**Coursework 2 – Programming 1**

Abdullah hashmat azeez 33400679

University of southampton (Malaysia campus)

15 DEC 2023

2023

Table of content

|  |  |  |
| --- | --- | --- |
| Section | Content | Page Number |
| 1 | Planning | 3 |
| 2 | Coding | 5 |
| 3 | Reflection | 15 |

1. **Planning**

To begin the planning, I wanted to work on a mistake I made in the coursework 1 for the same module. The mistake being that I did not do any complete programming before I started coding. So, for this coursework, the first thing I did was construct a class diagram and attempt a thorough plan before I started coding:

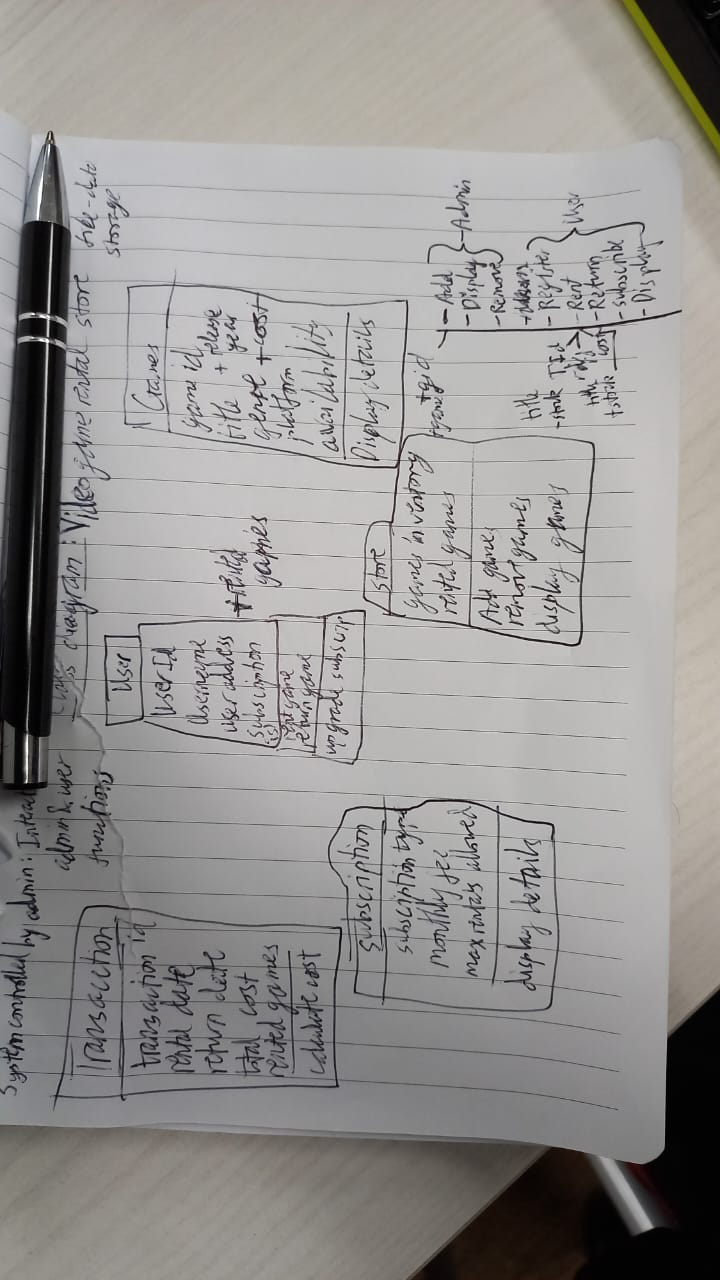
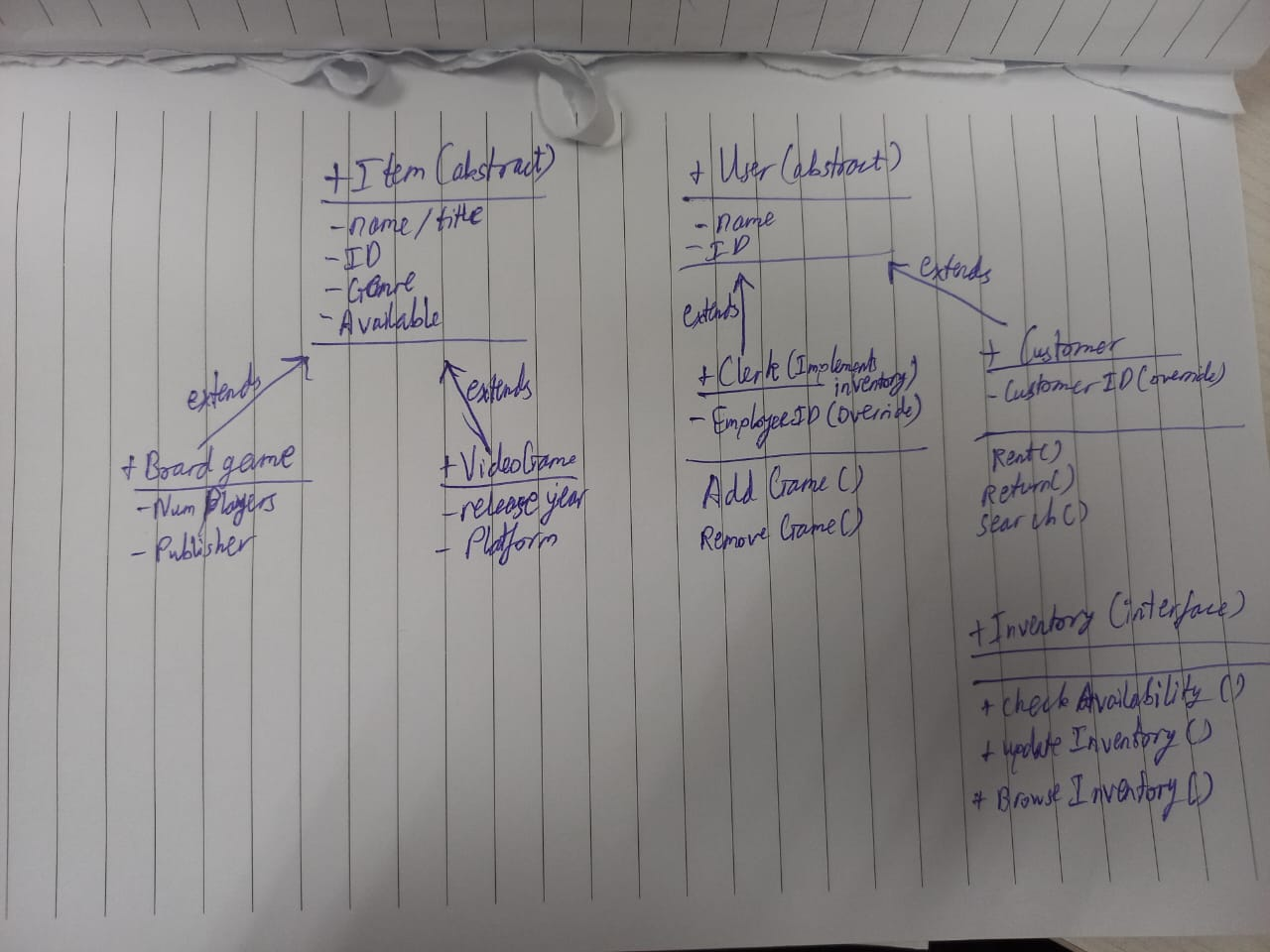
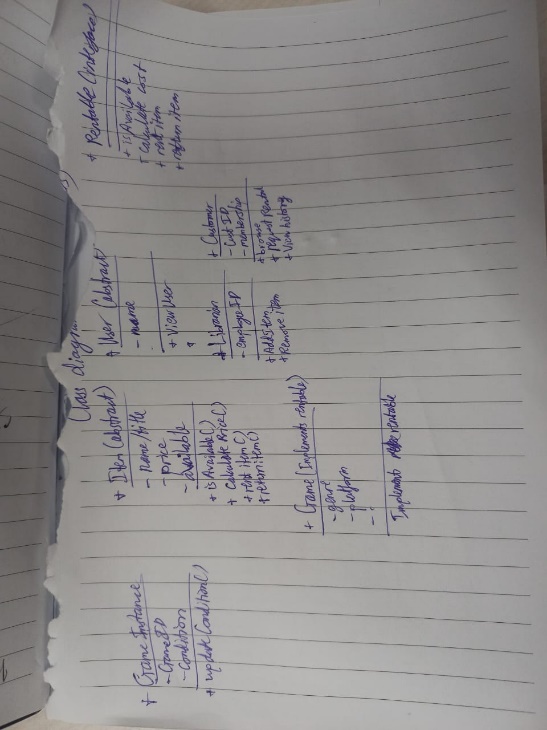
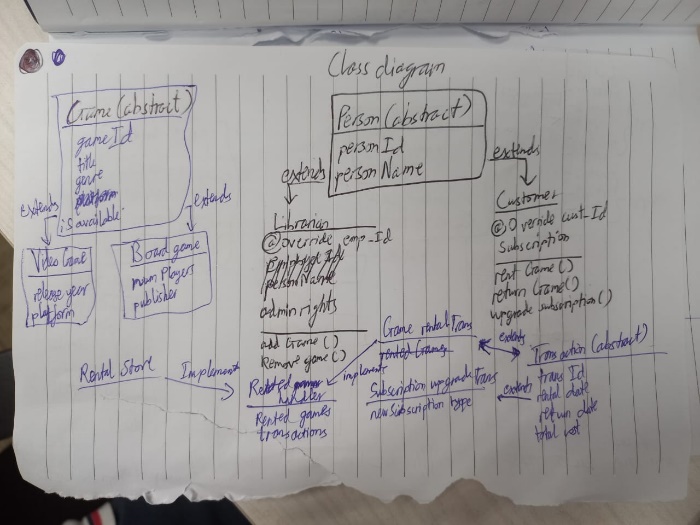


Image 1: A very crude first attempt at my class diagram

My general plan was that I wanted to make a game rental system that was like a library management system. My only user is the game clerk (librarian), who has access to administrative features of the system that enables the clerk to manipulate and manage stock for example. The clerk also has access to the user management, allowing them to register new users and assist them in renting and returning games. My goal is to create a somewhat practical system for this scenario. From here I add additional features such as the inclusion of a transaction system and a subscription system.

This is when I ran into my first problem, the draft had no inclusion whatsoever of inheritance, interface, or abstraction. I decided to make another draft of my class diagram to include these, and so I created multiple drafts until my final:

Image 2-4: A the draft and final class diagram

In the drafts shown above, I went through multiple thought processes and approaches to the program. In the end, I ended up simplifying the diagram a lot and sticking only to the foundation and base functionalities of the program. I redefined the purpose above as well. My new system will allow interaction with admins and users and will be able to distinguish between them (like a kiosk). Only keeping Item and User classes with their children. Later, after establishing this foundation in code, I intend to add the rest of the features slowly. These features include:

* Proper cost system
* A subscription attribute to allow the user to rent free of charge
  + Maybe add tiers to the subscription
* Maybe add items that can be sold instead of only renting (like merchandise)
* Stock system instead of available only
* A file read and write system implemented at least once somewhere
* Other improvements and additions to the extra features.

1. **Coding**

* After settling on a final draft for my class diagrams for the coursework, I began drafting the classes for the coursework.
* I started with the abstract classes of user and item and defining their general attributes, I then made the constructors, setters, and getters for the attributes.
  + I only made getters for the user class since I decided that it wouldn’t be possible to change the ID or name of a user since those things are fixed.
    - This might be subject to change later and I might add it as a feature to the program later
* I also made a function for these classes since I decided that their ID’s would be generated randomly. I also created abstract methods to be overridden for login, register, etc.
* User.java
* import java.util.Random;  
    
  public abstract class *user* {  
   *// attributes* private String **name**;  
   private String **ID**;  
    
   *//constructor* public user(String *name*) {  
   this.**name** = *name*;  
   this.**ID** = *generateRandomID*();  
   }  
    
   *//getters* public String getName() {  
   return **name**;  
   }  
    
   public String getID() {  
   return **ID**;  
   }  
    
   *//abstract methods* public abstract boolean login();  
    
   public abstract void register();  
    
   public abstract void forgetID();  
    
   *// generate randomID method* protected static String generateRandomID() {  
   *// Generate a random alphanumeric ID for the user* Random rand = new Random();  
   int num = rand.nextInt(10000);  
   return String.*format*("%04d", num);  
   }  
  }
* I did have a little bit of a hiccup in this part. I was at a dilemma of whether I should make the ID a string or an integer. So, I decided to do both, the users would have randomly generated string IDs and the games would have randomly calculated integer IDs.
* Item.java
* public abstract class *Item* {  
   *// attributes* private String **title**;  
   private int **ID**;  
   private String **genre**;  
   private boolean **available**;  
    
   *// item constructor* public Item(String *title*, String *genre*) {  
   this.**title** = *title*;  
   *// ID does not need to be defined in the constructor and will be displayed in the display function* this.**ID** = *generateRandomID*();  
   this.**genre** = *genre*;  
   this.**available** = true;  
   }  
    
   *// getters and setters* public String getTitle() {  
   return **title**;  
   }  
    
   public int getID() {  
   return **ID**;  
   }  
    
   public String getGenre() {  
   return **genre**;  
   }  
    
   public boolean isAvailable() {  
   return **available**;  
   }  
    
   public void setAvailable(boolean *available*) {  
   this.**available** = *available*;  
   }  
    
   *// method to generate a random ID* public static int generateRandomID() {  
   return (int) (Math.*random*() \* 1000);  
   }  
    
   *// method for output print formatting for display function* public String toString() {  
   return String.*format*("ID: %s | Title: %s | Genre: %s | Available: %s",  
   this.getID(), this.getTitle(), this.getGenre(), this.isAvailable());  
   }  
  }
* My item class has an extra function that is meant assist with the formatting when printing the output for the objects. I did this because each object will have their shared properties like title and then some objects like the different types of games will have their unique attributes like ‘platform’ for videoGames. Therefore, I saw the opportunity to use the override feature here for this toString method.
* After making my abstract parent classes, I started to draft my child classes.
* I started with the clerk class and made some decisions
  + The clerk is a singular person and there will be only ONE clerk
  + Therefore, a password will exist created by the clerk’s boss and cannot be changed. Thus, the password is pre-defined in the code and cannot be manipulated.
  + I had overridden the login function from the abstract class to match the pre-defined password with the input. I also did an override for the other abstract functions but due to my scenario, a clerk will not register nor forget their ID. Thus, these ended up being dummy functions I used for fun.
  + I did also have a randomly generated employee function, but I am still not sure if I will use it for validation or something.
  + My first issue came with some error in the code. I remember running into a similar issue in coursework 1. The solution was using a scanner.nextLine(); after a nextInt(); prompt. This is because there is a new line leftover after an integer prompt and using this nextLine(); consumes this extra bit.
* int choice = scanner.nextInt();  
  scanner.nextLine(); *// consume the new line after an integer*
  + I also made sure to make use of try-catch for my input validation for my menu.
  + Here I was planning the methods and making dummies for now. This was my first diversion from my class diagram. I realised that I wanted to make an extra function to identify the game by the ID for removing the game, especially.
* Customer.java
  + My immediate decision for the customer class was the data structure I was going to keep my existing and new customers in. I chose to work with a hashmap as I would anyways be having two values that need to be unique, the username and password.
  + In this class I was actually able to make use of all the overridden abstract methods such as register.
  + This is where I ran into my first major problem. As I was making the functions for the customer menu, which I naturally placed in the customer class. I was having an issue implementing it in the customer menu which I created in the store class. I believe the culprit was a little gimmick I made to make the login smoother for the user. So, I initialised the class differently and played with null:
* private static void customerMainMenu(*List*<*Item*> *gameList*) {  
   Scanner = new Scanner(System.*in*);  
   Customer = null;  
    
    
   *// Customer menu loop* while (true) {  
   System.*out*.println("\nCustomer Menu:");  
   System.*out*.println("1. Customer Login");  
   System.*out*.println("2. Register as a new customer");  
   System.*out*.println("3. Forget ID");  
   System.*out*.println("4. Back to Main Menu");  
   System.*out*.print("Enter your choice: ");  
    
   int choice = scanner.nextInt();  
   scanner.nextLine(); *// Consume the newline character* switch (choice) {  
   case 1:  
   customer = *loginCustomer*();  
   if (customer != null) {  
   customer.customerMenu(*gameList*);  
   }  
   break;  
   case 2:  
   customer.register();  
   break;  
   case 3:  
   customer.forgetID();  
   break;  
   case 4:  
   return; *// Return to the main menu* default:  
   System.*out*.println("Invalid choice. Please enter a valid option.");  
   }  
   }  
  }

For this to work, I had to create the login customer method in the main method for store. Due to this my reference for register and forgetID were not functioning properly. Again, I believe this was due to how I instantiated my customer class. I made a workaround by instantiating the class again:

case 2:  
 customer = new Customer("");  
 customer.register();  
 break;  
case 3:  
 if (customer != null) {  
 customer.forgetID();  
 } else {  
 System.*out*.println("Please log in before attempting to forget ID.");  
 }

I am however, quite scared this might negatively influence my customer class when it comes to interacting with the rest of the program as this infrastructure is quite unstable (especially with how customer was being instantiated twice). Thus, I decided to do another thing and just move the customer login methods to the main method:

private static void customerMainMenu(*List*<*Item*> *gameList*) {  
 Scanner = new Scanner(System.*in*);  
 Customer = null;  
  
  
 *// Customer menu loop* while (true) {  
 System.*out*.println("\nCustomer Menu:");  
 System.*out*.println("1. Customer Login");  
 System.*out*.println("2. Register as a new customer");  
 System.*out*.println("3. Forget ID");  
 System.*out*.println("4. Back to Main Menu");  
 System.*out*.print("Enter your choice: ");  
  
 int choice = scanner.nextInt();  
 scanner.nextLine(); *// Consume the newline character* switch (choice) {  
 case 1:  
 customer = *loginCustomer*();  
 if (customer != null) {  
 customer.customerMenu(*gameList*);  
 }  
 break;  
 case 2:  
 *registerCustomer*();  
 break;  
 case 3:  
 *forgetIDCustomer*();  
 break;  
 case 4:  
 return; *// Return to the main menu* default:  
 System.*out*.println("Invalid choice. Please enter a valid option.");  
 }  
 }  
}  
  
*// login customer method*private static Customer loginCustomer() {  
 Scanner = new Scanner(System.*in*);  
  
 System.*out*.print("\nEnter your username: ");  
 String username = scanner.nextLine();  
  
 System.*out*.print("Enter your ID: ");  
 String enteredID = scanner.nextLine();  
  
 *// check if the entered ID matches the stored ID for the given username* if (Customer.*userCredentials*.containsKey(username) &&  
 Customer.*userCredentials*.get(username).equals(enteredID)) {  
 System.*out*.println("Login successful!");  
 return new Customer(username);  
 } else {  
 System.*out*.println("Incorrect username or ID. Login failed.");  
 return null;  
 }  
}  
  
*//register customer method*private static void registerCustomer() {  
 Scanner = new Scanner(System.*in*);  
  
 System.*out*.print("\nEnter your desired username: ");  
 String username = scanner.nextLine();  
  
 *// check if the username is already taken* if (Customer.*userCredentials*.containsKey(username)) {  
 System.*out*.println("Username already exists. Please choose a different one.");  
 return;  
 }  
  
 *// generate a random ID for the new user* String newID = *user*.*generateRandomID*();  
  
 *// store the new user's credentials* Customer.*userCredentials*.put(username, newID);  
  
 System.*out*.println("Registration successful! Your ID is: " + newID);  
}  
  
private static void forgetIDCustomer() {  
 Scanner = new Scanner(System.*in*);  
  
 System.*out*.print("\nEnter your username: ");  
 String username = scanner.nextLine();  
  
 *// check if the entered username exists* if (Customer.*userCredentials*.containsKey(username)) {  
 String storedID = Customer.*userCredentials*.get(username);  
 System.*out*.println("Your ID is: " + storedID);  
 } else {  
 System.*out*.println("Username not found. Please register or try again.");  
 }  
}

After testing, the program works consistently and smoothly. I still feel I can do this in a better way. So, I will revisit this issue once I am done with the other classes if I have the time.

* I then proceeded to make the child classes for the Item abstract parent class. The two classes were VideoGame.java and BoardGame.java. I just defined the unique parameters, made getters and setters, and did an override to the toString method of Item.
* Now I needed to make the methods and functionality of my program regarding the items. I decided on arraylists for the data structure for the games. I instantiated it in the main method Store.java. I then built the methods for interacting with the arraylist in the clerk class first. I started with the addGame function and realised I needed to make another diversion from my class diagram. I had to make another class to facilitate a menu for the clerk to choose the item type to create. This is good flexibility as I can add more item types later if I please (Store merchandise is an item type I hope to implement later).
* I didn’t have too many major problems here thankfully. I was able to make the addGame, removeGame and displayGames features without too much hassle. I did have to do more research with printf and try-catch for a bit but nothing too troublesome.
* I tested the item creation and interaction with the arraylist and classes and felt satisfied enough to move to the customer class functionality.
* So, I constructed the class and basic attributes and functionalities for the customer class. There wasn’t too much hassle at first, but later I came across a plethora of problems. Aside from the issue with the registration customer methods not being able to be placed in the customer class. I made the second customer menu and the rent, return, display rented and available games functions. I added a search function later. I had to make a couple of extra private methods to help the larger search and display methods, this was nothing to different from what I did with the clerk class.
* When I was testing the program, I realized I had a minor issue of how the program would be case-specific in terms of the inputted usernames. So, the program would not differentiate between ‘Balls’ and ‘balls’, it would treat this as two different usernames. I fixed this by just setting everything to lower case for the perspective of the program:
* System.*out*.print("\nEnter your username: ");  
  String username = scanner.nextLine().toLowerCase();
* My next issue was with the memory of the program.
* During my testing I had come to realise there was an issue in my program, the issue stemmed from a workaround I created earlier with the loginCustomer() method.
  + I was having a difficult time trying implement and call the created customer from the registerCustomer() method and therefore for the timebeing, I made a quick workaround and just created a new customer within the function itself because that worked.
  + The consequence of this action turned up later when I was testing the functionality for my arraylist ‘rentedGames’.
* Here I realized that when I rented a game as a customer, the game entry object was added to the ‘rentedGames’ arraylist. However, when I logged out of the customer and logged back in, the rentedGames arraylist would be wiped clean but the object would still appear rented and unavailable. After looking through my code I realised it was because the loginCustomer method would create a new Customer each time and with each new Customer, a new rentedGames arraylist was created. Essentially, there was no memory or solid link.
  + Therefore, I realized I needed a short-term memory system for my program to remember the arraylist being tied to the rentedGames.
  + After research and talking to my friends, I decided on trying a read/write file system as this could also fulfil my additional task requirement.
  + I did some basic research and with the help of some friends, I constructed the methods for the saving and loading and initialised the data file into the constructor as well.
  + Thankfully, I didn’t run into too many issues, and this solved my issue of the rented games list not printing when logging in fairly easily.
  + I did have one issue later where the rentedGames() function was displaying the results of the previous debugging, this was because I didn’t have a method that erases and resets the arraylist in between runs of the program. The solution was just to create method that deletes the data on program exit.
* After I was done there, for no reason at all I ran into another problem where upon logging out, the customer menu was called again. This results in the user having to press the log out option twice.
  + After searching my code, I realised I accidentally called the customerMenu() method within the login function. I have no idea why this issue didn’t present itself until after the memory issue with customer.
* From here, for the most part, customer seemed to be working alright and well enough.
* Here I referred to the coursework guidelines, to see if I had missed anything. I realised I had forgotten to add a modify function to the clerk so they would be able to modify data. I started drafting the function and implementation for this. I immediately realised I would need my program to be able to differentiate between the types of objects and their unique attributes to modify. Making this task slightly harder than I thought it would be. However, upon reference to my previous works, I remembered I had a similar issue before and made use of a fancy command in java known as ‘instanceof’. This helped immensely for the modify feature.
* I am also slightly worried and unsure of my implementation of interfaces. So, far I have one interface that implements three fairly frequently used methods, but I am not sure if I should do more.
* The next thing I need to do is create automated tests. I wasn’t too sure how to implement this, but I believe that my currently pre-defined objects and the comments as well as the confirmation print statements all contribute to the testing part of the program. Along with the rigorous testing I’ve done myself of my program.
* One other thing I wanted to work on before moving on was the implementation of another child class for item. Something that wasn’t a game but rather an item that can be sold. So, I started working on my merchandise class and it’s integration into my program.
  + For the most part it went fairly smoothly, I was able to make good use of the parent class. However, I came to realise that the merchandise class was actually working quite differently from the game classes. Therefore, more menus had to be made and consideration for the functionality.
  + For some reason I couldn’t just make a whole method for the merchandise functions. I think scope issue and so I needed to make helper methods for the merch methods. Nothing too troublesome but it bothers me to an extent.
  + I had some menu skipping due to no scanner.nextLine() in customer buy merch. Simply inputting this line after the integer prompt fixed this issue but solving it took a bit of time.
* Another thing I wanted to do was revamp the availability system. I originally intended to just have that because I was more focused on fixing the functionality of the program and settling that but once I neared the end, I wanted to attempt at implementing a stock system. So, I initialised the attribute in the parent class item and went ahead implementing it into the program and the other classes and functions.
* Generally, so far, I’m quite happy with my program. I had to make a few fixes and upgrades:
  + I had to improve the search function to display a better output and make sure it would account for incomplete or ‘lazy’ searches where the user would only input a partial bit of the title. This was done mainly using if logic with the .contains() tool.
  + I had to improve the addGame() methods of the clerk to make sure that name duplicates were not possible. This was also mainly done with if logic and the .equals() tool. I was also able to implement this into the addMerchandise() method.
* Around this time, I had received the feedback for my coursework 1. After reviewing it, I decided to make corrections and improve my coursework 2 program based of this.
  + One thing I implemented was an improvement to the search function. I improved it by making it so that the input trims the whitespaces and accounts for incomplete inputs in the search.
  + I improved the display for the inventory and merchandise.
  + I had an issue where the data for my rentedGames arraylist was not being properly deleted in between debugging. I solved this by including a line of code that resets the file each time the program is restarted.
* Runtime.*getRuntime*().addShutdownHook(new Thread(() -> Customer.*deleteDataFile*()));
  + I made more improvements to the merchandise section
    - I made a very large section when I showed my program to a senior and they suggested improving the merchandise section to allow for consecutive purchases and more improvements.
    - Adding some validation and considering cases like checking out when the cart is empty.
    - I also had an issue where my item stock wasn’t depleting when the item was being added to the cart, I had to solve this by creating additional methods.
  + One of the last changes I made was making sure that a game object can exist with the same name only if the same name doesn’t exist for either board or video games.
    - Basically the game ‘monopoly’ can be a video game and a board game.

1. **Reflections**

* That covers a good bit on the problems I encountered in my program and the solutions I sought for them.
* I want to talk about a bit about what I was proud about for my program. I took this coursework as an opportunity to work on the mistakes and things I didn’t like about coursework 1. One thing I really didn’t like about my coursework 1 was the organisation and planning. I did not have a solid plan for my program and in the end the code and structure were very messy, and it became the cause for many issues throughout that I had to find workarounds for.
* So, for the coursework 2, I made sure to put a lot of time into the planning and thought process of the program structure. The class diagram I constructed helped a lot and I had a much better idea for the coding process. I was a lot more organized in constructing the menus and methods for the program. This definitely helped a lot as I could tell the number of miscellaneous issues, I was facing was a lot less and the program code worked generally well on the first set of tries. Another thing I would be a bit proud of is the implementation of merchandise and the save/load system for customer.
* Attached below is my relation tree of the final program:

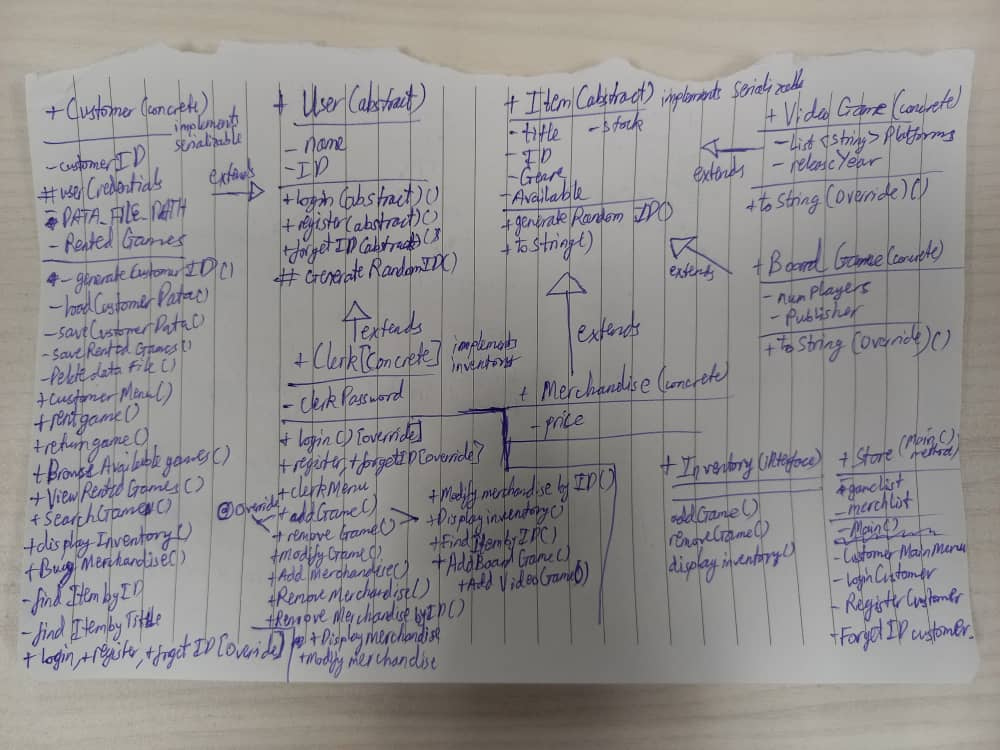


Image 5: A very crude near-final attempt at my relation tree

I have attached the final and revised version of the relation tree.

Reading list:

1. <https://www.geeksforgeeks.org/why-is-scanner-skipping-nextline-after-use-of-other-next-functions/>
2. <https://www.freecodecamp.org/news/what-is-a-java-hashmap/>