image section has been set to a valid file path

Sprint 2:**Achieved:**

The implementation of a check was straightforward with a comparison to the default value being made. If the check was found to fail, then a message would be shown to the user and the prop would not be submitted which prevents issues in generating prop details.

Feedback:

After this small change the client was very pleased with the prop management system due to all objectives being met while being easy to use.

Testing:

Search Function:

Test type: Valid input

Data type: Normal data

Input value: "red"



Pass/Fail: Pass

Reason for test: to see if entering a characteristic returns the appropriate props

Expected outcome: The expected values which the user should be entering are characteristics of props such as year, period and colour. When one of these keywords are entered in the search box all the props with that characteristic should be returned.

Actual outcome: This test displayed all props which had the characteristic of red which makes the test successful

SEARCH

Prop ID	Prop Type	Prop Colour	Prop Image
2	Chair	Red	
5	Light	Red	

Search Function:

Test type: Invalid

Data type: Erroneous data

Input value: "hotel"

Pass/Fail: Fail

Reason for test: To determine if the search function handles an incorrect input by displaying a message to the user

Expected outcome: When using the search function, the user will typically be entering characteristics of props, however if the user decides to enter an incorrect query the program should inform them which is the expected result.

Actual outcome: Prior to testing when the user input an incorrect search term the results table would be blank, the only form of feedback would be found in the error code in the output terminal.

Fix: By checking the length of the results list we are able to display an error message and stop the rest of the program from executing.

Prior to the fix (actual outcome):

Prop ID	Prop Type	Prop Colour	Prop Image

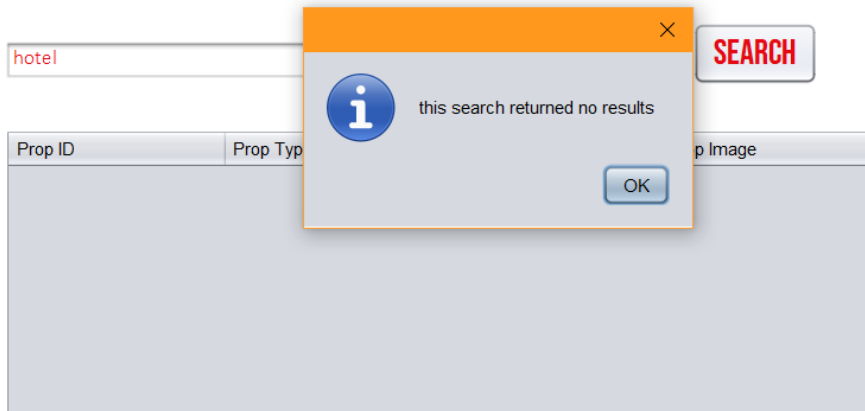
The code used to fix:

```

289         tm.addColumn("Prop Image");
290
291         //checks if there were no results to return error message
292         if(result1.isEmpty()){
293             JOptionPane.showMessageDialog(null, "this search returned no results");
294         }
295         else{
296
297         //block used to generate the table model which fill the results table
298         for (int i = 0; i <= result1.size()-1; i++) { //loops based on the number of results

```

Message displayed after fix:



Search function:

Test type: Invalid

Data type: Erroneous data

Input value: "red table"

Pass/Fail: Fail


Reason for test: To see if the results table shows all props which match the characteristics when multiple terms are entered in the search box

Expected outcome: When the user attempts to enter multiple search terms the program should consider each of the values and return the appropriate search results.

Actual outcome: Prior to testing the program would only consider the first term while not considering the other words.

Fix: To fix this the search function is repeated for each word in the search box. This means that the results returned are a combination of the words provided by the user.

Prior to fix (actual outcome):

Prop ID	Prop Type	Prop Colour	Prop Image
2	Chair	Red	

Code used to fix:



It takes the search terms as a list in the words array. Each word is extracted through the use of `.split(" ")` which uses the space between words to separate. The SQL statement found in `customQuery` is added to with the value of each word. This allows the search to return the props which match the characteristics provided.

```
if (prop.contains(" ")) {
    String[] words = prop.split(" ");

    try
    {
        Class.forName("com.mysql.cj.jdbc.Driver");
        Connection connection = DriverManager.getConnection("jdbc:mysql://localhost:3306/propdb","root","rocket08");
        //Statement statement = connection.createStatement();
        Statement statement = connection.createStatement (ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_READ_ONLY);
        String customQuery = "SELECT * FROM props WHERE" + words[0] + "IN(propID, propType, colour, period, propDate, propDescription)";
        for (int i = 0; i < words.length; i++) {
            customQuery = customQuery.concat("OR" + words[i] + "IN(propID, propType, colour, period, propDate, propDescription)");
        }

        System.out.println(" this is the custom query being run " + customQuery);
        rs = statement.executeQuery(customQuery);
    }
    catch (Exception e) {
        e.printStackTrace();
    }
}
```

Search results after fix:

Prop ID	Prop Type	Prop Colour	Prop Image
1	Chair	Orange	
2	Chair	Red	

Login Page:

Test type: Valid

Data type: Normal data

Input value: username: "admin" password: "pass"

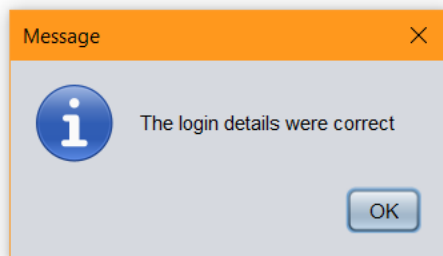
Pass/Fail: pass

Reason for test: To determine if the login system works as expected when correct login details are entered

Expected outcome: When the user logs in using their correct email and password the expected outcome is to alert them that the login was successful through a pop up message and transition the current window to the search page.

Actual outcome: This test displayed the correct message and changed the window to the search page.

Fix: This was a successful test so no changes had to be made.



USERNAME / EMAIL:

admin

PASSWORD:

pass

LOGIN

CREATE ACCOUNT

Login Page:

Test type: Valid

Data type: Boundary data

Input value: (incorrect login details) username: "admin" password: "pas"

Pass/Fail: Fail

Reason for test: To determine if the login system would alert the user to the use of incorrect login details

Expected outcome: The expected outcome was providing the user with an error message which would allow them to remain on the login page and enter correct details

Actual outcome: When incorrect details were entered previously the page would not respond, instead it would just remain on the login page. This could leave the user confused as to whether the program was functioning or not.

Fix: To remedy this issue an error message is displayed when the user enters the wrong login details and the user is not logged in.

Prior to fix (actual outcome):

USERNAME / EMAIL:

admin

PASSWORD:

pas

LOGIN

CREATE ACCOUNT

Code used to fix:

This section of code is used to compare the hashed value of the password which the user entered to the hashed value which is found in the database. If the password is equal to the value found in the database then the appropriate message is displayed, the same is applied to if the password is incorrect. By applying the hash function to the password it improves the security of the password storage system.

```
while(resultSet.next()){
    if(password.equals(resultSet.getString("password"))){
        JOptionPane.showMessageDialog(null, "The login details were correct");
        login = true;
    }
    else{
        JOptionPane.showMessageDialog(null, "The login details were incorrect");
        login = false;
    }
}
```

Account Creation:

Test type: Valid

Data type: Normal data

Input value: Appropriate login details such as a valid email address and phone number of correct length

Pass/Fail: Pass

Reason for test: To determine if the program would allow the account to be created when all the appropriate login details were entered

Expected outcome: When all the valid information is entered to create the account then the user should be informed that it has been created and their details should be added to the database.

Actual outcome: The user receives a pop up to inform them and the window view switches to the login page so they can use the credentials entered. This test was successful so no changes had to be made

ACCOUNT CREATION:

FIRST NAME

SURNAME

EMAIL

PHONE

JOB ROLE

CURRENT LOCATION

ACCOUNT PASSWORD

Message Account Created Successfully

10	James	Miner	jmig@gmail.com	07493515110	Kunner	The Crown	48112384006046a200d97e55919d0d
11	Rafa	Davison	rafad@outlook.com	07779275979	PropMan	Hellboy	c269ea2f77e5e9f38e2eb7237375eae

Account creation:

Test type: Invalid

Data type: Erroneous

Input value: Invalid account details such as those which do not meet the requirements of containing an "@" and a correct length of phone number

Pass/Fail: Fail

Reason for test: To ensure that the database has accurate values when storing the details of users. This test aims to increase the likelihood of accurate employee details

Expected outcome: The user is informed that the login details are incorrect and the characteristics which have to be met. Once the popup is closed the window should remain on the login page with all the details still filled in so only the incorrect details have to be remedied.

Actual outcome: Previously the user would be able to create an account using any details such as an email not containing "@". This would create errors in the database if there were inaccurate values entered by the user.

Fix: To remedy this, there is now a check on the values entered to see if they are of the correct length and contain appropriate values with the account creation process not proceeding without these features being met.

Prior to fix:

ACCOUNT CREATION:

FIRST NAME	<input type="text" value="Matthew"/>
SURNAME	<input type="text" value="Cook"/>
EMAIL	<input type="text" value="hotel"/>
PHONE NUMBER	<input type="text" value="123"/>
JOB ROLE LABEL	<input type="text" value="Runner"/>
CURRENT PRODUCTION	<input type="text" value="The Crown"/>
ACCOUNT PASSWORD	<input type="text" value="password"/>

CREATE ACCOUNT

Code used to fix:

Checking the presence of capitals was implemented using a separate function called capitalCheck. It functions by iterating the number of characters in the password and checking each value to determine if the string contains both a capital and a lower case. Through the use of flags, it allows the program to return true if there is a capital and lowercase.

```

private static boolean capitalCheck(String str) {
    char ch;
    boolean capitalFlag = false;
    boolean lowerCaseFlag = false;
    boolean numberFlag = false;
    for(int i=0; i < str.length(); i++) {
        ch = str.charAt(i);
        if (Character.isDigit(ch)) {
            numberFlag = true;
        }
        else if (Character.isUpperCase(ch)) {
            capitalFlag = true;
        }
        else if (Character.isLowerCase(ch)) {
            lowerCaseFlag = true;
        }
        if(numberFlag && capitalFlag && lowerCaseFlag)
            return true;
    }
    return false;
}

```

Checking for length and @:

This if statement first checks that the length of the phone number is 11 which is standard in the uk. It then uses a regex expression to determine whether the string contains a number using the \d character check. It finally uses the capitalCheck function to determine the presence of both a capital and lower case. If these are all found to be true then a message is displayed and the account is created. Otherwise a message is displayed to the user outlining the requirements for the account details.

```

if(matchFound == true && phoneNumberField.getText().length() == 11 && passwordField.getText().matches(".*\\d.*") == true && capitalCheck(passwordField.getText())){
    sC.accountCreation(firstNameField.getText(), surnameField.getText(), emailField.getText(), phoneNumberField.getText(), (String)jobRoleCombo.getSelectedText());
    JOptionPane.showMessageDialog(null, "Account Created Successfully");
    Ulman.createAccountToLogin();
}
else{
    JOptionPane.showMessageDialog(null, "1. Email must contain @ \n 2. Phone number of 11 characters \n 3. Number and capital in password");
}

```

After fix:

FIRST NAME

SURNAME

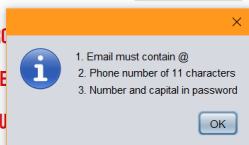
EMAIL

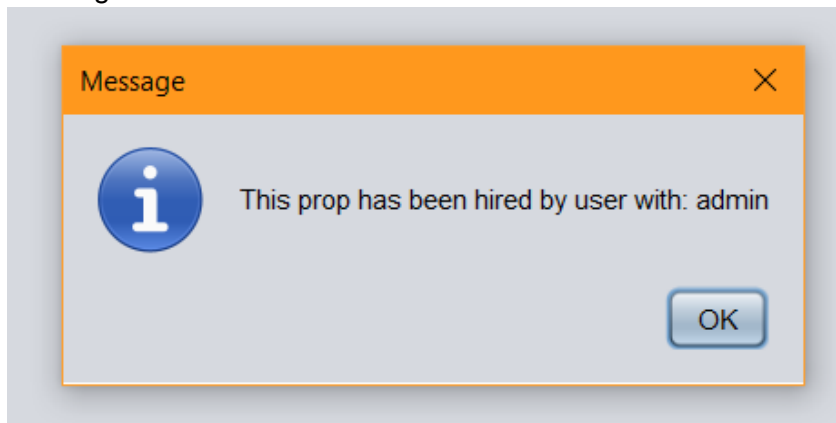
PHONE NUMBER

JOB ROLE

CURRENT ACCOUNT

CREATE ACCOUNT



Hire Function:**Test type:** Valid**Data type:** Normal data**Input value:** Hire button pressed**Pass/Fail:** Pass**Reason for test:** To ensure that when all characteristics are correct there are no errors**Expected outcome:** When an available prop has been selected to hire by a valid account the user should be alerted, the prop should be made unavailable to hire by anyone else and an order should be created.**Actual outcome:** The program functioned as expected so the message was displayed to the user and the appropriate changes were made in the database.**Message to user:****Order created in the orders table:**

22	8	2	null
----	---	---	------

The prop details have been changed to make it not available:

8	Light	Black	Georgian	1030	NULL	U:\homework A Levels \Computing \VCA \project\images \#.jpg	1
---	-------	-------	----------	------	------	---	---

Hire function:

Test type: Invalid

Data type: Normal data

Input value: Hire button pressed on an unavailable prop

Pass/Fail: Fail

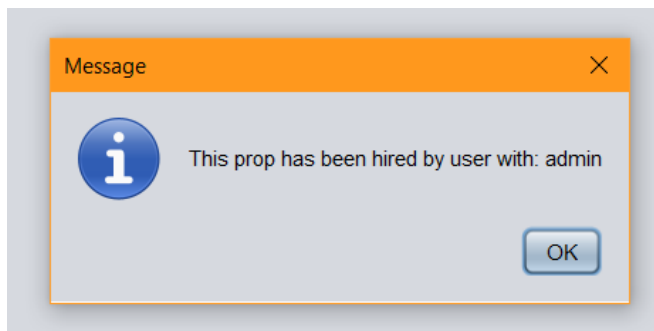
Reason for test: If the check is not made then an unavailable prop may be hired.

Expected outcome: Once a prop has been selected and the hire button has been pressed a check should be made to see if the prop is available to hire. An error message should be displayed to the user to show that the prop is not available to hire.

Actual outcome: Since the check is not performed on the availability column it means that the hire process continues as normal and an order is created.

Fix: By implementing a check once the button is pressed it prevents the process from continuing based on the value of the availability column in the prop database.

Actual outcome:



Code used

Prop available is a variable which stores the value of column 8 from the prop table. This will be 1 if available and 0 if not, this check ensures that the prop is free before proceeding with the hire. The hire is completed using add order which generates a row using the prop id, user id and users current job. When the prop is not available, an error message is displayed to the user.

```
try {
    if (propAvailable.equals("1")){ //checks if the prop is available to hire
        resSet.first();
        String currentJob = resSet.getString("job");
        String userID = resSet.getString("userID");
        sC.addOrder(propID, userID, currentJob);
        JOptionPane.showMessageDialog(null, "This prop has been hired by user with: ".IP.emailGlobal);
    }
    else{
        JOptionPane.showMessageDialog(null, "This prop is not available to hire currently");
    }
}
```

Add to basket function:

Test type: Valid

Data type: Normal data

Input value: Add to basket button press

Pass/Fail: Fail

Reason for test: To test if the basket function will generate duplicates in hires and the hiring of unavailable items

Expected outcome: When adding a prop to the users basket the program should check if the prop is available to hire and if the prop is not already in the users basket.

Actual outcome: The program would add the prop to the basket regardless of availability and existing basket.

Fix: The checking of whether the basket contains the prop already is implemented using the basket manager class. The basket manager class generates a binary tree of the prop IDs and checks the position of the new prop in the table. If the prop is already in the tree then the position is returned, otherwise -1 is returned. When -1 is returned the program knows that the basket does not already contain this prop so the new item can be added.

Code fix:

```
public int addToBasket(ArrayList<Integer> arr , int searchValue){
    loginPage IP = new loginPage();
    basketManager bM = new basketManager();
    //int arr[] = {5, 4, 7, 2, 11};
    bM.treeins(arr );
    bM.inorderRec(bM.root);

    //System.out.println("this is the final basekt array: "+IP.tempBasketArray);
    System.out.println("this is the position of the value " + binarySearch(IP.tempBasketArray , searchValue , 0 , IP.tem
    return binarySearch(IP.tempBasketArray , searchValue , 0 , arr.size()-1);
}
```

Prop Detail Page:

Test type: Valid

Data type: Normal input

Input value: Detail button pressed

Pass/Fail: Pass

Reason for test: To ensure that when all the correct characteristics are entered that the detail function works as intended

Expected outcome: When a prop is selected the prop details page can then be pressed to see further information about the prop, this page also allows the prop to be hired or added to the basket.

Actual outcome: This functioned as expected since the prop was selected which allowed the information to be shown, no fixes had to be made.


LOG OUT


RECOMMENDATIONS

PROP DETAILS PAGE

1

SEARCH

Prop ID	Prop Type	Prop Colour	Prop Image
1	Chair	Orange	



PROP INDEX: 1

TYPE: CHAIR

COLOUR: ORANGE

PERIOD: PERIOD

DATE: 1970

DESCRIPTION: NULL

HIRE PROP

ADD TO BASKET

RETURN TO SEARCH

Prop Detail Page:

Test type: Valid

Data type: Normal data

Input value: Prop details button pressed

Pass/Fail: Fail

Reason for test: To make sure the user experience is clear when interacting with the search function and details button

Expected outcome: When the prop details button is pressed but there is no prop selected the expected outcome is that the user is informed through a pop up.

Actual outcome: However this test failed with the program instead not reacting which could leave the user confused as to why details are not being displayed.

Fix: Using a check on the selection status of the results table allows the program to determine if a row has been selected or not

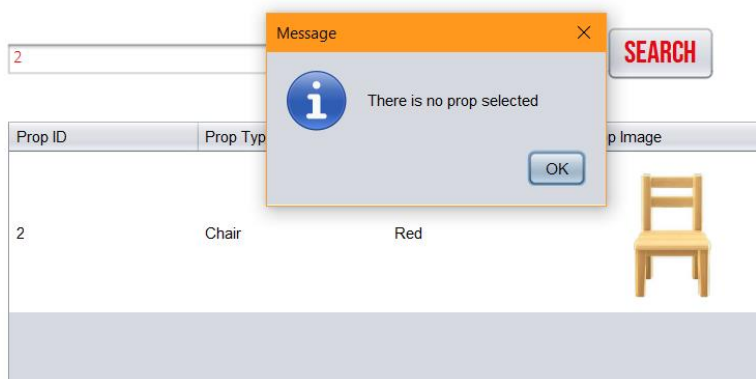
Code:

To fix this issue when the button is pressed, a check is first made to see if the tables selection is empty using the get selection model. If there is no selection made then the message is displayed. When there is a selection made then the else block is run which allows the rest of the function to call.

```
private void detailButtonActionPerformed(java.awt.event.ActionEvent evt) {  
    if(resultsTable.getSelectionModel().isSelectionEmpty()){  
        JOptionPane.showMessageDialog(null, "There is no prop selected");  
    }  
    else{  
        |  
        sqlConnection sC = new sqlConnection();  
    }  
}
```

After fix:

The error message is shown clearly to the user which allows them to understand why the details button is not displaying further information. It also prevents the code from throwing an exception since the invalid code is never run.



Recommendation System:

Test type: Valid

Data type: Normal data

Input value: Recommendation button pressed

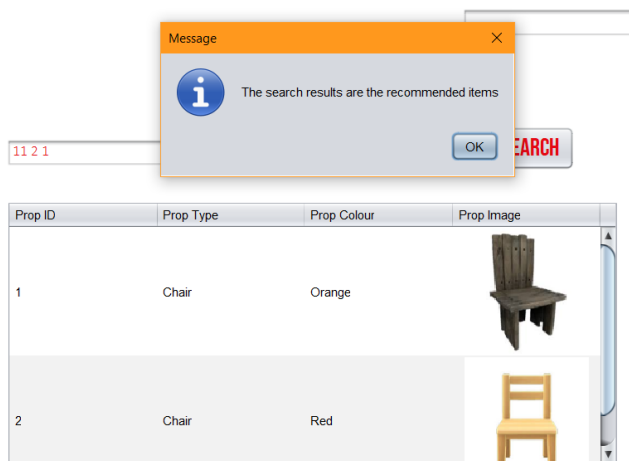
Pass/Fail: Pass

Reason for test: To determine whether the recommendation system is working as expected when all the variables are correct

Expected outcome: When an account with previous hires uses the recommendation button the user should be shown a message and the recommendations are displayed in the results table.

These props have been determined by the similarity of the users previous hires

Actual outcome: This outcome was successful so no changes had to be made.



Recommendation System:

Test type: Invalid

Data type: Boundary data

Input value: User pressing recommendation button without a valid history of hires

Pass/Fail: Fail

Reason for test: To ensure that the props recommended are accurate instead of recommending props based on only a single previous hire

Expected outcome: When there is no valid hire history for the current user there should be an alert to explain why there are no results

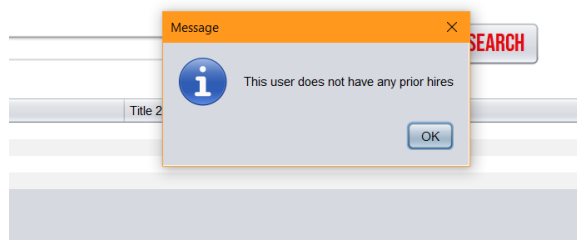
Actual outcome: This test failed with the page not responding when the button was pressed.

Fix: By determining the number of orders the user has it can display the appropriate message.

Code:

The code used to display the error message checks the size of the linked list which stores the previous order information. If value is null then there are no previous orders.

```
if(userOrdersList2.get(0).equals("null")){  
    System.out.println("this is empty !!!");  
    JOptionPane.showMessageDialog(null, "This user does not have any prior hires");  
}
```



Basket System:

Test type: Valid

Data type: Normal data

Input value: Add to basket button pressed

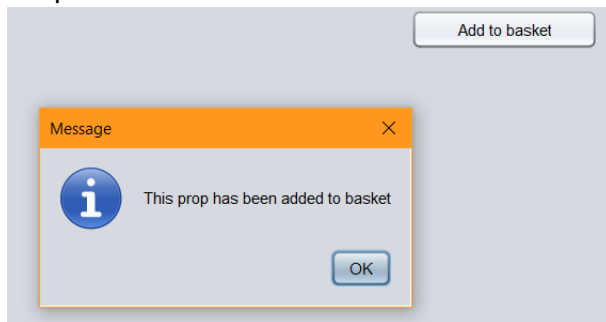
Pass/Fail: Passed

Reason for test: To ensure that the basket function works as expected when all characteristics are correct

Expected outcome: When the user attempts to add an available prop to their basket which has not already been added the program should display a message to the user and their basket should be updated.

Actual outcome: The popop displays the information to the user, the basket array is updated.

Output:



Basket System:

Test type: Valid

Data type: Erroneous data

Input value: Add to basket button pressed

Pass/Fail: Fail

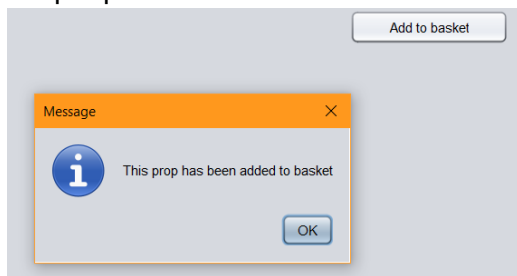
Reason for test: To stop invalid items from being added to the basket and hired

Expected outcome: When the user attempts to hire a prop which is not available to hire the user should be informed that the prop cannot currently be hired.

Actual outcome: The program would add the prop to the basket despite it not being available.

Fix: By applying the same check as made when hiring a prop it prevents the invalid items from being added

Output prior to fix:



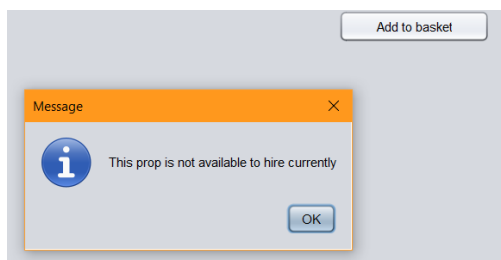
Code:

To ensure that the prop is available the column titled propAvailable is checked to determine the value. If the value is 1 then the prop is available, otherwise the prop cannot be hired and is not added to the basket.

```
if (propAvailable.equals("1")){
    if(bM.addToBasket(IP.basketArray, templnt) == -1){ //used as a check to see if the item has already been a
                                                        //returns -1 if the item is not found
        JOptionPane.showMessageDialog(null, "This prop has been added to basket");
        IP.basketArray.add(templnt);
    }

    else{
        JOptionPane.showMessageDialog(null, "This prop is already in the basket so has not been added");
    }
}
else{JOptionPane.showMessageDialog(null, "This prop is not available to hire currently");}
```

After fix:



Basket System:

Test type: Invalid

Data type: Erroneous data

Input value: Adding a prop which is already present to the basket

Pass/Fail: Fail

Reason for test: To check whether adding duplicate props to the basket will be detected

Expected outcome: When the prop is being added to the basket the program adds the value to the basket it should first check to see if the item has already been added to the basket. If the prop is already in the basket then the item should not be added and the user should be informed through a pop up message.

Actual outcome: This test failed with the item instead automatically being added to the basket.

Fix: Sorting and searching the existing list to first determine if the item is already present

Code:

To check whether the item has already been added, a binary tree sort is first performed using the functions in the basket manager class. This leads to a list which is ordered and allows a binary search to be applied. The binary search function then returns the position of the item in the list, and returns -1 if the item is not found. This process is applied each time an item is added to the basket with the item only being added when it is not already present.

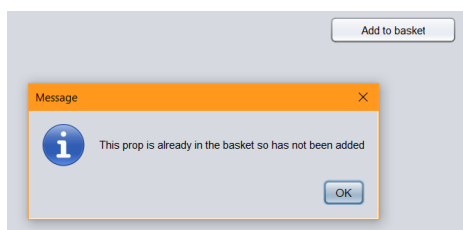
```
public int addToBasket(ArrayList<Integer> arr , int searchValue){
    loginPage IP = new loginPage();
    basketManager bM = new basketManager();
    //int arr[] = {5, 4, 7, 2, 11};
    bM.treeins(arr);
    bM.inorderRec(bM.root);

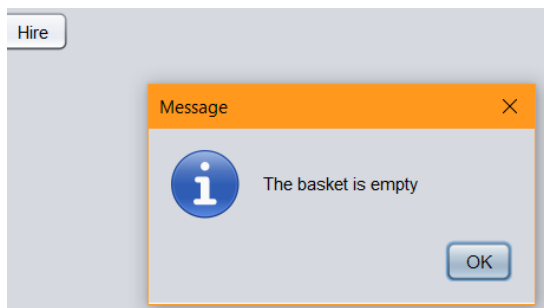
    //System.out.println("this is the final basekt array: "+IP.tempBasketArray);
    System.out.println("this is the position of the value " + binarySearch(IP.tempBasketArray , searchValue , 0 , IP.tempBasketArray.size()-1));
    return binarySearch(IP.tempBasketArray , searchValue , 0 , arr.size()-1);
}

//used as a check to see if the item has already been added
//returns -1 if the item is not found
if(bM.addToBasket(IP.basketArray , tempInt) == -1){
```

Result after fix:

This is displayed to the user if the item is already found in the list.



Basket System:**Test type:** Valid**Data type:** Normal data**Input value:** Hire button pressed**Pass/Fail:** Pass**Reason for test:** To ensure that the collective hiring of items through the use of the basket is functioning as expected with the correct number of orders being executed.**Expected outcome:** Once the basket has been created and there are items to hire, the basket button should be used to automatically hire all these items. This includes changing the status of these props to 0 instead of 1 and creating a record of the hire for each prop. However if there are no props in the basket the user should be informed and the process should stop.**Actual outcome:** This test was successful with the check of the basket size upon the button press allowing the code to show an error message to the user

Add Prop Function:

Test type: Invalid

Data type: Erroneous data

Input value: Image file not selected

Pass/Fail: Fail

Reason for test: To ensure a level of quality in the submissions of new props to the database

Expected outcome: When there is no image path selected for the prop photo the prop submission form should not complete. While the other sections of the form are optional, by ensuring an image is submitted it allows the other information to be inferred by the user. The expected outcome is that the user is informed and the process stops.

Actual outcome: The form was still submitting regardless of whether there was a file selected which meant that there were props in the database which did not have an image associated with them.

Fix: Determines whether the default file path has changed or not

Code:

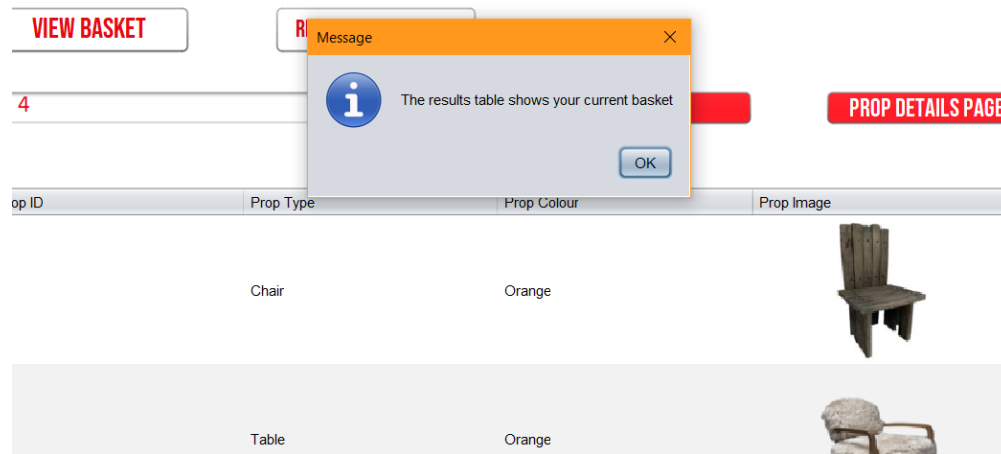
This is used to check the value of the file path, if the value is the same as the default then the message is displayed. If the file path has been altered then the program proceeds.

```
// TODO add your handling code here.  
String imagePath = filePathLabel.getText();  
if (imagePath.equals("This is the file path: none")) {  
    JOptionPane.showMessageDialog(null, "There is no prop image selected");  
}  
else {
```

Outcome after fix:

This now displays when the user attempts to submit a prop into the database without the submission of a valid image file path.



Basket view function:**Test type:** Valid**Data type:** Normal data**Input value:** View basket button pressed**Pass/Fail:** Pass**Reason for test:** To test whether the view basket button correctly displays all the items in the basket when the user has created a basket**Expected outcome:** The results table shows the items which the user has added to their basket while they have been logged**Actual outcome:** The items displayed accurately with the use of the item IDs which are stored in the basket applied to the search function**Output:**

Evaluation:

Evaluation Table:

Success Criteria ID	Met (True / False)	Reflection
Objective 1	True	Overall, the client is happy with the login system due to the ease of use and the communication with the user which makes the system clear. All features have been added and the ability to create a new account is an efficient process.
Objective 1.1	True	By creating a MD5 hash algorithm it allows the user to enter their password which will be securely stored in the database. It also can be compared when logging in by applying the hash function each time.
Objective 1.2	True	The redirection of users was not necessary, instead a simple pop up was able to inform the user when an incorrect login is entered. It also prevents the system from progressing without entering a valid login.
Objective 1.3	True	The "create account" button displays the user a new page which allows the information to be entered through a combination of text boxes and drop-down menus. It stores the values in the database using the employee table.
Objective 1.4	True	The check of the data entered is applied to the phone number, email and password. In addition, the use of a complexity check on the password generates improved security for the whole system. It also prompts the user when invalid values are entered which makes the user experience clearer.
Objective 1.5	True	A clear navigation for the login and account generation was achieved through the use of buttons and smaller layered windows. This is supported by the feedback received from the user who found it easy and straightforward to use.
Objective 2	True	The ability to search and find props was a main problem with the previous system. This solution made the searching of available props and items easier. The user found the method of searching intuitive and clear.
Objective 2.1	True	The system for searching matches to words found in the search box. It applies a for loop for the number of words found which means that the search is applied for each term.

Objective 2.2	True	The search system determines the number of results and displays a message if the number of results is 0. This allows users to understand why there are no results, it also allows unexpected inputs to be dealt with without generating errors.
Objective 2.3	True	When a prop in the results table is selected it allows the details button to be pressed which displays the information about the prop. The user can also hire and add the item to the basket which fulfils the objective criteria.
Objective 2.4	True	The results are displayed with the use of results table. This allows important information to be displayed while also providing space for the image of the prop. Another feature is the ability to scroll through the props which the user found easy and simple to use due to the similarity to other programs.
Objective 3	True	The recommendation system looks at the orders table to find the recent hires of the user. It can then recommend the most similar props in the database based on the characteristics which match.
Objective 3.1	True	The user can easily navigate to the recommendations since all that has to be done is the press of a button.
Objective 3.2	True	Once the user has selected recommendations they are displayed easily to the user through the use of the results table. This has the advantage of keeping the user familiar with the interface and allowing props to be hired and viewed easily.
Objective 3.3	True	The recommendation system works based on the previous items hired. It generates a similarity index for each of the items in the database which it can then recommend to the user. It also does not count the same props which have already been hired.
Objective 4	True	The basket system allows for quick browsing of props since they can be added while looking at the details of the props. This processes all the orders at the end of hiring which makes the process quicker.
Objective 4.1	True	The button is found on the item detail page and is next to the hire button which means the user can easily find the button. It is also clearly labelled which removes confusion of the function.

Objective 4.2	True	Whenever the add to basket button is pressed the existing basket is ordered and checked to determine the validity of the item. This prevents confusion when adding items to the basket. A message is also displayed when there is an error so the user understands why certain items cannot be added.
Objective 4.3	True	The list which stores the ID of the props added is generated and cleared when a user logs in and logs out. This prevents there being an overlap when different user's login.
Objective 4.4	True	On the search page the user can conclude and hire all the items found in the users basket. This uses the same process as a normal hire and generates the order and records the information.
Objective 4.5	True	The results table is used once again to display the current basket which is shown when the show basket button is pressed. This reduces the redundancy of code and keeps the user using a familiar interface which was an important objective of the client.
Objective 4.6	True	Once the basket has been displayed the props can be selected to remove the item from the basket. This all takes place using clearly labelled buttons which makes the process easy.
Objective 5	True	Overall the hiring process for props is easy with most elements being handled automatically such as the record generation. The ability to add props to the system was appreciated by the client as it allows the program to grow in the future.
Objective 5.1	True	When the hire button is pressed it automatically creates the order based on the users ID and sets the availability to false in the database.
Objective 5.2	True	The page which is used to returning items is basic but this allows the process to be easy. Only the item ID is required and a message is displayed when successful.
Objective 5.3	True	The form to enter a new prop uses a file selection that allows the user to easily find their image. The other fields can have values entered to store information. When the add button is pressed the presence of an image file is checked and the item is automatically added to the database with the values.

Stake Holder Feedback:

Once the project was completed, I emailed both the main client and the prop master who manages the warehouse. This allowed me to gain a perspective on how successful the project was and whether there were elements which could be improved for next time. By interviewing more than one stake holder I was able to get opinions on the different functions of the system.

Main Client (Set decorator)

1. How easy did you find the program to use?

I felt that it was extremely easy to use, and I rarely find that I'm confused or don't know the process for getting things done. This is probably because of the clear buttons which allow me to navigate between the windows, normally with programs I find that the different symbols make understanding difficult. I also like the use of the central search screen from which all other functions branch off from, this allows me to navigate to the search page whenever I'm looking for a button.

2. Do you think this system has made day to day work easier?

I think it has absolutely improved things, mostly because of how much quicker things are. There is far less walking and looking for a specific item, instead I can just search for it on the system and see whether its available or not. It also makes tracking how long everything takes so much easier since you can see when the prop is going to next be available. The basket system also makes browsing easier, its familiar and quick so I use it every time I want to hire a prop.

3. Would there be anything which could be improved upon in the future?

By storing more information about the props I might be able to hire props without inspecting in person, for example the condition the prop is in could be a characteristic stored. However I understand that this would be more work when submitting props and could put strain on the system.

Other Stakeholder (prop master)

1. How easy did you find the program to use?

I think things are really simple on this new system, it feels familiar to work on most of the time. Mostly since each page has a single function apart from the search page. Another reason is because of the message pop ups which explain what happened, it stops the problem where you press a button and worry whether the change has been made or not.

2. Do you think this system has made day to day work easier?

Prior to this there was a complicated system of using emails and paper notes to keep track of orders and inventory, now it's all managed automatically which means I can now focus more of my attention on keeping the inventory in good condition and managing my team. The main time sink before was the use of emails so I'm glad that is no longer a factor.

3. Would there be anything which could be improved upon in the future?

I think currently the system works well, by adding more features it could make the workflow more complicated. Potentially generating a pdf or receipt for each order could help since it would allow the sheet to be printed and handed to the delivery team, this could make their job easier.

Possible Future Extensions for the Program and Improvements:

Overall, from the responses with the client and stakeholders I believe that this project has been able to successfully deliver the objectives originally set by overcoming some of the issues which were initially posed. The 5 main objectives have all been successfully completed including the sub goals which suggests that the planning and timeframe were appropriately applied. Both testers found that the program was easy to use and intuitive which was one of the main concerns going into the project due to the reluctance to use new technology from many of the stakeholders involved. It has improved the process of their work by making it easier and more efficient.

However due to the timeframe restriction placed on the project it means that there are additional features which could not be included, an example is the use of a web-based interface which could mean the system would be able to expand to other departments such as the delivery team. It would also provide the benefit of being able to browse the warehouse from a different location, this could be relevant due to the restrictions put in place due to covid. This addition fell outside of the bounds for this project since it would have led to greater development time and testing procedure.

Appendix:

UI Manager Class:

```
1. package netflixpropstorage;
2.
3. //class that handles displaying the different JFrames
4. public class UIManager {
5.     loginPage loginPage = new loginPage();
6.     searchPage searchPage = new searchPage();
7.     propDetails propDetails = new propDetails();
8.     createAccount createAccount = new createAccount();
9.     addProp addProp = new addProp();
10.    addPropExisting addPropExisting = new addPropExisting();
11.    basketCheck basketCheck = new basketCheck();
12.
13.
14.    //to transition from the login page to the search page
15.    void loginToSearch() {
16.        loginPage.setVisible(false);
17.        loginPage.dispose();
18.        searchPage.setVisible(true);
19.
20.    }
21.
22.    //used to transition from the search page to the login page
23.    //executed when the log out button is pressed
24.    void searchToLogin() {
25.        loginPage.setVisible(true);
26.        searchPage.setVisible(false);
27.
28.    }
29.
30.    //transition from the login page to the create account page
31.    void loginToCreateAccount() {
32.        loginPage.setVisible(false);
33.        createAccount.setVisible(true);
34.    }
35.
36.    //transition from the create account page to the login page
37.    //used once the account has been created
38.    void createAccountToLogin() {
39.        loginPage.setVisible(true);
40.        createAccount.setVisible(false);
41.    }
42.
43.    //to move from the search page to the add prop page
44.    void searchToAddProp() {
45.        searchPage.setVisible(false);
46.        addProp.setVisible(true);
47.    }
48.
49.    //from the add prop page to return existing prop page
50.    void addPropToExisting() {
51.        addProp.setVisible(false);
52.        addPropExisting.setVisible(true);}}
```

Add Prop Class:

```
1. package netflixpropstorage;
2.
3. import java.awt.Color;
4. import javax.swing.JFileChooser;
5. import javax.swing.JOptionPane;
6.
7.
8. public class addProp extends javax.swing.JFrame {
9.
10.     public addProp() {
11.         getContentPane().setBackground(Color.WHITE);
12.         this.setExtendedState(this.MAXIMIZED_BOTH);
13.         initComponents();
14.         setDefaultCloseOperation(this.DISPOSE_ON_CLOSE);
15.     }
16.
17.
18.     @SuppressWarnings("unchecked")
19.     // <editor-fold defaultstate="collapsed" desc="Generated Code">
20.     private void initComponents() {
21.
22.         jLabel1 = new javax.swing.JLabel();
23.         getPhotoButton = new javax.swing.JButton();
24.         filePathLabel = new javax.swing.JLabel();
25.         jLabel13 = new javax.swing.JLabel();
26.         typeCombo = new javax.swing.JComboBox < > ();
27.         jLabel14 = new javax.swing.JLabel();
28.         colourCombo = new javax.swing.JComboBox < > ();
29.         jLabel15 = new javax.swing.JLabel();
30.         periodCombo = new javax.swing.JComboBox < > ();
31.         jLabel16 = new javax.swing.JLabel();
32.         eraCombo = new javax.swing.JComboBox < > ();
33.         jLabel17 = new javax.swing.JLabel();
34.         descriptionField = new javax.swing.JTextField();
35.         submitPropButton = new javax.swing.JButton();
36.         returnHired = new javax.swing.JButton();
37.
38.         setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);
39.         setBackground(new java.awt.Color(255, 255, 255));
40.         setMinimumSize(new java.awt.Dimension(1920, 1080));
41.         setPreferredSize(new java.awt.Dimension(1920, 1080));
42.         getContentPane().setLayout(new org.netbeans.lib.awtextra.AbsoluteLayout());
43.
44.         jLabel1.setFont(new java.awt.Font("Bebas Neue", 0, 72)); // NOI18N
45.         jLabel1.setForeground(new java.awt.Color(229, 9, 20));
46.         jLabel1.setHorizontalAlignment(javax.swing.SwingConstants.CENTER);
47.         jLabel1.setText("Add Prop Page:");
48.         getContentPane().add(jLabel1, new org.netbeans.lib.awtextra.AbsoluteConstraints(-150, 40, 1920, 58));
49.
50.         getPhotoButton.setBackground(new java.awt.Color(255, 255, 255));
51.         getPhotoButton.setFont(new java.awt.Font("Bebas Neue", 0, 24)); // NOI18N
52.         getPhotoButton.setForeground(new java.awt.Color(229, 9, 20));
53.         getPhotoButton.setText("Get Photo File Path");
54.         getPhotoButton.addActionListener(new java.awt.event.ActionListener() {
55.             public void actionPerformed(java.awt.event.ActionEvent evt) {
56.                 getPhotoButtonActionPerformed(evt);
57.             }
58.         });
```

```
59.         getContentPane().add(getPhotoButton, new org.netbeans.lib.awtextra.AbsoluteConstraints(580, 160, -1,
        -1));
60.
61.         filePathLabel.setFont(new java.awt.Font("Bebas Neue", 0, 24)); // NOI18N
62.         filePathLabel.setForeground(new java.awt.Color(229, 9, 20));
63.         filePathLabel.setText("This is the file path: none");
64.         getContentPane().add(filePathLabel, new org.netbeans.lib.awtextra.AbsoluteConstraints(580, 230, 690,
        -1));
65.
66.         jLabel13.setFont(new java.awt.Font("Bebas Neue", 0, 24)); // NOI18N
67.         jLabel13.setForeground(new java.awt.Color(229, 9, 20));
68.         jLabel13.setText("Prop Type:");
69.         getContentPane().add(jLabel13, new org.netbeans.lib.awtextra.AbsoluteConstraints(640, 290, 142, -1));
70.
71.         typeCombo.setModel(new javax.swing.DefaultComboBoxModel < > (new String[] {
72.             "Other",
73.             "Chair",
74.             "Light",
75.             "Sofa",
76.             "Table Big",
77.             "Table Small",
78.             " "
79.         }));
80.         getContentPane().add(typeCombo, new org.netbeans.lib.awtextra.AbsoluteConstraints(850, 300, 109, -
        1));
81.
82.         jLabel14.setFont(new java.awt.Font("Bebas Neue", 0, 24)); // NOI18N
83.         jLabel14.setForeground(new java.awt.Color(229, 9, 20));
84.         jLabel14.setText("Prop Colour:");
85.         getContentPane().add(jLabel14, new org.netbeans.lib.awtextra.AbsoluteConstraints(640, 340, 142, -1));
86.
87.         colourCombo.setModel(new javax.swing.DefaultComboBoxModel < > (new String[] {
88.             "Other",
89.             "Red",
90.             "Green",
91.             "Yellow",
92.             "Blue",
93.             "Orange",
94.             "Pink/Purple",
95.             " "
96.         }));
97.         getContentPane().add(colourCombo, new org.netbeans.lib.awtextra.AbsoluteConstraints(850, 350, 109, -
        1));
98.
99.         jLabel15.setFont(new java.awt.Font("Bebas Neue", 0, 24)); // NOI18N
100.        jLabel15.setForeground(new java.awt.Color(229, 9, 20));
101.        jLabel15.setText("Prop Period:");
102.        getContentPane().add(jLabel15, new org.netbeans.lib.awtextra.AbsoluteConstraints(640, 390, 142, -1));
103.
104.        periodCombo.setModel(new javax.swing.DefaultComboBoxModel < > (new String[] {
105.            "Other",
106.            "Georgian",
107.            "Victorian",
108.            "Retro",
109.            "Modern"
110.        }));
111.        getContentPane().add(periodCombo, new org.netbeans.lib.awtextra.AbsoluteConstraints(850, 400, 109, -
        1));
112.
113.        jLabel16.setFont(new java.awt.Font("Bebas Neue", 0, 24)); // NOI18N
114.        jLabel16.setForeground(new java.awt.Color(229, 9, 20));
```

```

115.     jLabel6.setText("Prop Era:");
116.     getContentPane().add(jLabel6, new org.netbeans.lib.awtextra.AbsoluteConstraints(640, 440, 142, -1));
117.
118.     eraCombo.setModel(new javax.swing.DefaultComboBoxModel < > (new String[] {
119.         "Other",
120.         "1900",
121.         "1910",
122.         "1920",
123.         "1930",
124.         "1940",
125.         "1950",
126.         "1960",
127.         "1970",
128.         "1980"
129.     }));
130.     getContentPane().add(eraCombo, new org.netbeans.lib.awtextra.AbsoluteConstraints(850, 450, 109, -1));
131.
132.     jLabel7.setFont(new java.awt.Font("Bebas Neue", 0, 24)); // NOI18N
133.     jLabel7.setForeground(new java.awt.Color(229, 9, 20));
134.     jLabel7.setText("Prop Description:");
135.     getContentPane().add(jLabel7, new org.netbeans.lib.awtextra.AbsoluteConstraints(640, 490, 142, -1));
136.
137.     descriptionField.setHorizontalAlignment(javax.swing.JTextField.LEFT);
138.     getContentPane().add(descriptionField, new org.netbeans.lib.awtextra.AbsoluteConstraints(850, 490,
139.         210, 27));
140.
141.     submitPropButton.setBackground(new java.awt.Color(255, 255, 255));
142.     submitPropButton.setFont(new java.awt.Font("Bebas Neue", 0, 24)); // NOI18N
143.     submitPropButton.setForeground(new java.awt.Color(229, 9, 20));
144.     submitPropButton.setText("Submit Prop");
145.     submitPropButton.addActionListener(new java.awt.event.ActionListener() {
146.         public void actionPerformed(java.awt.event.ActionEvent evt) {
147.             submitPropButtonActionPerformed(evt);
148.         }
149.     });
150.     getContentPane().add(submitPropButton, new org.netbeans.lib.awtextra.AbsoluteConstraints(740, 560,
151.         134, 54));
152.
153.     returnHired.setBackground(new java.awt.Color(255, 255, 255));
154.     returnHired.setFont(new java.awt.Font("Bebas Neue", 0, 24)); // NOI18N
155.     returnHired.setForeground(new java.awt.Color(229, 9, 20));
156.     returnHired.setText("Return a hired prop");
157.     returnHired.addActionListener(new java.awt.event.ActionListener() {
158.         public void actionPerformed(java.awt.event.ActionEvent evt) {
159.             returnHiredActionPerformed(evt);
160.         }
161.     });
162.     getContentPane().add(returnHired, new org.netbeans.lib.awtextra.AbsoluteConstraints(830, 160, 232,
163.         40));
164.
165.     pack();
166. } // </editor-fold>
167.
168. //run when the get photo button is pressed
169. private void getPhotoButtonActionPerformed(java.awt.event.ActionEvent evt) {
170.     JFileChooser file = new JFileChooser("D:\\Homework A Levels\\Computing\\NEA\\projectImages");
171.     //creates a file chooser button
172.     file.setMultiSelectionEnabled(false); //allows the user to only select one file
173.     file.setFileSelectionMode(JFileChooser.FILES_AND_DIRECTORIES); //allows both files and directories to
174.     be displayed
175.     file.setFileHidingEnabled(false); //shows file

```

```

171.         if (file.showOpenDialog(null) == JFileChooser.APPROVE_OPTION) {
172.             java.io.File f = file.getSelectedFile(); //gets the file path of the selected file from the file
               chooser
173.             filePathLabel.setText(f.getPath()); //sets the label to the path
174.         }
175.
176.     }
177.
178.     //run when the submit prop button is pressed
179.     private void submitPropButtonActionPerformed(java.awt.event.ActionEvent evt) {
180.         String imagePath = filePathLabel.getText(); //gets the file path from the value which is found in the
               label
181.         if (imagePath.equals("This is the file path: none")) { //determines whether a file has been selected
182.             JOptionPane.showMessageDialog(null, "There is no prop image selected");
183.         } else {
184.             imagePath = imagePath.replace("\\", "\\"); //adds the escape characters to allow the file path
               to function when stored
185.             sqlConnection sc = new sqlConnection();
186.             sc.addProp((String) typeCombo.getSelectedItem(), (String) colourCombo.getSelectedItem(), (String)
               periodCombo.getSelectedItem(), (String) eraCombo.getSelectedItem(), descriptionField.getText(), imagePath);
187.             //creates a new prop using the characteristics entered
188.         }
189.     }
190.
191.     //transitions to the return a hired prop page when button pressed
192.     private void returnHiredActionPerformed(java.awt.event.ActionEvent evt) {
193.         UIManager uiMan = new UIManager();
194.         uiMan.addPropToExisting();
195.     }
196.
197.
198.     public static void main(String args[]) {
199.         /* Set the Nimbus look and feel */
200.         //<editor-fold defaultstate="collapsed" desc=" Look and feel setting code (optional) ">
201.         /* If Nimbus (introduced in Java SE 6) is not available, stay with the default look and feel.
202.          * For details see http://download.oracle.com/javase/tutorial/uiswing/lookandfeel/plaf.html
203.          */
204.         try {
205.             for (javax.swing.UIManager.LookAndFeelInfo info:
               javax.swing.UIManager.getInstalledLookAndFeels()) {
206.                 if ("Nimbus".equals(info.getName())) {
207.                     javax.swing.UIManager.setLookAndFeel(info.getClassName());
208.                     break;
209.                 }
210.             }
211.         } catch (ClassNotFoundException ex) {
212.             java.util.logging.Logger.getLogger(addProp.class.getName()).log(java.util.logging.Level.SEVERE,
               null, ex);
213.         } catch (InstantiationException ex) {
214.             java.util.logging.Logger.getLogger(addProp.class.getName()).log(java.util.logging.Level.SEVERE,
               null, ex);
215.         } catch (IllegalAccessException ex) {
216.             java.util.logging.Logger.getLogger(addProp.class.getName()).log(java.util.logging.Level.SEVERE,
               null, ex);
217.         } catch (javax.swing.UnsupportedLookAndFeelException ex) {
218.             java.util.logging.Logger.getLogger(addProp.class.getName()).log(java.util.logging.Level.SEVERE,
               null, ex);
219.         }
220.         //</editor-fold>
221.
222.         /* Create and display the form */

```

```

223.     java.awt.EventQueue.invokeLater(new Runnable() {
224.         public void run() {
225.             new addProp().setVisible(true);
226.         }
227.     });
228. }
229.
230. // Variables declaration - do not modify
231. private javax.swing.JComboBox < String > colourCombo;
232. private javax.swing.JTextField descriptionField;
233. private javax.swing.JComboBox < String > eraCombo;
234. private javax.swing.JLabel filePathLabel;
235. private javax.swing.JButton getPhotoButton;
236. private javax.swing.JLabel jLabel1;
237. private javax.swing.JLabel jLabel3;
238. private javax.swing.JLabel jLabel4;
239. private javax.swing.JLabel jLabel5;
240. private javax.swing.JLabel jLabel6;
241. private javax.swing.JLabel jLabel7;
242. private javax.swing.JComboBox < String > periodCombo;
243. private javax.swing.JButton returnHired;
244. private javax.swing.JButton submitPropButton;
245. private javax.swing.JComboBox < String > typeCombo;
246. // End of variables declaration
247. }

```

Add Existing Prop Class:

```

1. package netflixpropstorage;
2.
3. import java.awt.Color;
4. import javax.swing.JOptionPane;
5.
6. public class addPropExisting extends javax.swing.JFrame {
7.
8.     public addPropExisting() {
9.         getContentPane().setBackground(Color.WHITE); //set the background colour
10.        getContentPane().setSize(714, 361); //set the preferred size
11.        initComponents();
12.        setLocationRelativeTo(null); //centre the window when shown
13.        setDefaultCloseOperation(this.DISPOSE_ON_CLOSE);
14.    }
15.
16.    @SuppressWarnings("unchecked")
17.    // <editor-fold defaultstate="collapsed" desc="Generated Code">
18.    private void initComponents() {
19.
20.        jLabel1 = new javax.swing.JLabel();
21.        jLabel2 = new javax.swing.JLabel();
22.        idField = new javax.swing.JTextField();
23.        submitButton = new javax.swing.JButton();
24.
25.        setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);
26.        setMinimumSize(new java.awt.Dimension(714, 361));
27.        getContentPane().setLayout(new org.netbeans.lib.awtextra.AbsoluteLayout());

```



```

28.
29.     jLabel1.setFont(new java.awt.Font("Bebas Neue", 0, 48)); // NOI18N
30.     jLabel1.setForeground(new java.awt.Color(229, 9, 20));
31.     jLabel1.setHorizontalAlignment(javax.swing.SwingConstants.CENTER);
32.     jLabel1.setText("Return Existing Prop");
33.     getContentPane().add(jLabel1, new org.netbeans.lib.awtextra.AbsoluteConstraints(0, 20, 710, 74));
34.
35.     jLabel2.setFont(new java.awt.Font("Bebas Neue", 0, 18)); // NOI18N
36.     jLabel2.setForeground(new java.awt.Color(229, 9, 20));
37.     jLabel2.setText("Enter the prop id:");
38.     getContentPane().add(jLabel2, new org.netbeans.lib.awtextra.AbsoluteConstraints(200, 150, 120, -1));
39.
40.     idField.setFont(new java.awt.Font("Calibri", 0, 14)); // NOI18N
41.     idField.setForeground(new java.awt.Color(229, 9, 20));
42.     idField.setMinimumSize(new java.awt.Dimension(714, 361));
43.     getContentPane().add(idField, new org.netbeans.lib.awtextra.AbsoluteConstraints(360, 150, 123, -1));
44.
45.     submitButton.setBackground(new java.awt.Color(255, 255, 255));
46.     submitButton.setFont(new java.awt.Font("Bebas Neue", 0, 14)); // NOI18N
47.     submitButton.setForeground(new java.awt.Color(229, 9, 20));
48.     submitButton.setText("Submit");
49.     submitButton.addActionListener(new java.awt.event.ActionListener() {
50.         public void actionPerformed(java.awt.event.ActionEvent evt) {
51.             submitButtonActionPerformed(evt);
52.         }
53.     });
54.     getContentPane().add(submitButton, new org.netbeans.lib.awtextra.AbsoluteConstraints(282, 237, 90, -
55. 1));
56.     pack();
57. } // </editor-fold>
58.
59. //run when the submit button is pressed
60. private void submitButtonActionPerformed(java.awt.event.ActionEvent evt) {
61.     sqlConnection sC = new sqlConnection();
62.     sC.returnProp(idField.getText()); //changes the availability value of the corresponding prop
63.     JOptionPane.showMessageDialog(null, "This prop has been returned");
64. }
65.
66. public static void main(String args[]) {
67.     /* Set the Nimbus look and feel */
68.     //<editor-fold defaultstate="collapsed" desc=" Look and feel setting code (optional) ">
69.     /* If Nimbus (introduced in Java SE 6) is not available, stay with the default look and feel.
70.      * For details see http://download.oracle.com/javase/tutorial/uiswing/lookandfeel/plaf.html
71.      */
72.     try {
73.         for (javax.swing.UIManager.LookAndFeelInfo info:
74.             javax.swing.UIManager.getInstalledLookAndFeels()) {
75.             if ("Nimbus".equals(info.getName())) {
76.                 javax.swing.UIManager.setLookAndFeel(info.getClassName());
77.                 break;
78.             }
79.         } catch (ClassNotFoundException ex) {
80.             java.util.logging.Logger.getLogger(addPropExisting.class.getName()).log(java.util.logging.Level.S
81. EVERE, null, ex);
82.         } catch (InstantiationException ex) {
83.             java.util.logging.Logger.getLogger(addPropExisting.class.getName()).log(java.util.logging.Level.S
EVERE, null, ex);
84.         } catch (IllegalAccessException ex) {

```

```

84.         java.util.logging.Logger.getLogger(addPropExisting.class.getName()).log(java.util.logging.Level.S
EVERE, null, ex);
85.     } catch (javax.swing.UnsupportedLookAndFeelException ex) {
86.         java.util.logging.Logger.getLogger(addPropExisting.class.getName()).log(java.util.logging.Level.S
EVERE, null, ex);
87.     }
88.     //</editor-fold>
89.
90.     /* Create and display the form */
91.     java.awt.EventQueue.invokeLater(new Runnable() {
92.         public void run() {
93.             new addPropExisting().setVisible(true);
94.         }
95.     });
96. }
97.
98. // Variables declaration - do not modify
99. private javax.swing.JTextField idField;
100. private javax.swing.JLabel jLabel1;
101. private javax.swing.JLabel jLabel2;
102. private javax.swing.JButton submitButton;
103. // End of variables declaration
104. }

```

Basket Manager Class:

```

1. package netflixpropstorage;
2. import java.util.ArrayList;
3.
4. public class basketManager {
5.     ArrayList < Integer > basket = new ArrayList < Integer > (); //an array list of prop ID which will be
sorted and searched
6.
7.     //declares the node object which will populate the binary search tree
8.     class Node {
9.         int key; //declares the key as an int
10.        Node left, right; //declares the left and right node to be generated
11.
12.        public Node(int item) {
13.            key = item; //sets the value of key to the item from parameter
14.            left = right = null; //null values to the left and right nodes
15.        }
16.    }
17.
18.    //the root node of the binary search tree
19.    Node root;
20.
21.    //constuctor for the class
22.    basketManager() {
23.        root = null;
24.    }
25.
26.    //insert a new value into the tree through insert rec
27.    void insert(int key) {
28.        root = insertRec(root, key);
29.    }

```

```
30.
31.    //A recursive function to insert a new key into binary search tree
32.    Node insertRec(Node root, int key) {
33.
34.        //check if the tree is empty and return a new node if so
35.        if (root == null) {
36.            root = new Node(key);
37.            return root; //returns the new root node
38.        }
39.
40.        //if the tree isnt empty then recursively loop down the tree
41.        if (key < root.key)
42.            root.left = insertRec(root.left, key); //insert left if less
43.        else if (key > root.key)
44.            root.right = insertRec(root.right, key); //insert right if more
45.
46.
47.        return root;
48.    }
49.
50.    //traverse the binary tree in order based on the initial node
51.    void inorderRec(Node root) {
52.        if (root != null) {
53.            inorderRec(root.left);
54.            loginPage lp = new loginPage();
55.            lp.tempBasketArray.add(root.key); //calls the function and adds the value to the array
56.            inorderRec(root.right);
57.
58.
59.        }
60.    }
61.
62.    //inserts new values into the tree based on the size of the array
63.    void treeins(ArrayList < Integer > arr) {
64.        for (int i = 0; i < arr.size(); i++) {
65.            insert(arr.get(i)); //function used to create new node
66.        }
67.
68.    }
69.
70.    //used to apply a binary search to the now ordered list
71.    //takes the sorted list , search value , start and end values
72.    public int binarySearch(ArrayList < Integer > sortedArray, int value, int first, int last) {
73.        int middle = first + ((last - first) / 2); //determines the middle point of the array
74.
75.        if (last < first) {
76.            return -1; //return -1 if the values entered are not valid or search value not found
77.        }
78.
79.        if (value == sortedArray.get(middle)) {
80.            return middle; //return the middle index if the value is found
81.        } else if (value < sortedArray.get(middle)) {
82.            return binarySearch(sortedArray, value, first, middle - 1); //applies search again if not found
83.        } else {
84.            return binarySearch(sortedArray, value, middle + 1, last); //-1 or +1 if greater or less than
85.            middle
86.        }
87.
88.        //applied each time an item is added to basket
89.        public int addToBasket(ArrayList < Integer > arr, int searchValue) {
```

```
90.     loginPage lP = new loginPage();
91.     basketManager bM = new basketManager();
92.     bM.treeins(arr); //generates the binary tree
93.     bM.inorderRec(bM.root); //orders the list through an in order traversal
94.
95.     return binarySearch(lP.tempBasketArray, searchValue, 0, arr.size() - 1);
96.     //returns the position of the item in the array
97. }
98.
99. public static void main(String args[]) {}
100. }
```

Create Account Class:

```
1. package netflixpropstorage;
2.
3. import java.awt.Color;
4. import java.util.regex.Matcher;
5. import java.util.regex.Pattern;
6. import javax.swing.JOptionPane;
7.
8. public class createAccount extends javax.swing.JFrame {
9.
10.     public createAccount() {
11.         getContentPane().setBackground(Color.WHITE);
12.         initComponents();
13.         setDefaultCloseOperation(this.DISPOSE_ON_CLOSE);
14.     }
15.
16.     @SuppressWarnings("unchecked")
17.     // <editor-fold defaultstate="collapsed" desc="Generated Code">
18.     private void initComponents() {
19.
20.         jLabel1 = new javax.swing.JLabel();
21.         firstNameLabel = new javax.swing.JLabel();
22.         firstNameField = new javax.swing.JTextField();
23.         surnameLabel = new javax.swing.JLabel();
24.         surnameField = new javax.swing.JTextField();
25.         emailLabel = new javax.swing.JLabel();
26.         emailField = new javax.swing.JTextField();
27.         phoneNumberLabel = new javax.swing.JLabel();
28.         phoneNumberField = new javax.swing.JTextField();
29.         jobRoleLabel = new javax.swing.JLabel();
30.         jobRoleCombo = new javax.swing.JComboBox < > ();
31.         phoneNumberLabel1 = new javax.swing.JLabel();
32.         currentProductionField = new javax.swing.JTextField();
33.         phoneNumberLabel2 = new javax.swing.JLabel();
34.         passwordField = new javax.swing.JTextField();
35.         createAccountButton = new javax.swing.JButton();
36.
37.         setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);
38.
39.         jLabel1.setFont(new java.awt.Font("Bebas Neue", 0, 48)); // NOI18N
40.         jLabel1.setForeground(new java.awt.Color(229, 9, 20));
41.         jLabel1.setHorizontalAlignment(javax.swing.SwingConstants.CENTER);
```

```
42.        jLabel1.setText("Account Creation:");
43.
44.        firstNameLabel.setFont(new java.awt.Font("Bebas Neue", 0, 24)); // NOI18N
45.        firstNameLabel.setForeground(new java.awt.Color(229, 9, 20));
46.        firstNameLabel.setText("First Name");
47.
48.        surnameLabel.setFont(new java.awt.Font("Bebas Neue", 0, 24)); // NOI18N
49.        surnameLabel.setForeground(new java.awt.Color(229, 9, 20));
50.        surnameLabel.setText("Surname");
51.
52.        emailLabel.setFont(new java.awt.Font("Bebas Neue", 0, 24)); // NOI18N
53.        emailLabel.setForeground(new java.awt.Color(229, 9, 20));
54.        emailLabel.setText("Email");
55.
56.        phoneNumberLabel.setFont(new java.awt.Font("Bebas Neue", 0, 24)); // NOI18N
57.        phoneNumberLabel.setForeground(new java.awt.Color(229, 9, 20));
58.        phoneNumberLabel.setText("Phone Number");
59.
60.        jobRoleLabel.setFont(new java.awt.Font("Bebas Neue", 0, 24)); // NOI18N
61.        jobRoleLabel.setForeground(new java.awt.Color(229, 9, 20));
62.        jobRoleLabel.setText("Job Role Label");
63.
64.        jobRoleCombo.setModel(new javax.swing.DefaultComboBoxModel < > (new String[] {
65.            "SetDec",
66.            "PropMan",
67.            "Runner",
68.            "Apprentice",
69.            " "
70.        }));
71.
72.        phoneNumberLabel1.setFont(new java.awt.Font("Bebas Neue", 0, 24)); // NOI18N
73.        phoneNumberLabel1.setForeground(new java.awt.Color(229, 9, 20));
74.        phoneNumberLabel1.setText("Current Production");
75.
76.        phoneNumberLabel2.setFont(new java.awt.Font("Bebas Neue", 0, 24)); // NOI18N
77.        phoneNumberLabel2.setForeground(new java.awt.Color(229, 9, 20));
78.        phoneNumberLabel2.setText("Account Password");
79.
80.        createAccountButton.setBackground(new java.awt.Color(255, 255, 255));
81.        createAccountButton.setFont(new java.awt.Font("Bebas Neue", 0, 24)); // NOI18N
82.        createAccountButton.setForeground(new java.awt.Color(229, 9, 20));
83.        createAccountButton.setText("Create Account");
84.        createAccountButton.addActionListener(new java.awt.event.ActionListener() {
85.            public void actionPerformed(java.awt.event.ActionEvent evt) {
86.                createAccountButtonActionPerformed(evt);
87.            }
88.        });
89.
90.        javax.swing.GroupLayout layout = new javax.swing.GroupLayout(getContentPane());
91.        getContentPane().setLayout(layout);
92.        layout.setHorizontalGroup(
93.            layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
94.                .addGroup(layout.createSequentialGroup()
95.                    .addComponent(jLabel1, javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
96.                    .addGroup(layout.createSequentialGroup()
97.                        .addContainerGap(578, Short.MAX_VALUE)
98.                        .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
99.                            .addGroup(layout.createSequentialGroup()
100.                                .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.TRAILING)
101.                                    .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.TRAILING)
```

```

102.                .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.TR
    AILING)
103.                .addComponent(surnameLabel, javax.swing.GroupLayout.PREFERRED_SIZE,
    94, javax.swing.GroupLayout.PREFERRED_SIZE)
104.                .addComponent(firstNameLabel, javax.swing.GroupLayout.PREFERRED_SIZE,
    94, javax.swing.GroupLayout.PREFERRED_SIZE))
105.                .addGap(65, 65, 65))
106.                .addGroup(javax.swing.GroupLayout.Alignment.LEADING,
    layout.createSequentialGroup())
107.                .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.TR
    AILING)
108.                .addComponent(jobRoleLabel,
    javax.swing.GroupLayout.Alignment.LEADING)
109.                .addComponent(phoneNumberLabel,
    javax.swing.GroupLayout.Alignment.LEADING, javax.swing.GroupLayout.PREFERRED_SIZE, 124,
    javax.swing.GroupLayout.PREFERRED_SIZE))
110.                .addGap(36, 36, 36)))
111.                .addComponent(emailLabel, javax.swing.GroupLayout.PREFERRED_SIZE, 94,
    javax.swing.GroupLayout.PREFERRED_SIZE)
112.                .addComponent(phoneNumberLabel2, javax.swing.GroupLayout.PREFERRED_SIZE, 160,
    javax.swing.GroupLayout.PREFERRED_SIZE)
113.                .addComponent(phoneNumberLabel1))
114.                .addGap(28, 28, 28)
115.                .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
116.                .addComponent(passwordField, javax.swing.GroupLayout.PREFERRED_SIZE, 125,
    javax.swing.GroupLayout.PREFERRED_SIZE)
117.                .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.TRAILING)
118.                .addComponent(firstNameField, javax.swing.GroupLayout.PREFERRED_SIZE, 125,
    javax.swing.GroupLayout.PREFERRED_SIZE)
119.                .addComponent(surnameField, javax.swing.GroupLayout.PREFERRED_SIZE, 125,
    javax.swing.GroupLayout.PREFERRED_SIZE)
120.                .addComponent(emailField, javax.swing.GroupLayout.PREFERRED_SIZE, 125,
    javax.swing.GroupLayout.PREFERRED_SIZE)
121.                .addComponent(phoneNumberField, javax.swing.GroupLayout.PREFERRED_SIZE, 125,
    javax.swing.GroupLayout.PREFERRED_SIZE))
122.                .addComponent(currentProductionField, javax.swing.GroupLayout.PREFERRED_SIZE,
    125, javax.swing.GroupLayout.PREFERRED_SIZE)
123.                .addComponent(jobRoleCombo, javax.swing.GroupLayout.PREFERRED_SIZE, 125,
    javax.swing.GroupLayout.PREFERRED_SIZE))
124.                .addGap(574, 574, 574))
125.                .addGroup(javax.swing.GroupLayout.Alignment.TRAILING, layout.createSequentialGroup())
126.                .addComponent(createAccountButton, javax.swing.GroupLayout.PREFERRED_SIZE, 175,
    javax.swing.GroupLayout.PREFERRED_SIZE)
127.                .addGap(617, 617, 617)))
128.            );
129.            layout.setVerticalGroup(
130.                layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
131.                .addGroup(layout.createSequentialGroup()
132.                .addGap(28, 28, 28)
133.                .addComponent(jLabel11)
134.                .addGap(53, 53, 53)
135.                .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
136.                .addComponent(firstNameLabel)
137.                .addComponent(firstNameField, javax.swing.GroupLayout.PREFERRED_SIZE,
    javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE))
138.                .addGap(18, 18, 18)
139.                .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
140.                .addGroup(layout.createSequentialGroup()
141.                .addComponent(surnameLabel)
142.                .addGap(18, 18, 18)
143.                .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.BASELINE)

```

```

144.         .addComponent(emailLabel)
145.         .addComponent(emailField, javax.swing.GroupLayout.PREFERRED_SIZE,
    javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE)))
146.         .addComponent(surnameField, javax.swing.GroupLayout.PREFERRED_SIZE,
    javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE))
147.         .addGap(18, 18, 18)
148.         .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
149.             .addComponent(phoneNumberLabel)
150.             .addComponent(phoneNumberField, javax.swing.GroupLayout.PREFERRED_SIZE,
    javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE))
151.         .addGap(22, 22, 22)
152.         .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
153.             .addComponent(jobRoleLabel)
154.             .addComponent(jobRoleCombo, javax.swing.GroupLayout.PREFERRED_SIZE,
    javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE))
155.         .addGap(18, 18, 18)
156.         .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
157.             .addGroup(layout.createSequentialGroup()
158.                 .addComponent(phoneNumberLabel1)
159.                 .addGap(18, 18, 18)
160.                 .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.BASELINE)
161.                     .addComponent(phoneNumberLabel2)
162.                     .addComponent(passwordField, javax.swing.GroupLayout.PREFERRED_SIZE,
    javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE)))
163.             .addComponent(currentProductionField, javax.swing.GroupLayout.PREFERRED_SIZE,
    javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE))
164.         .addGap(67, 67, 67)
165.         .addComponent(createAccountButton, javax.swing.GroupLayout.PREFERRED_SIZE, 45,
    javax.swing.GroupLayout.PREFERRED_SIZE)
166.         .addContainerGap(189, Short.MAX_VALUE))
167.     );
168.
169.     pack();
170. } // </editor-fold>
171.
172. //used to determine the presence of a lower case, capital and number
173. private static boolean capitalCheck(String password) {
174.     char ch; //variable which temporarily stores the current character
175.     boolean capitalFlag = false; //flag to check for presence of capital
176.     boolean lowerCaseFlag = false; //flag for lower case
177.     boolean numberFlag = false; //flag for numbers
178.
179.     //loops based on the length of the password
180.     for (int i = 0; i < password.length(); i++) {
181.         ch = password.charAt(i);
182.         if (Character.isDigit(ch)) { //if the character is a number then set number flag
183.             numberFlag = true;
184.         } else if (Character.isUpperCase(ch)) { //if character is a capital then set flag
185.             capitalFlag = true;
186.         } else if (Character.isLowerCase(ch)) { //if the character is lower case then set flag
187.             lowerCaseFlag = true;
188.         }
189.         if (numberFlag && capitalFlag && lowerCaseFlag) //check if all flags are true
190.             return true;
191.     }
192.     return false;
193. }
194.
195. private void createAccountButtonActionPerformed(java.awt.event.ActionEvent evt) {
196.     hashFunction hF = new hashFunction();
197.     sqlConnection sC = new sqlConnection();

```

```

198.         UIManager UIman = new UIManager();
199.
200.         //regex pattern which checks for the presence of @ in the string
201.         Pattern pattern = Pattern.compile(".*@.*", Pattern.CASE_INSENSITIVE);
202.
203.         //applies regex library using previous regex and the email field string
204.         Matcher matcher = pattern.matcher(emailField.getText());
205.         boolean emailCheck = matcher.find(); //true if @ is found
206.
207.         //hash the password for database storage
208.         String passwordHashed = hF.hashFunction(passwordField.getText());
209.
210.         //checks all fields contain the correct characteristics and only runs if all are true
211.         if (emailCheck == true && phoneNumberField.getText().length() == 11 &&
passwordField.getText().matches(".*\\d.*") == true && capitalCheck(passwordField.getText())) {
212.             //generates a new user profile using the characteristics entered as parameters
213.             sC.accountCreation(firstNameField.getText(), surnameField.getText(), emailField.getText(),
phoneNumberField.getText(), (String) jobRoleCombo.getSelectedItem(), currentProductionField.getText(),
passwordHashed);
214.             JOptionPane.showMessageDialog(null, "Account Created Successfully"); //output to user
215.             UIman.createAccountToLogin(); //switch window
216.         }
217.
218.         //executed if the previous if check fails
219.         else {
220.             JOptionPane.showMessageDialog(null, "1. Email must contain @ \n 2. Phone number of 11 characters
\n 3. Number and capital in password");
221.         }
222.
223.
224.
225.     }
226.
227.     public static void main(String args[]) {
228.         /* Set the Nimbus look and feel */
229.         //<editor-fold defaultstate="collapsed" desc=" Look and feel setting code (optional) ">
230.         /* If Nimbus (introduced in Java SE 6) is not available, stay with the default look and feel.
231.          * For details see http://download.oracle.com/javase/tutorial/uiswing/lookandfeel/plaf.html
232.          */
233.         try {
234.             for (javax.swing.UIManager.LookAndFeelInfo info:
javax.swing.UIManager.getInstalledLookAndFeels()) {
235.                 if ("Nimbus".equals(info.getName())) {
236.                     javax.swing.UIManager.setLookAndFeel(info.getClassName());
237.                     break;
238.                 }
239.             }
240.         } catch (ClassNotFoundException ex) {
241.             java.util.logging.Logger.getLogger(createAccount.class.getName()).log(java.util.logging.Level.SEVERE, null, ex);
242.         } catch (InstantiationException ex) {
243.             java.util.logging.Logger.getLogger(createAccount.class.getName()).log(java.util.logging.Level.SEVERE, null, ex);
244.         } catch (IllegalAccessException ex) {
245.             java.util.logging.Logger.getLogger(createAccount.class.getName()).log(java.util.logging.Level.SEVERE, null, ex);
246.         } catch (javax.swing.UnsupportedLookAndFeelException ex) {
247.             java.util.logging.Logger.getLogger(createAccount.class.getName()).log(java.util.logging.Level.SEVERE, null, ex);
248.         }
249.         //</editor-fold>

```



```
250.
251.     /* Create and display the form */
252.     java.awt.EventQueue.invokeLater(new Runnable() {
253.         public void run() {
254.             new createAccount().setVisible(true);
255.         }
256.     });
257. }
258.
259. // Variables declaration - do not modify
260. private javax.swing.JButton createAccountButton;
261. private javax.swing.JTextField currentProductionField;
262. private javax.swing.JTextField emailField;
263. private javax.swing.JLabel emailLabel;
264. private javax.swing.JTextField firstNameField;
265. private javax.swing.JLabel firstNameLabel;
266. private javax.swing.JLabel jLabel1;
267. private javax.swing.JComboBox < String > jobRoleCombo;
268. private javax.swing.JLabel jobRoleLabel;
269. private javax.swing.JTextField passwordField;
270. private javax.swing.JTextField phoneNumberField;
271. private javax.swing.JLabel phoneNumberLabel;
272. private javax.swing.JLabel phoneNumberLabel1;
273. private javax.swing.JLabel phoneNumberLabel2;
274. private javax.swing.JTextField surnameField;
275. private javax.swing.JLabel surnameLabel;
276. // End of variables declaration
277. }
```

Detail Manager Class:

```
1. package netflixpropstorage;
2.
3. import java.awt.Color;
4. import java.awt.Font;
5. import java.awt.Image;
6. import java.awt.event.ActionEvent;
7. import java.awt.event.ActionListener;
8. import java.sql.ResultSet;
9. import java.sql.SQLException;
10. import java.text.ParseException;
11. import java.util.logging.Level;
12. import java.util.logging.Logger;
13. import javax.swing.*;
14.
15. //a class used to generate the page showing a selected prop
16. public class detailManager {
17.
18.     String propID; //stores the id of the prop being displayed
19.     String propAvailable; //determines whether the props is currently on hire
20.
21.     public void generateDetailPage(String col1, String col2, String col3, String col4, String col5, String
        col6, String col7, String col8) {
22.         JFrame f = new JFrame(); //creating instance of JFrame
23.         propID = col1; //sets the id of the prop
```

```
24.      propAvailable = col8; //stores the availability of the prop
25.      f.getContentPane().setBackground(Color.WHITE); //set background to white
26.
27.      //sets the labels to the characteristics of the prop found in the parameters
28.      JLabel detail1 = new JLabel("Prop index: " + col1); //sets the label to the prop id
29.      detail1.setBounds(700, 130, 1000, 40); //determines the size and position of the label
30.      detail1.setFont(new Font("Bebas Neue", Font.PLAIN, 35)); //sets the font and size
31.      detail1.setForeground(new Color(229, 9, 20)); //colour of the label using the rgb value from netflix
32.
33.      JLabel detail2 = new JLabel("Type: " + col2);
34.      detail2.setBounds(700, 190, 1000, 40);
35.      detail2.setFont(new Font("Bebas Neue", Font.PLAIN, 35));
36.      detail2.setForeground(new Color(229, 9, 20));
37.
38.      JLabel detail3 = new JLabel("Colour: " + col3);
39.      detail3.setBounds(700, 250, 1000, 40);
40.      detail3.setFont(new Font("Bebas Neue", Font.PLAIN, 35));
41.      detail3.setForeground(new Color(229, 9, 20));
42.
43.      JLabel detail4 = new JLabel("Period: " + col4);
44.      detail4.setBounds(700, 310, 1000, 40);
45.      detail4.setFont(new Font("Bebas Neue", Font.PLAIN, 35));
46.      detail4.setForeground(new Color(229, 9, 20));
47.
48.      JLabel detail5 = new JLabel("Date: " + col5);
49.      detail5.setBounds(700, 370, 1000, 40);
50.      detail5.setFont(new Font("Bebas Neue", Font.PLAIN, 35));
51.      detail5.setForeground(new Color(229, 9, 20));
52.
53.      JLabel detail6 = new JLabel("Description: " + col6);
54.      detail6.setBounds(700, 430, 1000, 40);
55.      detail6.setFont(new Font("Bebas Neue", Font.PLAIN, 35));
56.      detail6.setForeground(new Color(229, 9, 20));
57.
58.      //creates the image of the prop
59.      ImageIcon icon = new ImageIcon(col7); //stores the file in a icon object
60.      Image scaleImage = icon.getImage().getScaledInstance(450, 450, Image.SCALE_DEFAULT); //scales the
image to the right size
61.      icon = new ImageIcon(scaleImage); //sets the image icon to the the icon so it can be assigned to a
label
62.      JLabel detail7 = new JLabel(icon); //sets the label to the value of the icon
63.      detail7.setBounds(110, 110, 450, 450); //size and position of the image
64.
65.      //title text
66.      JLabel detail8 = new JLabel("Prop Details");
67.      detail8.setBounds(440, 30, 1000, 40); //x axis, y axis, width, height
68.      detail8.setFont(new Font("Bebas Neue", Font.PLAIN, 50));
69.      detail8.setForeground(new Color(229, 9, 20));
70.
71.      f.add(detail1); //adding button in JFrame
72.      f.add(detail2);
73.      f.add(detail3);
74.      f.add(detail4);
75.      f.add(detail5);
76.      f.add(detail6);
77.      f.add(detail7);
78.      f.add(detail8);
79.
80.
81.      JButton b = new JButton("Hire Prop");
82.      b.setBounds(700, 500, 160, 40);
```

```
83.         b.setFont(new Font("Bebas Neue", Font.PLAIN, 20));
84.         b.setForeground(new Color(229, 9, 20));
85.         b.setBackground(Color.white);
86.
87.         JButton c = new JButton("Add to basket");
88.         c.setBounds(700, 550, 160, 40);
89.         c.setFont(new Font("Bebas Neue", Font.PLAIN, 20));
90.         c.setForeground(new Color(229, 9, 20));
91.         c.setBackground(Color.white);
92.
93.         JButton d = new JButton("Return to search");
94.         d.setBounds(700, 600, 160, 40);
95.         d.setFont(new Font("Bebas Neue", Font.PLAIN, 20));
96.         d.setForeground(new Color(229, 9, 20));
97.         d.setBackground(Color.white);
98.
99.         b.addActionListener(new ActionListener() {
100.             public void actionPerformed(ActionEvent e) { //executed when the hire button is pressed
101.                 //checks the validity of the string entered by the user
102.
103.                 //confirms with the user
104.                 int dialogButton = JOptionPane.YES_NO_OPTION;
105.                 int dialogResult = JOptionPane.showConfirmDialog(null, "Are you sure you want to hire this
prop", "Warning", dialogButton);
106.
107.                 if (dialogResult == JOptionPane.YES_OPTION) {} else {
108.                     JOptionPane.showMessageDialog(null, "Prop has not been hired");
109.                 }
110.                 sqlConnection sC = new sqlConnection();
111.                 loginPage lP = new loginPage();
112.                 ResultSet resSet = sC.emailToID(loginPage.emailGlobal); //stores the id of the user
113.                 try {
114.                     if (propAvailable.equals("1")) { //checks if the prop is available to hire
115.                         resSet.first(); //sets the current row to the first row
116.                         String currentJob = resSet.getString("job"); //retrieves the current job
117.                         String userID = resSet.getString("userID"); //retrieves the user id
118.                         sC.addOrder(propID, userID, currentJob); //generates an order
119.                         JOptionPane.showMessageDialog(null, "This prop has been hired by user with email: " +
lP.emailGlobal);
120.                     } else {
121.                         JOptionPane.showMessageDialog(null, "This prop is not available to hire currently");
122.                     }
123.
124.
125.                 } catch (SQLException ex) {
126.                     Logger.getLogger(detailManager.class.getName()).log(Level.SEVERE, null, ex);
127.                 } catch (ParseException ex) {
128.                     Logger.getLogger(detailManager.class.getName()).log(Level.SEVERE, null, ex);
129.                 }
130.             }
131.
132.
133.         });
134.
135.         c.addActionListener(new ActionListener() {
136.             public void actionPerformed(ActionEvent e) { //executed when the add to basket button is pressed
137.
138.                 int dialogButton = JOptionPane.YES_NO_OPTION;
139.                 int dialogResult = JOptionPane.showConfirmDialog(null, "Are you sure you want to add this
prop to basket", "Warning", dialogButton);
140.                 if (dialogResult == JOptionPane.YES_OPTION) {
```

```
141.         basketManager bM = new basketManager();
142.         loginPage lP = new loginPage();
143.         int tempInt = Integer.parseInt(col1); //used to hold the ID of the prop
144.         if (propAvailable.equals("1")) {
145.             if (bM.addToBasket(lP.basketArray, tempInt) == -1) { //used as a check to see if the
item has already been added
146.                 //returns -1 if the item is not found
147.                 JOptionPane.showMessageDialog(null, "This prop has been added to basket");
148.                 lP.basketArray.add(tempInt);
149.             } else {
150.                 JOptionPane.showMessageDialog(null, "This prop is already in the basket so has
not been added");
151.             }
152.         } else {
153.             JOptionPane.showMessageDialog(null, "This prop is not available to hire currently");
154.         }
155.
156.
157.     } else {
158.         JOptionPane.showMessageDialog(null, "This prop has not been added to basket");
159.     }
160.
161.
162. }
163.
164. });
165.
166. //runs when the return search to button pressed
167. d.addActionListener(new ActionListener() {
168.     public void actionPerformed(ActionEvent e) {
169.         f.setVisible(false); //hides the current window to show the search page
170.     }
171. });
172.
173.
174. f.add(b); //adds the buttons to the frame
175. f.add(c);
176. f.add(d);
177.
178. f.setSize(1080, 720);
179. f.setLayout(null); //using no layout managers
180. f.setVisible(true); //making the frame visible
181. f.setDefaultCloseOperation(f.DISPOSE_ON_CLOSE);
182. }
183. }
```

Hash Function Class:

```
1. package netflixpropstorage;
2.
3. import java.math.BigInteger;
4. import java.security.MessageDigest;
5. import java.security.NoSuchAlgorithmException;
6.
7.
8. public class hashFunction {
9.
10.     public static String hashFunction(String password)
11.     {
12.         try {
13.
14.             //creates message digest object from library using MD5 encryption
15.             MessageDigest md = MessageDigest.getInstance("MD5");
16.
17.             //stores an array of bytes based on the input password
18.             byte[] messageDigest = md.digest(password.getBytes());
19.
20.             //convert byte array into big integer which stores all the digits
21.             BigInteger no = new BigInteger(1, messageDigest);
22.
23.             //convert message digest into hex value
24.             String hashtext = no.toString(16);
25.             while (hashtext.length() < 32) {
26.                 hashtext = "0" + hashtext;
27.             }
28.             return hashtext;
29.         }
30.
31.         catch (NoSuchAlgorithmException e) {
32.             throw new RuntimeException(e);
33.         }
34.     }
35.
36.     public static void main(String args[]) {
37.
38.
39.     }
40. }
```

Login Page Class:

```
1. package netflixpropstorage;
2. import java.awt.Color;
3. import java.util.ArrayList;
4.
5. public class loginPage extends javax.swing.JFrame {
6.
7.     //used to temporarily hold the prop ID when applying the tree sort in basket manager
8.     public static ArrayList < Integer > tempBasketArray = new ArrayList < Integer > ();
9.
10.     //the array which stores the ID of all props added to basket during the login session
11.     public static ArrayList < Integer > basketArray = new ArrayList < Integer > ();
```

```
12.
13.     //identifies which user is currently logged in
14.     public static String emailGlobal;
15.
16.     public loginPage() {
17.
18.         getContentPane().setBackground(Color.WHITE); //sets the frame background to white
19.         initComponents(); //adds components
20.         this.setExtendedState(this.MAXIMIZED_BOTH); //sets the frame to the size of the window
21.
22.     }
23.
24.     @SuppressWarnings("unchecked")
25.     // <editor-fold defaultstate="collapsed" desc="Generated Code">
26.     private void initComponents() {
27.
28.         pageTitle = new javax.swing.JLabel();
29.         username = new javax.swing.JTextField();
30.         passwordLabel = new javax.swing.JLabel();
31.         usernameLabel = new javax.swing.JLabel();
32.         login = new javax.swing.JButton();
33.         createAccount = new javax.swing.JButton();
34.         password = new javax.swing.JTextField();
35.         jLabel1 = new javax.swing.JLabel();
36.
37.         setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);
38.         getContentPane().setLayout(new org.netbeans.lib.awtextra.AbsoluteLayout());
39.
40.         pageTitle.setFont(new java.awt.Font("Bebas Neue", 0, 72)); // NOI18N
41.         pageTitle.setForeground(new java.awt.Color(229, 9, 20));
42.         pageTitle.setHorizontalAlignment(javax.swing.SwingConstants.CENTER);
43.         pageTitle.setText("Netflix");
44.         getContentPane().add(pageTitle, new org.netbeans.lib.awtextra.AbsoluteConstraints(80, 50, 1383, 68));
45.
46.         username.setFont(new java.awt.Font("Calibri Light", 0, 24)); // NOI18N
47.         username.setForeground(new java.awt.Color(229, 9, 20));
48.         getContentPane().add(username, new org.netbeans.lib.awtextra.AbsoluteConstraints(780, 330, 223, -1));
49.
50.         passwordLabel.setFont(new java.awt.Font("Bebas Neue", 0, 36)); // NOI18N
51.         passwordLabel.setForeground(new java.awt.Color(229, 9, 20));
52.         passwordLabel.setText("Password:");
53.         getContentPane().add(passwordLabel, new org.netbeans.lib.awtextra.AbsoluteConstraints(620, 400, -1,
34));
54.
55.         usernameLabel.setFont(new java.awt.Font("Bebas Neue", 0, 36)); // NOI18N
56.         usernameLabel.setForeground(new java.awt.Color(229, 9, 20));
57.         usernameLabel.setText("Username / Email:");
58.         getContentPane().add(usernameLabel, new org.netbeans.lib.awtextra.AbsoluteConstraints(530, 330, 218,
32));
59.
60.         login.setBackground(new java.awt.Color(229, 9, 20));
61.         login.setFont(new java.awt.Font("Bebas Neue", 0, 24)); // NOI18N
62.         login.setForeground(new java.awt.Color(255, 255, 255));
63.         login.setText("Login");
64.         login.addActionListener(new java.awt.event.ActionListener() {
65.             public void actionPerformed(java.awt.event.ActionEvent evt) {
66.                 loginActionPerformed(evt);
67.             }
68.         });
69.         getContentPane().add(login, new org.netbeans.lib.awtextra.AbsoluteConstraints(680, 470, 170, -1));
70.
```

```

71.     createAccount.setBackground(new java.awt.Color(255, 255, 255));
72.     createAccount.setFont(new java.awt.Font("Bebas Neue", 0, 24)); // NOI18N
73.     createAccount.setForeground(new java.awt.Color(229, 9, 20));
74.     createAccount.setText("Create Account");
75.     createAccount.addActionListener(new java.awt.event.ActionListener() {
76.         public void actionPerformed(java.awt.event.ActionEvent evt) {
77.             createAccountActionPerformed(evt);
78.         }
79.     });
80.     getContentPane().add(createAccount, new org.netbeans.lib.awtextra.AbsoluteConstraints(680, 530, 170,
-1));
81.
82.     password.setFont(new java.awt.Font("Calibri Light", 0, 24)); // NOI18N
83.     password.setForeground(new java.awt.Color(229, 9, 20));
84.     getContentPane().add(password, new org.netbeans.lib.awtextra.AbsoluteConstraints(780, 394, 223, 40));
85.
86.     jLabel1.setFont(new java.awt.Font("Bebas Neue", 0, 72)); // NOI18N
87.     jLabel1.setForeground(new java.awt.Color(229, 9, 20));
88.     jLabel1.setHorizontalAlignment(javax.swing.SwingConstants.CENTER);
89.     jLabel1.setText("Login Page");
90.     getContentPane().add(jLabel1, new org.netbeans.lib.awtextra.AbsoluteConstraints(90, 120, 1363, 67));
91.
92.     pack();
93. } // </editor-fold>
94.
95. //function applied when the login button is pressed by the user
96. //used to determine if the login was successful
97. private void loginActionPerformed(java.awt.event.ActionEvent evt) {
98.
99.     emailGlobal = username.getText(); //stores the email of the user currently logged in
100.    sqlConnection sC = new sqlConnection();
101.
102.    //compares the values entered to those found in the database
103.    if (sC.sqlLoginCheck(username.getText(), password.getText()) == true) {
104.        UIManager uiMan = new UIManager();
105.        uiMan.loginToSearch(); //shows the search page when login successful
106.        this.setVisible(false); //hides this window and shows the search when logged in
107.    }
108. }
109.
110. //shows the user the account creation page when the button is pressed
111. private void createAccountActionPerformed(java.awt.event.ActionEvent evt) {
112.     UIManager UIman = new UIManager();
113.     UIman.loginToCreateAccount();
114. }
115.
116. public static void main(String args[]) {
117.     /* Set the Nimbus look and feel */
118.     //<editor-fold defaultstate="collapsed" desc=" Look and feel setting code (optional) ">
119.     /* If Nimbus (introduced in Java SE 6) is not available, stay with the default look and feel.
120.      * For details see http://download.oracle.com/javase/tutorial/uiswing/lookandfeel/plaf.html
121.      */
122.     try {
123.         for (javax.swing.UIManager.LookAndFeelInfo info:
javax.swing.UIManager.getInstalledLookAndFeels()) {
124.             if ("Nimbus".equals(info.getName())) {
125.                 javax.swing.UIManager.setLookAndFeel(info.getClassName());
126.                 break;
127.             }
128.         }
129.     } catch (ClassNotFoundException ex) {

```

```

130.         java.util.logging.Logger.getLogger(loginPage.class.getName()).log(java.util.logging.Level.SEVERE,
131.             null, ex);
132.     } catch (InstantiationException ex) {
133.         java.util.logging.Logger.getLogger(loginPage.class.getName()).log(java.util.logging.Level.SEVERE,
134.             null, ex);
135.     } catch (IllegalAccessException ex) {
136.         java.util.logging.Logger.getLogger(loginPage.class.getName()).log(java.util.logging.Level.SEVERE,
137.             null, ex);
138.     } catch (javax.swing.UnsupportedLookAndFeelException ex) {
139.         java.util.logging.Logger.getLogger(loginPage.class.getName()).log(java.util.logging.Level.SEVERE,
140.             null, ex);
141.     }
142. }
143. // End of editor-fold
144.
145. /* Create and display the form */
146. java.awt.EventQueue.invokeLater(new Runnable() {
147.     public void run() {
148.         new loginPage().setVisible(true);
149.     }
150. });
151.
152. // Variables declaration - do not modify
153. private javax.swing.JButton createAccount;
154. private javax.swing.JLabel jLabel1;
155. private javax.swing.JButton login;
156. private javax.swing.JLabel pageTitle;
157. private javax.swing.JTextField password;
158. private javax.swing.JLabel passwordLabel;
159. private javax.swing.JTextField username;
160. private javax.swing.JLabel usernameLabel;
161. // End of variables declaration
162. }

```

Recommendation System Class:

```

1. package netflixpropstorage;
2.
3. import com.mysql.cj.jdbc.result.ResultSetMetaData;
4. import java.sql.ResultSet;
5. import java.sql.SQLException;
6. import java.util.LinkedList;
7. import java.util.logging.Level;
8. import java.util.logging.Logger;
9. import javax.swing.JOptionPane;
10.
11. public class recommendationSystem extends searchPage {
12.
13.     LinkedList < String > allPropList = new LinkedList < > ();
14.     LinkedList < String > userOrdersList = new LinkedList < > ();
15.     LinkedList < String > userOrdersList2 = new LinkedList < > ();
16.     LinkedList < String > propDetails1 = new LinkedList < > ();
17.     LinkedList < String > propDetails2 = new LinkedList < > ();
18.     LinkedList < String > propDetails3 = new LinkedList < > ();

```



```

19.    LinkedList < String > record = new LinkedList < > (); //the linked list which will store the results from
    the rs
20.    LinkedList < String > publicRecommendationList = new LinkedList < > (); //stores the results of the
    function in a linked list
21.    LinkedList < String > finallist = new LinkedList < > (); //stores the final set of values
22.    ResultSet resSet;
23.    boolean resultsFound = true;
24.    int temporary = 0;
25.
26.    //used to create a linked list which stores the ID of props which are most similar
27.    public LinkedList < String > generateRecc(String emailField) {
28.
29.        sqlConnection sC = new sqlConnection();
30.
31.        accessProps(emailField); //calls the access props function using the email found in parameter
32.        LinkedList < Integer > similarScores = new LinkedList < > (); //stores the similarity score of each
    prop
33.
34.        for (int i = 0; i < sC.numberOfProps() + 1; i++) { //this creates a linked list with the size equal
    to the number of total props
35.            similarScores.add(0); //fills the list with blank values which can be replaced
36.        }
37.
38.        compareValues(propDetails1, similarScores); //runs the programme for prop1
39.        compareValues(propDetails2, similarScores); //same but for prop 2
40.        compareValues(propDetails3, similarScores); //same but for prop 3
41.
42.        for (int i = 0; i < similarScores.size(); i++) {
43.            finallist.add(similarScores.get(i).toString()); //adds the values from similar scores to the
    final list
44.            String temp = finallist.get(i) + String.format("%03d", i); //adds the index to the end and uses 3
    digits
45.            finallist.set(i, temp); //sets each index to the value on the previous line which is the
    similarity + the index
46.        }
47.
48.        reverseList rL = new reverseList(finallist.size()); //creates an instance of the reverse list class
49.        rL.mergeSort(finallist, 0, finallist.size() - 1); //orders the final list using a merge sort based on
    the similarity values
50.        rL.reverse(finallist, finallist.size()); //the previous sort is smallest to largest so this reverses
    using stack
51.
52.        // finallist.get() means get that value from final list such as 2003
53.        // substring selects only the last 3 digits
54.        // replace first removes all leading zeros using regex
55.        System.out.println("this is the ID of the first most similar prop " +
    finallist.get(0).substring(finallist.get(0).length() - 3).replaceFirst("^0+(?!$)", ""));
56.        publicRecommendationList.add(finallist.get(0).substring(finallist.get(0).length() -
    3).replaceFirst("^0+(?!$)", ""));
57.
58.        System.out.println("this is the ID of the second most similar prop " +
    finallist.get(1).substring(finallist.get(0).length() - 3).replaceFirst("^0+(?!$)", ""));
59.        publicRecommendationList.add(finallist.get(1).substring(finallist.get(0).length() -
    3).replaceFirst("^0+(?!$)", ""));
60.
61.        System.out.println("this is the ID of the third most similar prop " +
    finallist.get(2).substring(finallist.get(0).length() - 3).replaceFirst("^0+(?!$)", ""));
62.        publicRecommendationList.add(finallist.get(2).substring(finallist.get(0).length() -
    3).replaceFirst("^0+(?!$)", ""));
63.
64.        this.setVisible(false);

```

```
65.         return publicRecommendationList;
66.     }
67.
68.     @Override //polymorphs the similar function found in searchPage but works for linked lists
69.     public void resultSetToArray(ResultSet resSet) {
70.         record.clear(); //empties the linked list
71.
72.         try {
73.
74.             ResultSetMetaData metaData = (ResultSetMetaData) resSet.getMetaData();
75.             int columns = metaData.getColumnCount(); //get meta data and store the number of columns
76.             resSet.first(); //moves the result set to the first line
77.
78.             //a loop which runs based on the number of columns to add the results to record
79.             for (int i = 1; i < columns + 1; i++) {
80.                 String value = resSet.getString(i);
81.                 record.add(value);
82.             }
83.
84.             while (resSet.next()) { //loops based on the number of results
85.
86.                 for (int i = 1; i < columns + 1; i++) {
87.                     String value = resSet.getString(i);
88.                     record.add(value);
89.                 }
90.             }
91.         } catch (SQLException ex) {
92.             Logger.getLogger(searchPage.class.getName()).log(Level.SEVERE, null, ex);
93.             record.add("null");
94.
95.         }
96.
97.     }
98.
99.     //stores the characteristics of the props which the user most recently hired
100.    public void accessProps(String email) {
101.
102.        try {
103.            sqlConnection sC = new sqlConnection();
104.            resSet = sC.emailToID(email); //stores the id of the user based on their email
105.            resSet.first(); //moves the current row of the result set to the first row
106.
107.            ResultSet orderSet = sC.searchOrders(resSet.getString("userID")); //contains the orders from
            userid
108.            orderSet.first(); //moves the current row of the result set to the first row
109.
110.            ResultSet rs = sC.searchAllProps();
111.            resultSetToArray(rs); //converts the result set to a linked list using the over ride function
112.            allPropList.addAll(0, record); //sets the allPropList to the value of record
113.
114.            resultSetToArray(sC.emailToOrders(email)); //converts the result set to a linked list using the
            over ride function
115.            userOrdersList2.addAll(0, record); //sets the allPropList to the value of record
116.
117.            if (userOrdersList2.get(0).equals("null")) {
118.
119.                JOptionPane.showMessageDialog(null, "This user does not have any prior hires");
120.            }
121.
122.            propDetails1.clear(); //creates a list of characteristics by getting the value from the list of
            props
```

```
123.         propDetails1.add(userOrdersList2.get(4));
124.         propDetails1.add(userOrdersList2.get(5));
125.         propDetails1.add(userOrdersList2.get(6));
126.         propDetails1.add(userOrdersList2.get(7));
127.         propDetails1.add(userOrdersList2.get(8));
128.
129.         propDetails2.clear();
130.         propDetails2.add(userOrdersList2.get(13));
131.         propDetails2.add(userOrdersList2.get(14));
132.         propDetails2.add(userOrdersList2.get(15));
133.         propDetails2.add(userOrdersList2.get(16));
134.         propDetails2.add(userOrdersList2.get(17));
135.
136.         propDetails3.clear();
137.         propDetails3.add(userOrdersList2.get(22));
138.         propDetails3.add(userOrdersList2.get(23));
139.         propDetails3.add(userOrdersList2.get(24));
140.         propDetails3.add(userOrdersList2.get(25));
141.         propDetails3.add(userOrdersList2.get(26));
142.
143.
144.     } catch (SQLException ex) {
145.         resultsFound = false;
146.
147.         Logger.getLogger(recommendationPage.class.getName()).log(Level.SEVERE, null, ex);
148.     }
149.
150. }
151.
152. //used to compare the similarities of the users hires compared to other props
153. public LinkedList < Integer > compareValues(LinkedList < String > propDetails, LinkedList < Integer >
    similarScores) {
154.     for (int i = 0; i < propDetails.size(); i++) {
155.
156.         for (int j = 0; j < allPropList.size(); j++) { //loops based on the number of props in the
            database
157.
158.             if (allPropList.get(j).equals(propDetails.get(i))) {
159.                 similarScores.set(Math.floorDiv(j, 5) + 1, -100); //makes similarity negative if
                comparing the same prop
160.             }
161.
162.             if (allPropList.get(j).equals(propDetails.get(i))) { //if the prop has the same
                characteristic it gains a point
163.                 temporary++;
164.                 similarScores.set(Math.floorDiv(j, 5) + 1, similarScores.get(Math.floorDiv(j, 5) + 1) +
                    1);
165.             }
166.         }
167.     }
168.
169.     if (temporary < 1) {
170.         resultsFound = false;
171.     }
172.     return similarScores;
173.
174. }
175.
176. public static void main(String args[]) {
177.     // TODO code application logic here
```

Reverse List Class:

```
1. package netflixpropstorage;
2.
3. import java.util.*;
4.
5. public class reverseList {
6.
7.     //stores maximum count of elements stored in a stack
8.     int size;
9.     //stores index of top element of a stack
10.    int top;
11.    //stores address of array element
12.    LinkedList < String > a = new LinkedList < > ();
13.
14.    //function to check if the stack is empty or not
15.    boolean isEmpty() {
16.        return (top < 0);
17.    }
18.
19.    //function to create a stack of given capacity.
20.    reverseList(int n) {
21.        top = -1;
22.        size = n;
23.        for (int i = 0; i < size; i++) {
24.            a.add(" ");
25.        }
26.
27.    }
28.
29.    //pushes an element onto the stack
30.    boolean push(String x) {
31.
32.        // If Stack is full
33.        if (top >= size) {
34.            System.out.println(
35.                "Stack Overflow");
36.            return false;
37.        } else {
38.
39.            // Insert element into stack
40.            a.set(++top, x);
41.            return true;
42.        }
43.    }
44.
45.    //function to remove an element from stack.
46.    String pop() {
47.        //check if stack is empty
48.        if (top < 0) {
49.            System.out.println(
50.                "Stack Underflow");
51.            return null;
52.        }
53.
54.        //pop element from stack
55.        else {
56.            String x = a.get(top--);
57.            return x;
58.        }
59.    }
60. }
```

```
59.     }
60.
61.     //takes the linked list and returns it reversed
62.     public static LinkedList < String > reverse(LinkedList < String > arr, int n) {
63.         LinkedList < String > temp = new LinkedList < > ();
64.
65.         reverseList obj = new reverseList(n); //initialize a stack of capacity n
66.
67.         for (int i = 0; i < n; i++) {
68.
69.             //add arr to the stack
70.             obj.push(arr.get(i));
71.         }
72.
73.         //reverse the array elements
74.         for (int i = 0; i < n; i++) {
75.
76.             // Update arr[i]
77.             //arr[i] = obj.pop();
78.             arr.set(i, obj.pop());
79.         }
80.
81.         for (int i = 0; i < n; i++) {
82.             temp.add(arr.get(i));
83.         }
84.         return temp;
85.     }
86.
87.     //used to apply the sort
88.     public static void mergeSort(LinkedList < String > list, int from, int to) {
89.         if (from == to) {
90.             return;
91.         }
92.         int mid = (from + to) / 2;
93.         // sort the first and the second half
94.         mergeSort(list, from, mid);
95.         mergeSort(list, mid + 1, to);
96.         merge(list, from, mid, to);
97.     }
98.
99.
100.    public static void merge(LinkedList < String > list, int from, int mid,
    int to) {
101.        int n = to - from + 1; // size of the range to be merged
102.        String[] b = new String[n]; // merge both halves into a temporary
    array b
103.        int i1 = from; // next element to consider in the first range
104.        int i2 = mid + 1; // next element to consider in the second range
105.        int j = 0; // next open position in b
106.
107.        // as long as neither i1 nor i2 past the end, move the smaller into b
108.        while (i1 <= mid && i2 <= to) {
109.            if (list.get(i1).compareTo(list.get(i2)) < 0) {
110.                b[j] = list.get(i1);
111.                i1++;
112.            } else {
113.                b[j] = list.get(i2);
114.                i2++;
115.            }
116.            j++;
117.        }
```

```
118.
119.         //only one of the two while loops below is executed copy any
        remaining entries of the first half
120.         while (i1 <= mid) {
121.             b[j] = list.get(i1);
122.             i1++;
123.             j++;
124.         }
125.
126.         // copy any remaining entries of the second half
127.         while (i2 <= to) {
128.             b[j] = list.get(i2);
129.             i2++;
130.             j++;
131.         }
132.
133.         // copy back from the temporary array
134.         for (j = 0; j < n; j++) {
135.             list.set(from + j, b[j]);
136.         }
137.     }
138.
139.     public static void main(String args[]) {
140.
141.     }
142. }
```

Search Page Class:

```
1.  package netflixpropstorage;
2.
3.  import com.mysql.cj.jdbc.result.ResultSetMetaData;
4.  import java.awt.Color;
5.  import java.awt.Component;
6.  import java.awt.Image;
7.  import java.sql.ResultSet;
8.  import java.sql.SQLException;
9.  import java.text.ParseException;
10. import java.util.ArrayList;
11. import java.util.logging.Level;
12. import java.util.logging.Logger;
13. import java.util.stream.Collectors;
14. import javax.swing.*;
15. import javax.swing.ImageIcon;
16. import javax.swing.table.DefaultTableModel;
17. import net.proteanit.sql.DbUtils;
18.
19.
20. public class searchPage extends JFrame {
21.
22.     int selectedRow;
23.     public String searchBoxValue = "null";
24.
25.     //set of public variables which store the characteristics of prop. used for generating the detail page
```

```
26. String col1, col2, col3, col4, col5, col6, col7, col8;
27.
28.
29. public searchPage() {
30.     getContentPane().setBackground(Color.WHITE);
31.     initComponents();
32.     this.setExtendedState(this.MAXIMIZED_BOTH);
33. }
34.
35.
36. @SuppressWarnings("unchecked")
37. // <editor-fold defaultstate="collapsed" desc="Generated Code">
38. private void initComponents() {
39.
40.     jLabel1 = new javax.swing.JLabel();
41.     logoutButton = new javax.swing.JButton();
42.     searchBox = new javax.swing.JTextField();
43.     searchButton = new javax.swing.JButton();
44.     detailButton = new javax.swing.JButton();
45.     detailEra2 = new javax.swing.JLabel();
46.     detailType2 = new javax.swing.JLabel();
47.     addPropButton = new javax.swing.JButton();
48.     jScrollPane1 = new javax.swing.JScrollPane();
49.     resultsTable = new javax.swing.JTable();
50.     basketButton = new javax.swing.JButton();
51.     recommendationButton = new javax.swing.JButton();
52.     ordersButton = new javax.swing.JButton();
53.     jLabel2 = new javax.swing.JLabel();
54.     viewBasketButton = new javax.swing.JButton();
55.     removeFromBasketButton = new javax.swing.JButton();
56.
57.     setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);
58.     setBackground(new java.awt.Color(255, 255, 255));
59.     getContentPane().setLayout(new org.netbeans.lib.awtextra.AbsoluteLayout());
60.
61.     jLabel1.setFont(new java.awt.Font("Bebas Neue", 0, 72)); // NOI18N
62.     jLabel1.setForeground(new java.awt.Color(229, 9, 20));
63.     jLabel1.setHorizontalAlignment(javax.swing.SwingConstants.CENTER);
64.     jLabel1.setText("Netflix");
65.     jLabel1.setHorizontalTextPosition(javax.swing.SwingConstants.CENTER);
66.     getContentPane().add(jLabel1, new org.netbeans.lib.awtextra.AbsoluteConstraints(10, 10, 1542, -1));
67.
68.     logoutButton.setBackground(new java.awt.Color(255, 255, 255));
69.     logoutButton.setFont(new java.awt.Font("Bebas Neue", 0, 24)); // NOI18N
70.     logoutButton.setText("Log Out");
71.     logoutButton.addActionListener(new java.awt.event.ActionListener() {
72.         public void actionPerformed(java.awt.event.ActionEvent evt) {
73.             logoutButtonActionPerformed(evt);
74.         }
75.     });
76.     getContentPane().add(logoutButton, new org.netbeans.lib.awtextra.AbsoluteConstraints(120, 250, 170,
77. 40));
78.
79.     searchBox.setFont(new java.awt.Font("Calibri", 0, 20)); // NOI18N
80.     searchBox.setForeground(new java.awt.Color(229, 9, 20));
81.     searchBox.setToolTipText("");
82.     searchBox.setCursor(new java.awt.Cursor(java.awt.Cursor.TEXT_CURSOR));
83.     getContentPane().add(searchBox, new org.netbeans.lib.awtextra.AbsoluteConstraints(340, 390, 410,
84. 30));
85.
86.     searchButton.setBackground(new java.awt.Color(229, 9, 20));
```

```

85.     searchButton.setFont(new java.awt.Font("Bebas Neue", 0, 24)); // NOI18N
86.     searchButton.setForeground(new java.awt.Color(255, 255, 255));
87.     searchButton.setText("search");
88.     searchButton.addActionListener(new java.awt.event.ActionListener() {
89.         public void actionPerformed(java.awt.event.ActionEvent evt) {
90.             searchButtonActionPerformed(evt);
91.         }
92.     });
93.     getContentPane().add(searchButton, new org.netbeans.lib.awtextra.AbsoluteConstraints(810, 390, 170,
94.     30));
95.     detailButton.setBackground(new java.awt.Color(229, 9, 20));
96.     detailButton.setFont(new java.awt.Font("Bebas Neue", 0, 22)); // NOI18N
97.     detailButton.setForeground(new java.awt.Color(255, 255, 255));
98.     detailButton.setText("Prop Details Page");
99.     detailButton.addActionListener(new java.awt.event.ActionListener() {
100.        public void actionPerformed(java.awt.event.ActionEvent evt) {
101.            detailButtonActionPerformed(evt);
102.        }
103.    });
104.    getContentPane().add(detailButton, new org.netbeans.lib.awtextra.AbsoluteConstraints(1040, 390, 170,
105.    30));
106.    getContentPane().add(detailEra2, new org.netbeans.lib.awtextra.AbsoluteConstraints(272, 619, 67, -
107.    1));
108.    getContentPane().add(detailType2, new org.netbeans.lib.awtextra.AbsoluteConstraints(272, 270, 67, -
109.    1));
110.    addPropButton.setBackground(new java.awt.Color(255, 255, 255));
111.    addPropButton.setFont(new java.awt.Font("Bebas Neue", 0, 24)); // NOI18N
112.    addPropButton.setForeground(new java.awt.Color(229, 9, 20));
113.    addPropButton.setText("Add Prop Page");
114.    addPropButton.addActionListener(new java.awt.event.ActionListener() {
115.        public void actionPerformed(java.awt.event.ActionEvent evt) {
116.            addPropButtonActionPerformed(evt);
117.        }
118.    });
119.    getContentPane().add(addPropButton, new org.netbeans.lib.awtextra.AbsoluteConstraints(810, 250, 170,
120.    -1));
121.    resultsTable.setModel(new javax.swing.table.DefaultTableModel(
122.        new Object[][] {
123.            {
124.                null,
125.                null,
126.                null,
127.                null
128.            }, {
129.                null,
130.                null,
131.                null,
132.                null
133.            }, {
134.                null,
135.                null,
136.                null,
137.                null
138.            }, {
139.                null,
140.                null,
141.                null,
142.                null
143.            }
144.        }, {
145.            "Era",
146.            "Type",
147.            "Prop",
148.            "Status"
149.        }
150.    ));

```



```
141.         }
142.     },
143.     new String[] {
144.         "Title 1",
145.         "Title 2",
146.         "Title 3",
147.         "Title 4"
148.     }
149. ));
150. jScrollPane1.setViewportViewView(resultsTable);
151.
152. getContentPane().add(jScrollPane1, new org.netbeans.lib.awtextra.AbsoluteConstraints(340, 470, 870,
256));
153.
154. basketButton.setBackground(new java.awt.Color(255, 255, 255));
155. basketButton.setFont(new java.awt.Font("Bebas Neue", 0, 24)); // NOI18N
156. basketButton.setForeground(new java.awt.Color(229, 9, 20));
157. basketButton.setText("Hire Basket");
158. basketButton.addActionListener(new java.awt.event.ActionListener() {
159.     public void actionPerformed(java.awt.event.ActionEvent evt) {
160.         basketButtonActionPerformed(evt);
161.     }
162. });
163. getContentPane().add(basketButton, new org.netbeans.lib.awtextra.AbsoluteConstraints(340, 250, 171, -
1));
164.
165. recommendationButton.setBackground(new java.awt.Color(255, 255, 255));
166. recommendationButton.setFont(new java.awt.Font("Bebas Neue", 0, 24)); // NOI18N
167. recommendationButton.setForeground(new java.awt.Color(229, 9, 20));
168. recommendationButton.setText("Reccomendations");
169. recommendationButton.addActionListener(new java.awt.event.ActionListener() {
170.     public void actionPerformed(java.awt.event.ActionEvent evt) {
171.         recommendationButtonActionPerformed(evt);
172.     }
173. });
174. getContentPane().add(recommendationButton, new org.netbeans.lib.awtextra.AbsoluteConstraints(580,
250, -1, -1));
175.
176. ordersButton.setBackground(new java.awt.Color(255, 255, 255));
177. ordersButton.setFont(new java.awt.Font("Bebas Neue", 0, 24)); // NOI18N
178. ordersButton.setForeground(new java.awt.Color(229, 9, 20));
179. ordersButton.setText("Orders");
180. ordersButton.addActionListener(new java.awt.event.ActionListener() {
181.     public void actionPerformed(java.awt.event.ActionEvent evt) {
182.         ordersButtonActionPerformed(evt);
183.     }
184. });
185. getContentPane().add(ordersButton, new org.netbeans.lib.awtextra.AbsoluteConstraints(1040, 250, 171,
-1));
186.
187. jLabel2.setFont(new java.awt.Font("Bebas Neue", 0, 72)); // NOI18N
188. jLabel2.setForeground(new java.awt.Color(229, 9, 20));
189. jLabel2.setHorizontalAlignment(javax.swing.SwingConstants.CENTER);
190. jLabel2.setText("Search Page");
191. jLabel2.setTextPosition(javax.swing.SwingConstants.CENTER);
192. getContentPane().add(jLabel2, new org.netbeans.lib.awtextra.AbsoluteConstraints(0, 100, 1542, -1));
193.
194. viewBasketButton.setBackground(new java.awt.Color(255, 255, 255));
195. viewBasketButton.setFont(new java.awt.Font("Bebas Neue", 0, 24)); // NOI18N
196. viewBasketButton.setForeground(new java.awt.Color(229, 9, 20));
197. viewBasketButton.setText("View basket");
```

```

198.         viewBasketButton.addActionListener(new java.awt.event.ActionListener() {
199.             public void actionPerformed(java.awt.event.ActionEvent evt) {
200.                 viewBasketButtonActionPerformed(evt);
201.             }
202.         });
203.         getContentPane().add(viewBasketButton, new org.netbeans.lib.awtextra.AbsoluteConstraints(340, 320,
170, 40));
204.
205.         removeFromBasketButton.setBackground(new java.awt.Color(255, 255, 255));
206.         removeFromBasketButton.setFont(new java.awt.Font("Bebas Neue", 0, 20)); // NOI18N
207.         removeFromBasketButton.setForeground(new java.awt.Color(229, 9, 20));
208.         removeFromBasketButton.setText("Remove from basket");
209.         removeFromBasketButton.addActionListener(new java.awt.event.ActionListener() {
210.             public void actionPerformed(java.awt.event.ActionEvent evt) {
211.                 removeFromBasketButtonActionPerformed(evt);
212.             }
213.         });
214.         getContentPane().add(removeFromBasketButton, new org.netbeans.lib.awtextra.AbsoluteConstraints(580,
320, 170, 40));
215.
216.         pack();
217.     } // </editor-fold>
218.
219.
220.
221.     ArrayList < String > record = new ArrayList < String > (); //the arraylist which will store the results
from the rs
222.
223.     //this func converts a result set into an array and then returns the arraylist
224.     public void resultSetToArray(ResultSet resSet) {
225.         record.clear(); //empties the arraylist
226.         try {
227.             ResultSetMetaData metaData = (ResultSetMetaData) resSet.getMetaData();
228.             int columns = metaData.getColumnCount(); //get meta data and store the number of columns
229.             while (resSet.next()) { //loops based on the number of results
230.
231.                 for (int i = 1; i < columns + 1; i++) {
232.                     String value = resSet.getString(i);
233.                     record.add(value);
234.                 }
235.             }
236.         } catch (SQLException ex) {
237.             Logger.getLogger(searchPage.class.getName()).log(Level.SEVERE, null, ex);
238.             record.add("null");
239.         }
240.     }
241.
242.
243. }
244.
245. //used to log the user out by returning to the login page
246. private void logoutButtonActionPerformed(java.awt.event.ActionEvent evt) {
247.     // TODO add your handling code here:
248.     UImanager uiMan = new UImanager();
249.     uiMan.searchToLogin();
250.     this.setVisible(false);
251. }
252.
253. public void searchFunction(String searchText) {
254.     sqlConnection sC = new sqlConnection();
255.     sC.sqlSearchProp(searchText); //applies prop search when button is pressed using the textbox value

```

```

256.         searchBoxValue = searchText;
257.
258.         ArrayList < String > result1 = new ArrayList < String > (); //used to store the values from search
259.         resultSetToArray(sC.getRs()); //populates result1 with the results
260.         result1 = record; //record is an array list which is declared at the start it stores the results
261.
262.         //this block overrides the getColumnClass so that the 3rd column is read as an image rather than file
        path
263.         DefaultTableModel tM = new DefaultTableModel() {
264.             @Override
265.             public Class getColumnClass(int column) {
266.                 if (column == 3) return ImageIcon.class;
267.                 else return Object.class;
268.             }
269.         };
270.
271.         //generates the columns for the table
272.         tM.addColumn("Prop ID");
273.         tM.addColumn("Prop Type");
274.         tM.addColumn("Prop Colour");
275.         tM.addColumn("Prop Image");
276.
277.         //checks if there were no results to return error message
278.         if (result1.isEmpty()) {
279.             JOptionPane.showMessageDialog(null, "this search returned no results");
280.         }
281.
282.         //block used to generate the table model which fill the results table
283.         for (int i = 0; i <= result1.size() / 8 - 1; i++) { //loops based on the number of results
284.             ImageIcon img = new ImageIcon(result1.get(i * 8 + 6)); // load the image to a ImageIcon
285.             Image image = img.getImage(); // transform it
286.             Image newimg = image.getScaledInstance(120, 120, java.awt.Image.SCALE_SMOOTH); // scale it the
        smooth way
287.             tM.addRow(new Object[] {
288.                 result1.get(i * 8), result1.get(i * 8 + 1), result1.get(i * 8 + 2), new ImageIcon(newimg)
289.             });
290.             // ^^ creates a new row by drawing values from the result1 array which contains results
291.         }
292.         resultsTable.setModel(tM); //set the model to the results table
293.
294.
295.         //iterates through the rows and sets the row height so that the images are not cropped when rendered
296.         for (int row = 0; row < resultsTable.getRowCount(); row++) {
297.             int rowHeight = resultsTable.getRowHeight();
298.
299.             for (int column = 0; column < resultsTable.getColumnCount(); column++) {
300.                 Component comp = resultsTable.prepareRenderer(resultsTable.getCellRenderer(row, column), row,
        column);
301.                 rowHeight = Math.max(rowHeight, comp.getPreferredSize().height);
302.             }
303.             resultsTable.setRowHeight(row, rowHeight);
304.
305.         }
306.     }
307.
308.     //used to search for props and create the results table
309.     private void searchButtonActionPerformed(java.awt.event.ActionEvent evt) {
310.
311.         searchFunction(searchBox.getText());
312.
313.     }

```

```

314.
315. //used to create a page which shows the details of the selected prop
316. private void detailButtonActionPerformed(java.awt.event.ActionEvent evt) {
317.
318.     if (resultsTable.getSelectionModel().isSelectionEmpty()) {
319.         JOptionPane.showMessageDialog(null, "There is no prop selected");
320.     } else {
321.
322.         sqlConnection sC = new sqlConnection();
323.         detailManager dM = new detailManager();
324.
325.         try {
326.
327.             String search = (String) resultsTable.getValueAt(resultsTable.getSelectedRow(), 0);
328.             //^^ gets the prop id from the generated table
329.
330.             ResultSet resSet = sC.detailSearch(search); // uses the prop id from above to sql search
331.             resSet.first(); //moves the result set current row to the first line
332.
333.             //sets the values of each characteristic which can then be passed into the detail page
generator
334.             col1 = resSet.getString("propID");
335.             col2 = resSet.getString("propType");
336.             col3 = resSet.getString("colour");
337.             col4 = resSet.getString("period");
338.             col5 = resSet.getString("propDate");
339.             col6 = resSet.getString("propDescription");
340.             col7 = resSet.getString("images");
341.             col8 = resSet.getString("availableToHire");
342.
343.             dM.generateDetailPage(col1, col2, col3, col4, col5, col6, col7, col8); //creates new page
with details
344.
345.         } catch (SQLException ex) {
346.             Logger.getLogger(searchPage.class.getName()).log(Level.SEVERE, null, ex);
347.         }
348.     }
349. }
350.
351. //displays the add prop page when pressed
352. private void addPropButtonActionPerformed(java.awt.event.ActionEvent evt) {
353.     UImanager UIman = new UImanager();
354.     UIman.searchToAddProp();
355.
356. }
357.
358.
359. //hires the props currently in the users basket
360. private void basketButtonActionPerformed(java.awt.event.ActionEvent evt) {
361.
362.     sqlConnection sC = new sqlConnection();
363.     String currentJob; //stores the users current production
364.     loginPage lP = new loginPage();
365.
366.     //prevents the user from using the hire function if there are no items in the basket
367.     if (lP.basketArray.isEmpty()) {
368.         JOptionPane.showMessageDialog(null, "The basket is empty");
369.     } else {
370.         try {
371.             ResultSet resSet = sC.emailToID(loginPage.emailGlobal); //stores the ID based on user email
372.             resSet.first(); //moves the result set to the first row

```

```

373.         currentJob = resSet.getString("job"); //stores the current production of the user
374.         String userID = resSet.getString("userID"); //stores their ID
375.
376.         String propID = Integer.toString(lP.basketArray.get(0)); //retrieves the propID and stores as
    string
377.         sC.addOrder(propID, userID, currentJob); //generates a new order using the information
    retrieved
378.
379.         //if the basket has more than one item in it the generate order process is repeated
380.         if (lP.basketArray.size() > 1) {
381.             for (int i = 1; i < lP.basketArray.size(); i++) {
382.                 propID = Integer.toString(lP.basketArray.get(i));
383.                 sC.addOrder(propID, userID, currentJob);
384.             }
385.         }
386.
387.
388.     } catch (SQLException ex) {
389.         Logger.getLogger(basketCheck.class.getName()).log(Level.SEVERE, null, ex);
390.     } catch (ParseException ex) {
391.         Logger.getLogger(basketCheck.class.getName()).log(Level.SEVERE, null, ex);
392.     }
393.     //informs the user once the hire process is complete
394.     JOptionPane.showMessageDialog(null, "The props in the basket have been hired");
395. }
396. }
397.
398. //displays the automatic recommendations to the user
399. private void recommendationButtonActionPerformed(java.awt.event.ActionEvent evt) {
400.
401.     recommendationPage rP = new recommendationPage();
402.     recommendationSystem rS = new recommendationSystem();
403.     loginPage lP = new loginPage();
404.
405.     //generates a list of recommendations of prop IDs using the users email
406.     String listString = String.join(" ", rS.generateRecc(lP.emailGlobal));
407.
408.     searchBox.setText(listString); //sets the value of the search box to the recommendations IDs
409.
410.     //this checks if there are any recommendations and informs the user
411.     //if the generated list starts with a null value then no recommendations were found
412.     if (rS.findAllList.get(0).charAt(0) == '0') {
413.         searchBox.setText("");
414.         JOptionPane.showMessageDialog(null, "There were no recommendations found\nCheck your email and
    orders");
415.
416.     } else {
417.         searchFunction(searchBox.getText()); //displays the recommended props with the search function
418.         JOptionPane.showMessageDialog(null, "The search results show the recommended items");
419.     }
420.
421. }
422.
423. //it shows the user all props currently hired and the details of when they will be returned
424. private void ordersButtonActionPerformed(java.awt.event.ActionEvent evt) {
425.     sqlConnection sC = new sqlConnection();
426.     JOptionPane.showMessageDialog(null, "The results table shows your orders");
427.     sC.getOrderDetails();
428.     resultsTable.setModel(DbUtils.resultSetToTableModel(sC.rs)); //sets the result table to the orders
429. }
430.

```

```
431. private void viewBasketButtonActionPerformed(java.awt.event.ActionEvent evt) {
432.     loginPage LP = new loginPage();
433.     //checks the length of the basket array
434.     if (LP.basketArray.isEmpty()) {
435.         JOptionPane.showMessageDialog(null, "There are currently no items in your basket");
436.     } else {
437.         //converts from arraylist to string so that the values can be used in the search box
438.         String listString = LP.basketArray.stream().map(Object::toString).collect(Collectors.joining(",
439.         "));
440.         searchBox.setText(listString); //sets the value of the search box to the recommendations IDs
441.         searchFunction(searchBox.getText()); //displays the recommended props with the search function
442.         JOptionPane.showMessageDialog(null, "The results table shows your current basket");
443.     }
444. }
445. private void removeFromBasketButtonActionPerformed(java.awt.event.ActionEvent evt) {
446.     loginPage LP = new loginPage();
447.     //checks if the user has selected a prop to remove from the basket
448.     if (resultsTable.getSelectionModel().isSelectionEmpty() == true) {
449.         JOptionPane.showMessageDialog(null, "There is no prop selected");
450.     } else {
451.         //stores the value which the user has selected in the results table
452.         String selection = (String) resultsTable.getValueAt(resultsTable.getSelectedRow(), 0);
453.
454.         //checks if the item selected is in the basket
455.         if (LP.basketArray.indexOf(Integer.parseInt(selection)) == -1) {
456.             JOptionPane.showMessageDialog(null, "This item cannot be found in your basket");
457.         } else {
458.             //if all inputs are correct the item is removed from the basket
459.             LP.basketArray.remove(LP.basketArray.indexOf(Integer.parseInt(selection)));
460.             JOptionPane.showMessageDialog(null, "This prop has been removed");
461.         }
462.     }
463. }
464.
465. public String getSearchBoxValue() {
466.     return searchBoxValue;
467. }
468.
469. }
470.
471. public int getResultsTable() {
472.     return selectedRow;
473. }
474.
475. public static void main(String args[]) {
476.     /* Set the Nimbus look and feel */
477.     //<editor-fold defaultstate="collapsed" desc=" Look and feel setting code (optional) ">
478.     /* If Nimbus (introduced in Java SE 6) is not available, stay with the default look and feel.
479.      * For details see http://download.oracle.com/javase/tutorial/uiswing/lookandfeel/plaf.html
480.      */
481.     try {
482.         for (javax.swing.UIManager.LookAndFeelInfo info:
483.             javax.swing.UIManager.getInstalledLookAndFeels()) {
484.             if ("Nimbus".equals(info.getName())) {
485.                 javax.swing.UIManager.setLookAndFeel(info.getClassName());
486.                 break;
487.             }
488.         } catch (ClassNotFoundException ex) {
```

```

489.         java.util.logging.Logger.getLogger(searchPage.class.getName()).log(java.util.logging.Level.SEVERE
, null, ex);
490.     } catch (InstantiationException ex) {
491.         java.util.logging.Logger.getLogger(searchPage.class.getName()).log(java.util.logging.Level.SEVERE
, null, ex);
492.     } catch (IllegalAccessException ex) {
493.         java.util.logging.Logger.getLogger(searchPage.class.getName()).log(java.util.logging.Level.SEVERE
, null, ex);
494.     } catch (javax.swing.UnsupportedLookAndFeelException ex) {
495.         java.util.logging.Logger.getLogger(searchPage.class.getName()).log(java.util.logging.Level.SEVERE
, null, ex);
496.     }
497.     </editor-fold>
498.
499.     /* Create and display the form */
500.     java.awt.EventQueue.invokeLater(new Runnable() {
501.         public void run() {
502.             new searchPage().setVisible(true);
503.         }
504.     });
505. }
506.
507. // Variables declaration - do not modify
508. private javax.swing.JButton addPropButton;
509. private javax.swing.JButton basketButton;
510. private javax.swing.JButton detailButton;
511. private javax.swing.JLabel detailEra2;
512. private javax.swing.JLabel detailType2;
513. private javax.swing.JLabel jLabel1;
514. private javax.swing.JLabel jLabel2;
515. private javax.swing.JScrollPane jScrollPane1;
516. private javax.swing.JButton logoutButton;
517. private javax.swing.JButton ordersButton;
518. private javax.swing.JButton recommendationButton;
519. private javax.swing.JButton removeFromBasketButton;
520. private javax.swing.JTable resultsTable;
521. public javax.swing.JTextField searchBox;
522. private javax.swing.JButton searchButton;
523. private javax.swing.JButton viewBasketButton;
524. // End of variables declaration
525. }

```

SQL Connection Class:

```

1. package netflixpropstorage;
2.
3. import java.sql.Connection;
4. import java.sql.DriverManager;
5. import java.sql.ResultSet;
6. import java.sql.SQLException;
7. import java.sql.Statement;
8. import java.text.ParseException;
9. import java.text.SimpleDateFormat;
10. import java.time.LocalDate;
11. import java.time.format.DateTimeFormatter;
12. import java.util.Calendar;
13. import javax.swing.ImageIcon;
14. import javax.swing.JOptionPane;
15.

```

```
16. public class sqlConnection {
17.     ResultSet rs;
18.     ImageIcon icon;
19.
20.
21.
22.     public ResultSet getRs() {
23.         return rs;
24.     }
25.
26.     public ImageIcon getIcon() {
27.         return icon;
28.     }
29.
30.     //used to check the validity of login credentials
31.     public boolean sqlLoginCheck(String email, String password) {
32.         boolean login = false;
33.         hashFunction hF = new hashFunction();
34.         password = hF.hashFunction(password);
35.
36.         try {
37.             Class.forName("com.mysql.cj.jdbc.Driver");
38.             Connection connection =
DriverManager.getConnection("jdbc:mysql://localhost:3306/propdb", "root",
"rocket08");
39.             //Statement statement = connection.createStatement();
40.             Statement statement =
connection.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_READ_ONLY);
41.             String customQuery = "select * from employees where email = '" + email +
""";
42.             ResultSet resultSet = statement.executeQuery(customQuery);
43.
44.             while (resultSet.next()) {
45.                 if (password.equals(resultSet.getString("password"))) {
46.                     JOptionPane.showMessageDialog(null, "The login details were
correct");
47.                     login = true;
48.                 } else {
49.                     JOptionPane.showMessageDialog(null, "The login details were
incorrect");
50.                     login = false;
51.                 }
52.
53.             }
54.         } catch (SQLException e) {
55.             System.out.println("SQL Exception: " + e.toString());
56.         } catch (ClassNotFoundException cE) {
57.             System.out.println("Class Not Found Exception: " + cE.toString());
58.         }
59.         return login;
60.     }
61.
62.     //returns all the characteristics of the selected props based on a characteristic
63.     public void sqlSearchProp(String prop) {
64.
65.         //checks if multiple words have been entered
66.         if (prop.contains(" ")) {
67.
68.             String[] words = prop.split(" "); //creates a list of words from the
search values
```



```
69.
70.         try {
71.             Class.forName("com.mysql.cj.jdbc.Driver");
72.             Connection connection =
DriverManager.getConnection("jdbc:mysql://localhost:3306/propdb", "root",
"rocket08");
73.             //Statement statement = connection.createStatement();
74.             Statement statement =
connection.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_READ_ONLY);
75.             String customQuery = "SELECT * FROM props WHERE'" + words[0] +
"'IN(propID, propType, colour, period, propDate, propDescription)";
76.
77.             //runs a for loop based on the number of words entered which creates
a statement with all the terms
78.             for (int i = 0; i < words.length; i++) {
79.                 customQuery = customQuery.concat("OR'" + words[i] + "'IN(propID,
propType, colour, period, propDate, propDescription)");
80.             }
81.
82.             //the result set which stores the result from the search
83.             rs = statement.executeQuery(customQuery);
84.
85.         } catch (SQLException e) {
86.             System.out.println("SQL Exception: " + e.toString());
87.         } catch (ClassNotFoundException cE) {
88.             System.out.println("Class Not Found Exception: " + cE.toString());
89.         }
90.     }
91.
92.     //else block runs when there is only one word entered
93.     else {
94.         try {
95.             Class.forName("com.mysql.cj.jdbc.Driver");
96.             Connection connection =
DriverManager.getConnection("jdbc:mysql://localhost:3306/propdb", "root",
"rocket08");
97.             //Statement statement = connection.createStatement();
98.             Statement statement =
connection.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_READ_ONLY);
99.             String customQuery = "SELECT * FROM props WHERE'" + prop +
"'IN(propID, propType, colour, period, propDate, propDescription)";
100.             rs = statement.executeQuery(customQuery);
101.
102.
103.         } catch (SQLException e) {
104.             System.out.println("SQL Exception: " + e.toString());
105.         } catch (ClassNotFoundException cE) {
106.             System.out.println("Class Not Found Exception: " +
cE.toString());
107.         }
108.     }
109.
110. }
111.
112.     //creates a new row in the account table using the characteristics
entered as parameters
113.     public static void accountCreation(String firstName, String surname,
String email, String phone, String jobRole, String job, String password) {
114.
```

```
115.         try {
116.             Class.forName("com.mysql.cj.jdbc.Driver");
117.             Connection connection =
                DriverManager.getConnection("jdbc:mysql://localhost:3306/propdb", "root",
                "rocket08");
118.             Statement statement =
                connection.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                ResultSet.CONCUR_READ_ONLY);
119.             String customQuery = "INSERT INTO employees(firstName , surname,
                email, phone, jobRole, job, password) VALUES('" + firstName + "' , '" + surname +
                "'" + email + "' ,'" + phone + "' ,'" + jobRole + "' ,'" + job + "' ,'" + password +
                "'"");";
120.             statement.executeUpdate(customQuery);
121.         } catch (SQLException e) {
122.             System.out.println("SQL Exception: " + e.toString());
123.         } catch (ClassNotFoundException cE) {
124.             System.out.println("Class Not Found Exception: " +
                cE.toString());
125.         }
126.
127.     }
128.
129.     //creates a new row in the prop table using the characteristics entered
    as parameters
130.     public static void addProp(String propType, String colour, String period,
        String propDate, String propDescription, String images) {
131.
132.         try {
133.             Class.forName("com.mysql.cj.jdbc.Driver");
134.             Connection connection =
                DriverManager.getConnection("jdbc:mysql://localhost:3306/propdb", "root",
                "rocket08");
135.             //Statement statement = connection.createStatement();
136.             Statement statement =
                connection.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                ResultSet.CONCUR_READ_ONLY);
137.             String customQuery = "INSERT INTO props(propType , colour,
                period, propDate, propDescription, images) VALUES('" + propType + "' , '" + colour +
                "'" + period + "' ,'" + propDate + "' ,'" + propDescription + "' ,'" + images + "'");";
138.             System.out.println("this the query: " + customQuery);
139.             statement.executeUpdate(customQuery);
140.
141.         } catch (SQLException e) {
142.             System.out.println("SQL Exception: " + e.toString());
143.         } catch (ClassNotFoundException cE) {
144.             System.out.println("Class Not Found Exception: " +
                cE.toString());
145.         }
146.
147.     }
148.
149.     //returns all the characteristics of the selected props based on an
    characteristic but returns it as a result set
150.     public ResultSet detailSearch(String prop) {
151.         try {
152.             Class.forName("com.mysql.cj.jdbc.Driver");
153.             Connection connection =
                DriverManager.getConnection("jdbc:mysql://localhost:3306/propdb", "root",
                "rocket08");
```

```
154.         Statement statement =
        connection.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
        ResultSet.CONCUR_READ_ONLY);
155.         String customQuery = "SELECT * FROM props WHERE'" + prop +
        "'IN(propID, propType, colour, period, propDate, propDescription)";
156.         rs = statement.executeQuery(customQuery);
157.
158.
159.     } catch (SQLException e) {
160.         System.out.println("SQL Exception: " + e.toString());
161.     } catch (ClassNotFoundException cE) {
162.         System.out.println("Class Not Found Exception: " +
        cE.toString());
163.     }
164.
165.     return rs;
166.
167. }
168.
169. //returns the userID based on the email entered
170. public ResultSet emailToID(String email) {
171.
172.     try {
173.         Class.forName("com.mysql.cj.jdbc.Driver");
174.         Connection connection =
        DriverManager.getConnection("jdbc:mysql://localhost:3306/propdb", "root",
        "rocket08");
175.         Statement statement =
        connection.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
        ResultSet.CONCUR_READ_ONLY);
176.         String customQuery = "SELECT userID, job FROM propdb.employees
        where email = '" + email + "'";
177.         rs = statement.executeQuery(customQuery);
178.
179.
180.     } catch (SQLException e) {
181.         System.out.println("SQL Exception: " + e.toString());
182.     } catch (ClassNotFoundException cE) {
183.         System.out.println("Class Not Found Exception: " +
        cE.toString());
184.     }
185.     return rs;
186. }
187.
188. //adds a new row to the order table
189. public static void addOrder(String propID, String userID, String shootID)
        throws ParseException {
190.
191.     try {
192.         Class.forName("com.mysql.cj.jdbc.Driver");
193.         Connection connection =
        DriverManager.getConnection("jdbc:mysql://localhost:3306/propdb", "root",
        "rocket08");
194.         Statement statement =
        connection.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
        ResultSet.CONCUR_READ_ONLY);
195.
196.         //used to add a new row to the orders table using the appropriate
        values
197.         String customQuery = "INSERT INTO orders(propID, userID, shootID)
        VALUES('" + propID + "', '" + userID + "', '" + shootID + "')";
```

```

198.         statement.executeUpdate(customQuery);
199.
200.         //changes the availability of the prop so that it cannot be hired
        again until it is returned
201.         String customQuery2 = "UPDATE props SET availableToHire = FALSE
        WHERE propID = '" + propID + "'";
202.         statement.executeUpdate(customQuery2);
203.
204.         //stores the maximum orderID and stores it in an sql variable
205.         String customQuery3 = "set @orderId =(SELECT orderID FROM
        propdb.orders WHERE orderID=(SELECT max(orderID) FROM propdb.orders)); ";
206.         statement.executeUpdate(customQuery3);
207.
208.         //stores the current job of the user hiring the props
209.         String customQuery5 = "set @job =(SELECT job FROM
        propdb.employees WHERE employees.userID =' " + userID + "'";
210.         statement.executeUpdate(customQuery5);
211.
212.         //retrieves and stores the shoot id from the shoot location based
        on the users current job
213.         String customQuery6 = "set @shootId =(SELECT shootID FROM
        propdb.shootlocation WHERE shootlocation.production = @job);";
214.         statement.executeUpdate(customQuery6);
215.
216.         //gets the number of days based on the shoot id
217.         String customQuery7 = "select durationDays from shootlocation
        where shootId = @shootId";
218.
219.         //stores the duration of the shoot for the current user
220.         ResultSet durationDays;
221.         durationDays = statement.executeQuery(customQuery7);
222.         durationDays.first();
223.
224.         //sets the format of the dates and creates a new date object
225.         DateTimeFormatter dtf =
        DateTimeFormatter.ofPattern("yyyy/MM/dd");
226.
227.         //gets the local date
228.         LocalDateTime now = LocalDateTime.now();
229.
230.         //stores the current date as a string based on the previously set
        format
231.         String hireDate = dtf.format(now);
232.
233.         String returnDate = hireDate; //stores the date which the prop
        will be returned on
234.         SimpleDateFormat sdf = new SimpleDateFormat("yyyy/MM/dd");
235.         Calendar c = Calendar.getInstance();
236.         c.setTime(sdf.parse(returnDate)); //formats and sets the return
        date
237.
238.         //adds to the return date based on the duration of days for the
        shoot
239.         c.add(Calendar.DATE,
        Integer.parseInt(durationDays.getString("durationDays")));
240.         returnDate = sdf.format(c.getTime()); // dt is now the new date
241.
242.         //stores the information about this prop hire in the order detail
        table

```

```
243.         String customQuery4 = "INSERT INTO orderDetail(hireDate,
returnDate, quantity, orderID, shootID) VALUES('\" + hireDate + "\", '\" + returnDate +
\", '1', \" + \" @orderId , @shootId);";
244.         statement.executeUpdate(customQuery4);
245.
246.     } catch (SQLException e) {
247.         System.out.println("SQL Exception: \" + e.toString());
248.     } catch (ClassNotFoundException cE) {
249.         System.out.println("Class Not Found Exception: \" +
cE.toString());
250.     }
251.
252.     }
253.
254.     //changes the availability of a prop from false to true
255.     public static void returnProp(String propID) {
256.
257.         try {
258.             Class.forName("com.mysql.cj.jdbc.Driver");
259.             Connection connection =
DriverManager.getConnection("jdbc:mysql://localhost:3306/propdb", "root",
"rocket08");
260.             Statement statement =
connection.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_READ_ONLY);
261.
262.             String customQuery2 = "UPDATE props SET availableToHire = TRUE
WHERE propID = '\" + propID + \"';";
263.             statement.executeUpdate(customQuery2);
264.
265.         } catch (SQLException e) {
266.             System.out.println("SQL Exception: \" + e.toString());
267.         } catch (ClassNotFoundException cE) {
268.             System.out.println("Class Not Found Exception: \" +
cE.toString());
269.         }
270.
271.     }
272.
273.     //selects all the orders from a specific user
274.     public ResultSet searchOrders(String userID) {
275.
276.         try {
277.             Class.forName("com.mysql.cj.jdbc.Driver");
278.             Connection connection =
DriverManager.getConnection("jdbc:mysql://localhost:3306/propdb", "root",
"rocket08");
279.             Statement statement =
connection.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_READ_ONLY);
280.
281.             String customQuery = "SELECT * FROM orders WHERE userID = '\" +
userID + \"'";
282.             statement.executeQuery(customQuery);
283.             rs = statement.executeQuery(customQuery);
284.
285.         } catch (SQLException e) {
286.             System.out.println("SQL Exception: \" + e.toString());
287.         } catch (ClassNotFoundException cE) {
288.             System.out.println("Class Not Found Exception: \" +
cE.toString());
```

```
289.         }
290.         return rs;
291.     }
292.
293.     //determines the number of props currently available in the database
294.     int numberOfResults;
295.     public int numberOfProps() {
296.
297.         try {
298.             Class.forName("com.mysql.cj.jdbc.Driver");
299.             Connection connection =
                DriverManager.getConnection("jdbc:mysql://localhost:3306/propdb", "root",
                "rocket08");
300.             Statement statement =
                connection.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                ResultSet.CONCUR_READ_ONLY);
301.
302.             //returns the single value of how many props there are currently
303.             String customQuery = "SELECT COUNT(*) FROM props";
304.             statement.executeQuery(customQuery);
305.             rs = statement.executeQuery(customQuery);
306.
307.             rs.first();
308.             numberOfResults = rs.getInt(1); //sets the number of results to
                the value calculated
309.         } catch (SQLException e) {
310.             System.out.println("SQL Exception: " + e.toString());
311.         } catch (ClassNotFoundException cE) {
312.             System.out.println("Class Not Found Exception: " +
                cE.toString());
313.         }
314.         return numberOfResults;
315.     }
316.
317.     public ResultSet searchAllProps() {
318.
319.         try {
320.             Class.forName("com.mysql.cj.jdbc.Driver");
321.             Connection connection =
                DriverManager.getConnection("jdbc:mysql://localhost:3306/propdb", "root",
                "rocket08");
322.             Statement statement =
                connection.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                ResultSet.CONCUR_READ_ONLY);
323.
324.             String customQuery = "SELECT propID, propType, colour, period,
                propDate FROM props;";
325.             statement.executeQuery(customQuery);
326.             rs = statement.executeQuery(customQuery);
327.
328.         } catch (SQLException e) {
329.             System.out.println("SQL Exception: " + e.toString());
330.         } catch (ClassNotFoundException cE) {
331.             System.out.println("Class Not Found Exception: " +
                cE.toString());
332.         }
333.         return rs;
334.     }
335.
336.     //takes the user email as a parameter and returns their orders
337.     public ResultSet emailToOrders(String email) {
```

```
338.
339.         try {
340.             Class.forName("com.mysql.cj.jdbc.Driver");
341.             Connection connection =
DriverManager.getConnection("jdbc:mysql://localhost:3306/propdb", "root",
"rocket08");
342.             Statement statement =
connection.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_READ_ONLY);
343.
344.             //stores the employee id by converting from the email to the id
345.             String customUpdate = "set @EmpID = (select userID from
propdb.employees where employees.email = ' " + email + " ' );";
346.             statement.executeUpdate(customUpdate);
347.
348.             //selects all the orders based on the previously stored employee
id
349.             String customQuery = "SELECT Orders.*, props.propID,
props.propType, props.colour, props.period, props.propDate FROM Orders INNER JOIN
props ON orders.propID=props.propID where orders.userID = @EmpID order by orderID
DESC;";
350.             statement.executeQuery(customQuery);
351.             rs = statement.executeQuery(customQuery);
352.
353.         } catch (SQLException e) {
354.             System.out.println("SQL Exception: " + e.toString());
355.         } catch (ClassNotFoundException cE) {
356.             System.out.println("Class Not Found Exception: " +
cE.toString());
357.         }
358.         return rs;
359.     }
360.
361.     public ResultSet getOrderDetails() {
362.
363.         try {
364.             Class.forName("com.mysql.cj.jdbc.Driver");
365.             Connection connection =
DriverManager.getConnection("jdbc:mysql://localhost:3306/propdb", "root",
"rocket08");
366.             Statement statement =
connection.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_READ_ONLY);
367.
368.             String customQuery = "SELECT *\n" +
369.                 "FROM orderdetail\n" +
370.                 "INNER JOIN shootlocation ON orderdetail.shootID=
shootlocation.shootID;";
371.             statement.executeQuery(customQuery);
372.             rs = statement.executeQuery(customQuery);
373.
374.         } catch (SQLException e) {
375.             System.out.println("SQL Exception: " + e.toString());
376.         } catch (ClassNotFoundException cE) {
377.             System.out.println("Class Not Found Exception: " +
cE.toString());
378.         }
379.         return rs;
380.     }
```

CANDIDATE NUMBER: 6437

CENTRE: 12460

MATTHEW COOK