Data Science Degree Apprenticeship OOP Portfolio

Kit Ropner

BS2202: Object Oriented Software Development A 23/24 (S2)

Module Leader: Claire Ancient

Word Count:

Table of Contents

[Introduction 3](#_Toc169608625)

[Part 1: Project Scope and Client Requirements 5](#_Toc169608626)

[Project Scope 5](#_Toc169608627)

[Client Requirements 5](#_Toc169608628)

[Functional Requirements 5](#_Toc169608629)

[Extended Requirements 5](#_Toc169608630)

[Non-Functional Requirements 6](#_Toc169608631)

[Part 2: User Interface Diagrams 7](#_Toc169608632)

[Login Window 7](#_Toc169608633)

[Home Page 8](#_Toc169608634)

[Stock Page 8](#_Toc169608635)

[Report Page 10](#_Toc169608636)

[Part 3: Class and Database Diagrams 11](#_Toc169608637)

[Unified Modelling Language (UML) 11](#_Toc169608638)

[Entity Relationship Diagram (ERD) 11](#_Toc169608639)

[Design Patterns 12](#_Toc169608640)

[Reconsiderations 13](#_Toc169608641)

[Part 4: Testing Documentation 13](#_Toc169608642)

[Testing plan 13](#_Toc169608643)

[Evidence of testing results 13](#_Toc169608644)

[Evidence of problem solving 19](#_Toc169608645)

[Conclusion 21](#_Toc169608651)

[References 21](#_Toc169608652)

[Appendices 22](#_Toc169608653)

[Appendix A: Project Proposal 22](#_Toc169608654)

[Introduction 22](#_Toc169608655)

[Key System Features 22](#_Toc169608656)

[Core Item List 22](#_Toc169608657)

[Functional Requirements 22](#_Toc169608658)

[Extended Requirements 23](#_Toc169608659)

[Non-functional Requirements 23](#_Toc169608660)

# **Introduction**

Object Oriented Programming (OOP) is a programming paradigm which uses classes to modularise code (Doherty, 2024). Each class has its own attributes and behaviours that can be inherited by objects. This approach encourages reproducible and consistent code.

In this portfolio, I will demonstrate my knowledge and understanding of OOP and software development by creating a stock management system for a video game company. It will utilise:

* Object Oriented Programming (including inheritance)
* Software development techniques
* Database design (using MySQL)
* A Graphical User Interface (GUI) with JavaFX

In order to document my progress, this portfolio will cover:

* Project Scope and Client Requirements
* User Interface Diagrams
* Class and Database Diagrams
* Testing Documentation

# **Part 1: Project Scope and Client Requirements**

## Project Scope

A video game company wants to have a stock management system for its employees to keep track of (and update) its stock of games and consoles. It must feature a login system, homepage, reports and the ability to upload CSVs. Furthermore, visualisations should be included to provide an effective way of viewing stock levels at a glimpse. Additional features will allow staff to manage discounts.

Developing the system must make use of object-oriented design, relational databases and graphical user interfaces with JavaFX. To ensure the validity of these systems, I will be conducting manual tests.

## Client Requirements

In order to make sure all the needs of the customer are met; a list of requirements is necessary to keep track of software features.

Requirements are usually gathered by way of an interview with stakeholders, or surveys to customers. However, as this is a simulated project, I have come up with these requirements myself.

### Functional Requirements

*Mandatory features to meet to customer’s needs.*

*Priority in descending order (i.e. 1 is the most important)*

|  |  |  |
| --- | --- | --- |
| **Req. ID** | **Requirement Description** | **Priority** |
| 1 | Relational database system with object-oriented development in mind. It must contain both game and console tables with item names, stock levels and prices. Tables should also be stored for customers, orders and employees. See the ERD for full detail. | 1 |
| 2 | Intuitive GUI with a home page and data dashboard. See the Wireframe for full detail. | 2 |
| 3 | User Authentication that uses credentials from the database. This will prevent public access. | 3 |
| 4 | Stocks and sales recorded by upload of CSVs. | 4 |
| 5 | Analytics on transactions in the form of visualisations and reports. | 5 |
| 6 | Discount feature on select items to reduce price by 30%. | 6 |

### Extended Requirements

*Features that would be nice to have, but aren’t necessary.*

|  |  |
| --- | --- |
| **Req. ID** | **Requirement Description** |
| 7 | Additional information (i.e. an about page) on the developer as well as financial information (e.g. refunds, tax rates, currency conversion and discounts). |
| 8 | Alerts for low stock levels. |
| 9 | Search and filters to more easily find products. |
| 10 | Option for staff feedback |
| 11 | Option for refunds |

### Non-Functional Requirements

*Qualities of the system (e.g. performance and usability).*

|  |  |
| --- | --- |
| **Req. ID** | **Requirement Description** |
| 12 | GUI must be accessible and user-friendly, in line with Nielsen’s heuristics. |
| 13 | Data from the database should operate quickly with low read/write times. |
| 14 | Data should be stored securely and handled in line with GDPR. |
| 15 | System should be easy to maintain by making code and documentation readable. |

# **Part 2: User Interface Diagrams**

In order to create a comprehensive GUI, I first need to plan the design. I can do this using Balsamiq, an interactive wireframe builder. Wireframes are visual diagrams that outline the framework of an app.

My interface must be able to house all the requirements listed previously, whilst also accounting for usability. Of particular importance is Sustainable Development Goal 10 (SDG

10). As stated by the United Nations International Children’s Emergency Fund (UNICEF, 2024), SDG 10 aims to reduce inequalities.

In the context of a GUI, this could mean making the interface more accessible. For example, by offering dynamic text sizes to account for those with visual impairments.

This interface will be structured in 4 stages:

* A login window
* A home page
* A stock page
* A report page

## Login Window

Upon launch, a login popup appears (Figure 1), prompting the user to enter their credentials. As this is a company application, it will only accept employee credentials. Any wrong credentials will produce an error message.

A computer screen with a login and password

Description automatically generated

Figure 1 Login popup

## Home Page

If the correct credentials are entered, the user proceeds to the home page (Figure 2). This displays a welcome message, current date and relevant statistics (e.g. current stock levels and recent sale trends).

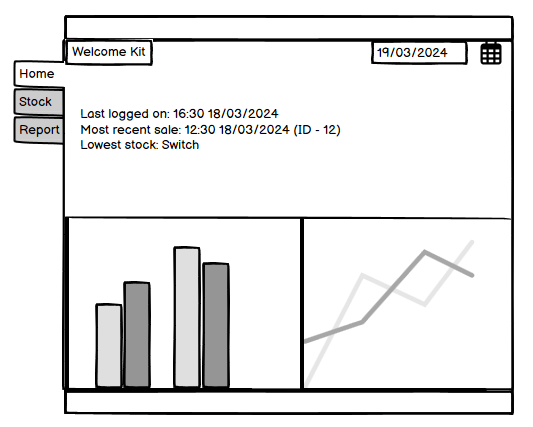


Figure 2 Home page

## Stock Page

Using the side bar, the user can access the stock page, where all items, their stock levels and prices are listed (Figure 3). Filters and searches can be used to filter the data and an upload button allows a user to upload their own stock and sale updates (Figure 4).

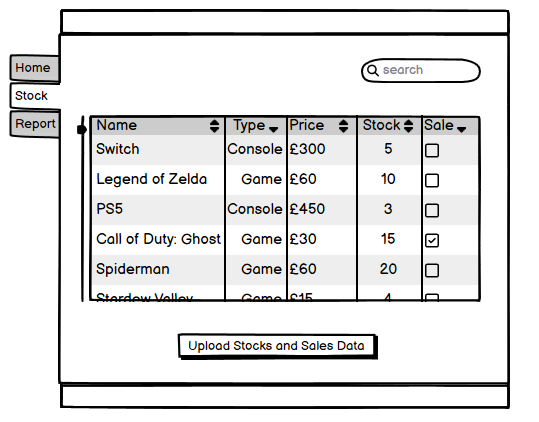


Figure 3 Stock page

A screenshot of a computer

Description automatically generated

Figure 4 Upload popup on stock page

## Report Page

Finally, the user can also access the Report page (Figure 5) from the side bar. To generate a report, the user must supply a date range, report type and product type. The examples of each are listed in the input boxes.

The user can also access previous reports from the bottom half of the screen. It can hold a maximum of ten reports before deleting the oldest one. It is therefore recommended to download reports for long-term storage.

A screenshot of a report

Description automatically generated

Figure 5 Report page

# **Part 3: Class and Database Diagrams**

## Unified Modelling Language (UML)

My Unified Modelling Language (UML) diagram (Figure 6) illustrates the structure of my stock management system.

Both games and consoles are products, and so inherit from class *Product*. This means they share common attributes like *Product ID* (used later as a primary key) and *Stock*.

Sometimes, the company offers sales (15% off) which is reflected in the *Sale Price*.

A screenshot of a computer

Description automatically generated

Figure 6 Full UML for games and consoles

## Entity Relationship Diagram (ERD)

The attributes (lowercased for convention) are brought forward from the UML diagram into the Entity Relationship Diagram (ERD) (Figure 7).

Both *Game* and *Console* tables inherit attributes from the Product class from earlier, whilst also possessing their unique attributes from their respective classes. All attributes are labelled with their corresponding data type.

As a game can belong to many consoles and a console can have many games, a join table is required (*Game\_Console)*. This makes use of their primary keys (*product\_id)* to relate them to each other.

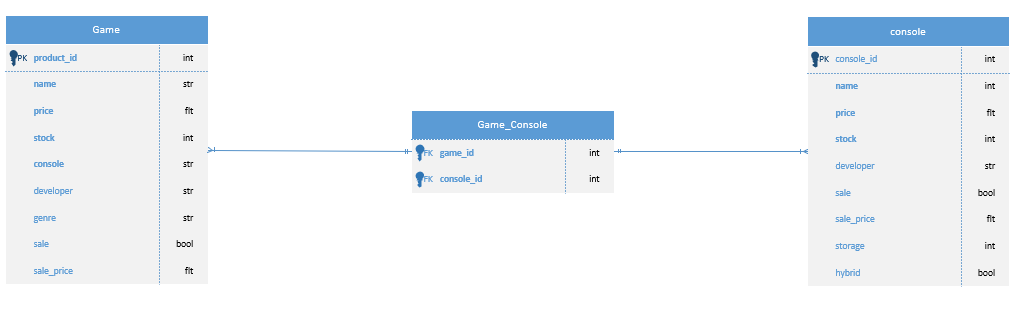


Figure 7 Initial Game ERD

To add a customer side to this UML, we will require an *Order* table that links together *Product*, *Customer* and *Employee* as foreign keys (Figure 8). Each product, customer and employee can belong to many orders, but each order must have one product, customer and employee.

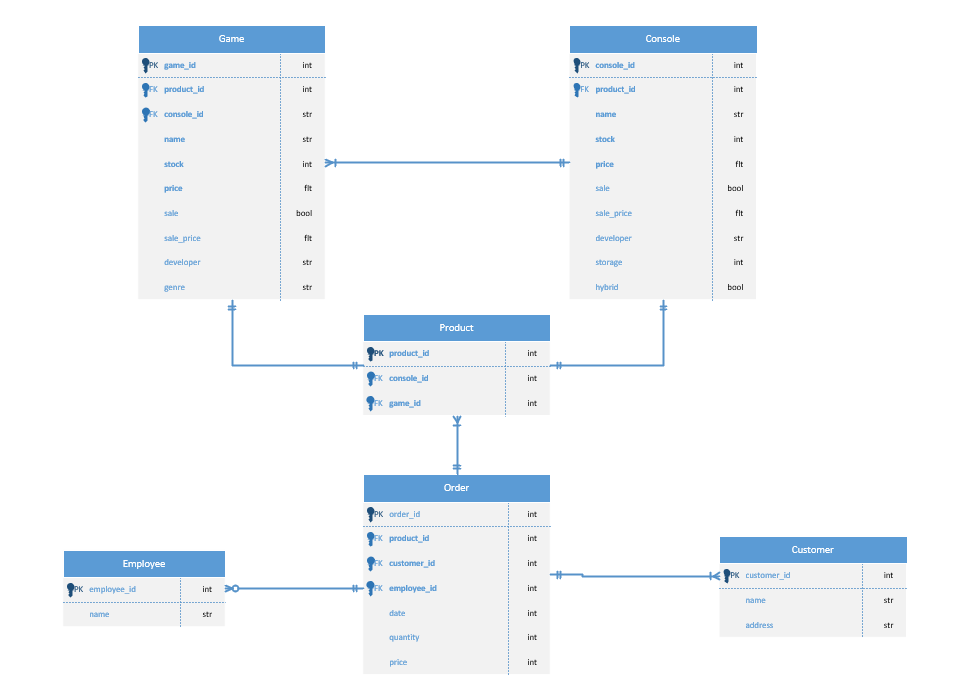


Figure 8 Full Game ERD

To reduce duplication of data and simplify data management, this relational database is in third normal form (3NF). This means that all non-key attributes are fully dependant on a primary key and has no transitive partial dependencies.

## Design Patterns

Design patterns are repeatable solutions for a common problem (GeeksForGeeks, 2024) and are frequently used in object-oriented software development. There are three types of design patters: creational, structural and behavioural which provide abstract solutions to their respective areas.

I have made use of design patterns during my design process. For example, the *Product* table in Figure 8 follows the structural design pattern Adapter to connect *Game* with *Console*.

## Reconsiderations

After implementing my database with Java. I realised that the *Product* was an inefficient way of placing *Console* and *Game* orders. Instead, I kept *Product* as a parent for *Console* and *Game* in the UML, but replaced it with an *Order-Item* table in the ERD as to not get confused.

In addition, the *Order-Item* contains the *order-id* and *quantity* (instead of *Order*) to act as a join between *Game, Console* and *Order*. Its function allows for multiple games and orders to belong to one order.

Also, the *product\_id* was removed from all other tables as it was redundant and a *password* field was added to the *Employee* table for login.

Lastly, the sale was reduced to 10% for more round numbers.

Unfortunately, as Visio is only licenced on University laptops, I cannot recreate the new diagram here, but its structure is present in my database.

# **Part 4: Testing Documentation**

To carry out sufficient Verification and Validation (V&V) I will create and carry out a testing plan, pointing out any errors and how I might solve them.

## Testing plan

Aim: Identify errors that would cause the system to crash or not function as expected.

Scope:

* Validate data
* Test inputs and outputs
* Assess GUI usability

Approach:

* Manual testing
* Try/Catch statements

## Evidence of testing results

Figure 9-12 tests the login feature with incorrect and correct passwords.

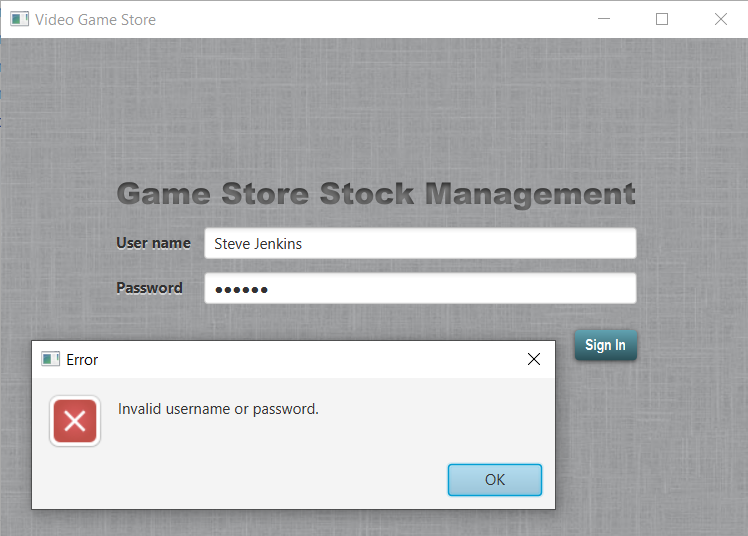


Figure 9 Test with wrong password

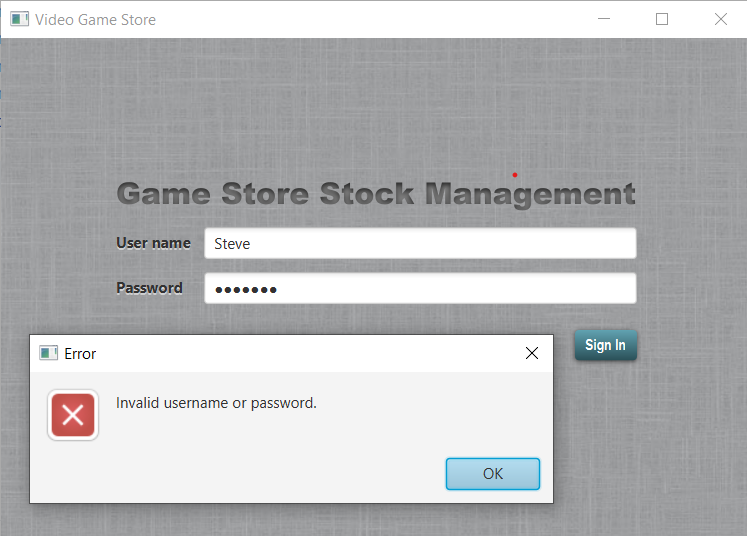


Figure 10 Test with wrong username

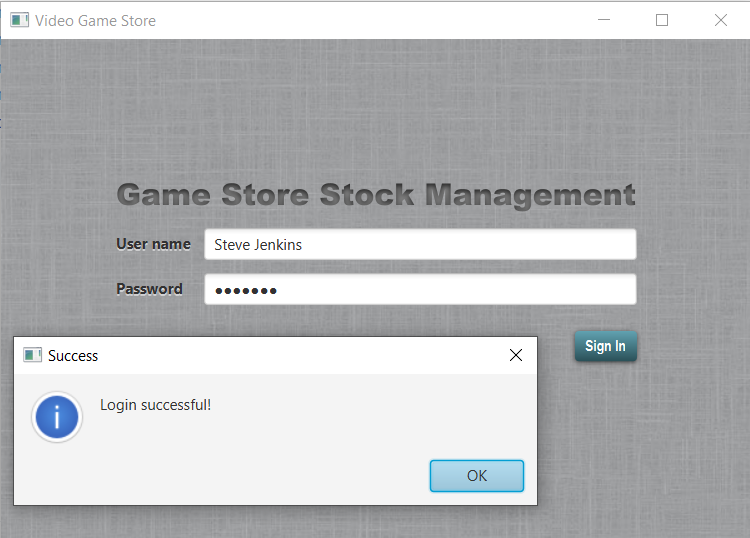


Figure 11 Test with right password and right username

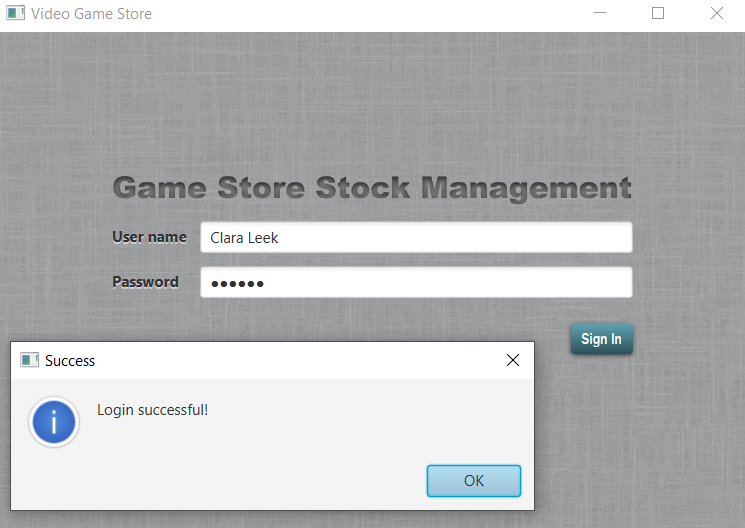


Figure 12 Test with different username and password

Figure 13-15 shows the stocks being updated through the stocks page csv uploader.

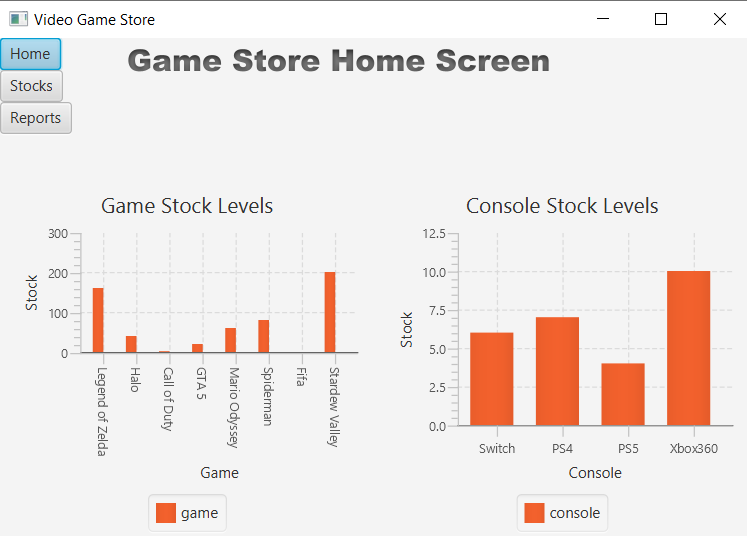


Figure 13 Home screen displaying graphs (Fifa and Call of Duty low on stock)

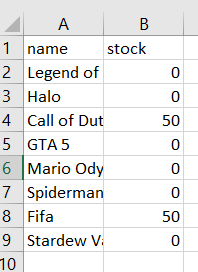


Figure 14 Adding stock via csv

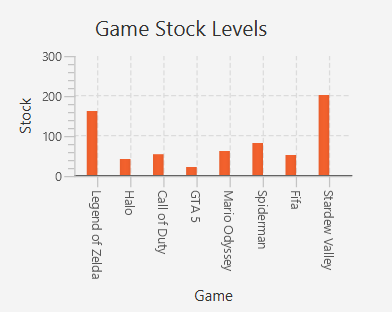


Figure 15 Game bar chart after adding stocks (Fifa and Call of Duty stocks increased)

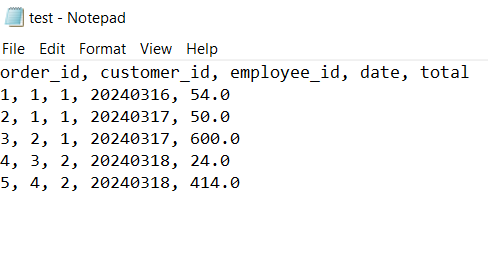


Figure 16 Report generated in a txt

Figure 16 shows the txt report generated on the Report page. The data and structure of the report align with what is expected.

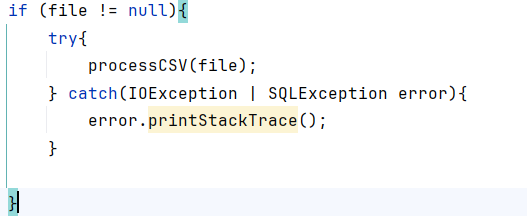


Figure 17 try/catch statement

I’ve used try/catch statements (Figure 17) in my code to flag errors that occur. *printStackTrace* outputs a trace to the console to help identify what caused the error.

Comments precede functions in the controller classes for context, but otherwise comments are sparse. As argued by Jalli (2024), “the code should always be clear enough to express itself”.

From interacting with the GUI, it performs as expected, with responsive buttons and animated graphs.

## Evidence of problem solving

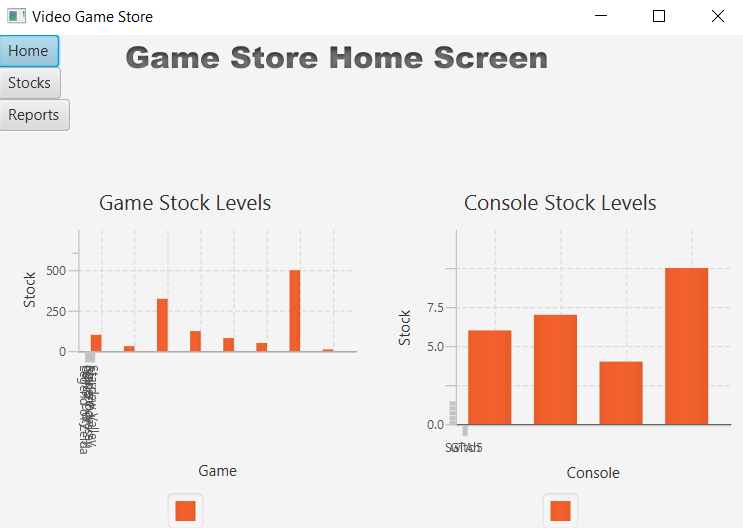


Figure 18 X Axis bunched into bottom left corner

The X Axis labels in Figure 18 have been clumped together, this a known issue[[1]](#footnote-1). To fix this, I can use *setAnimated(false)* on the bar chart.



Figure 19 Using SQL LIMIT with Hibernate

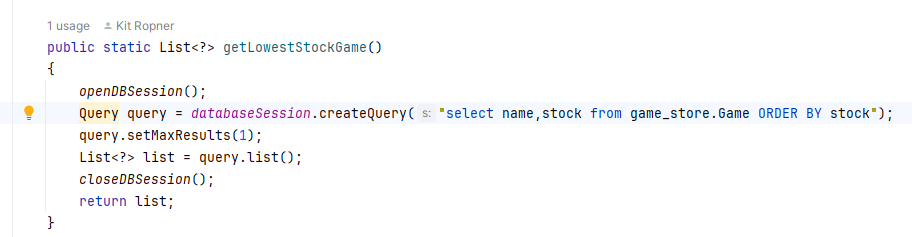


Figure 20 Using setMaxResults(1) instead of LIMIT

Using LIMIT in Hibernate Query Language (HQL) causes the error in Figure 19, this is a known issue[[2]](#footnote-2). Instead I can use *setMaxResults* after the query (Figure 20).

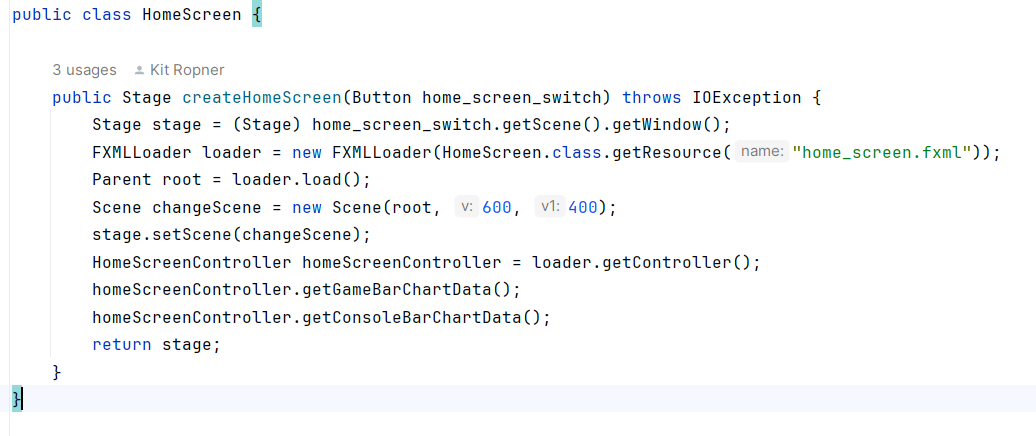


Figure 21 A separate function to call Home Screen to reduce duplicated code

As home screen features additional functions (like *getConsoleBarChartData* and *getGameBarChartData*), a separate function call means that these extra lines of code aren’t called repeated in each controller.

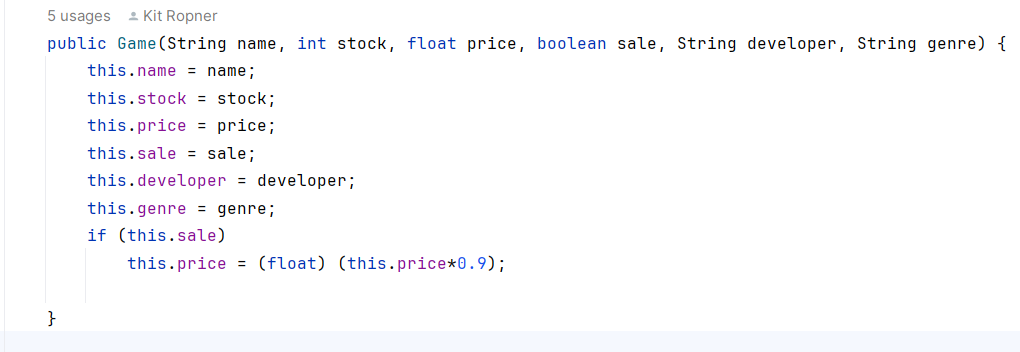


Figure 22 A constructor for Game with an if statement

My constructor for Game (Figure 21) features an if statement to incorporate my bonus feature (sales) that applies a 10% discount to *price* if *sale* is true.



Figure 23 Tidying up unused and possibly faulty code

Lastly, using IntelliJ’s warnings (Figure 22), I’ve reduced redundancies in my code.

# **Conclusion**

Making use of object-oriented design, relational databases and graphical user interfaces, I have demonstrated learning and understanding of OOP concepts.

Through this project, I have developed hands on experience with Java, JavaFX, CSS, SQL and Hibernate; resulting in an easy-to-use stock management system for basic inventory management.

Using data science, I have produced graphs for a quick summary for employees. Meanwhile, tables and reports provide a more detailed look at the data.

# References

Doherty, E., 2024. *What is object-oriented programming.* [Online]   
Available at: https://www.educative.io/blog/object-oriented-programming  
[Accessed 28 05 2024].

GeeksForGeeks, 2024. *Design Patterns in Java.* [Online]   
Available at: https://www.geeksforgeeks.org/java-design-patterns/  
[Accessed 28 05 2024].

UNICEF, 2024. *GOAL 10: REDUCED INEQUALITIES.* [Online]   
Available at: https://data.unicef.org/sdgs/goal-10-reduced-inequalities/  
[Accessed 31 05 2024].

# **Appendices**

## Appendix A: Project Proposal

### Introduction

|  |  |
| --- | --- |
| **Stock Management System Name:** | Video Game Stock Management |
| **Stock Management System Description:** | Manages stock for video games and consoles. Allows staff to view current stock levels through data dashboards and visualisations. Sales and stocks can be updated via csv and reports can be generated from queries. Staff can also manage discounts. A login system will ensure only validated employees are allowed to access the system. |

### Key System Features

|  |
| --- |
| * A homepage showing recent transactions * A page to filter through existing products to check stock levels * A report that returns a text-based output dependant on a specific query (e.g. how many PlayStations were sold in March 2020) * User authentication * Low stock level alerts * Filters * Feedback * 20% discounts on certain games/consoles |

### Core Item List

An example of the game table. The actual database will be relational with additional rows, columns and tables (including consoles, orders and employees).

|  |  |  |
| --- | --- | --- |
| **Item Description** | **Cost Price** | **Selling Price** |
| Legend of Zelda | 50 | 60 |
| Mario | 45 | 50 |
| Stardew Valley | 10 | 15 |

### Functional Requirements

Mandatory requirements to meet to customer’s needs.

*Priority in descending order*

|  |  |  |
| --- | --- | --- |
| **Req. ID** | **Requirement Description** | **Priority** |
| 1 | Relational database system with object-oriented development in mind. It must contain both game and console tables with item names, stock levels and prices. Tables should also be stored for customers, orders and employees. See the ERD for full detail. | 1 |
| 2 | Intuitive GUI with a home page and data dashboard. See the Wireframe for full detail. | 2 |
| 3 | User Authentication that uses credentials from the database. This will prevent public access. | 3 |
| 4 | Stocks and sales recorded by upload of CSVs. | 4 |
| 5 | Analytics on transactions in the form of visualisations and reports. | 5 |
| 6 | Discount feature on select items to reduce price by 30%. | 6 |

### Extended Requirements

Would be nice to have, but aren’t necessary.

|  |  |
| --- | --- |
| **Req. ID** | **Requirement Description** |
| 7 | Additional information (i.e. an about page) on the developer as well as financial information (e.g. refunds, tax rates, currency conversion and discounts). |
| 8 | Alerts for low stock levels. |
| 9 | Search and filters to more easily find products. |
| 10 | Option for staff feedback |
| 11 | Option for refunds |

### Non-functional Requirements

Qualities of the system (e.g. performance and usability)

|  |  |
| --- | --- |
| **Req. ID** | **Requirement Description** |
| 12 | GUI must be accessible and user-friendly, in line with Nielsen’s heuristics. |
| 13 | Data from the database should operate quickly with low read/write times. |
| 14 | Data should be stored securely and handled in line with GDPR. |
| 15 | System should be easy to maintain by making code and documentation readable. |

1. [[JDK-8198830] BarChart: auto-range of CategoryAxis not working on dynamically setting data - Java Bug System (openjdk.org)](https://bugs.openjdk.org/browse/JDK-8198830) [↑](#footnote-ref-1)
2. [spring - unexpected token: LIMIT - Stack Overflow](https://stackoverflow.com/questions/40574674/unexpected-token-limit) [↑](#footnote-ref-2)