Introduction

Our project is based on the functionality of an EB Games store. It tracks and manages the flow of stock and purchases made by customers in the store. It checks whether stock needs to be replenished from the supplier. It also tracks customer purchases and updates the store's inventory based on these purchases. Our project will also distinguish between items, such as games and consoles, and track the respective stock based on each console and its games.

Design Description Assessment Concepts

Originality and complexity of selected problem
The shop requires particular concepts of C++ and object-oriented design to create a fluid implementation of the system. The difficulty of the problem arises when needing to understand how a shop interacts with a customer. The list of items chosen by the customer must be stored in a data structure, where the memory for this data is allocated in the stack or the heap. The user input and output is demonstrated in the main program, where the user chooses items to buy, indicates whether they are a member and so on. The object-oriented programming and design is used to construct the relationships between the classes and how to simplify the code. Identifying where concepts such as polymorphism, abstract classes and virtual methods are used in the shop system eases the implementation. Exhaustive testing is to ensure that our code is rigid and does not break even with extreme cases.

Specification
We will ensure that all code matches the descriptions defined in our plan. We will run our program in increments to ensure it compiles appropriately and produces the correct output. We will assist each other with debugging errors in our code.

Readability
We will ensure that the code is readable by using the correct indentation for all state and behaviour in our code. For example, we will separate our attributes and methods in our header files by using a space. We will also include comments in our program to explain what each function is doing.

Efficiency
We will ensure that the project is efficient by correctly using pointers and dynamic memory. We will also implement destructors where required to free memory we no longer need. Any dynamically allocated memory will also be deleted when it is no longer needed.

Use of the concept "Memory allocation from stack and the heap"

Arrays: We will use dynamic arrays to keep track of stock in the store and to modify quantities of stock based on items purchased by the customer.

Strings: Strings will include item titles, shop name, shop address, supervisor name, supplier address and supplier postcode.

Classes: Shop, shop supervisor, supplier, item, console, game customer.

Objects: Console, Game, PS4, Xbox One, Switch, PC

Use of the concept “User Input and Output”

I/O of different data types: User inputs into terminal. The user selects the games and/or consoles they want to purchase. The price of the items is printed, and the user can confirm if they wish to make the purchase. The final output will be a receipt summarising the purchase made by the customer.

Use of the concept “Object-oriented programming and design”

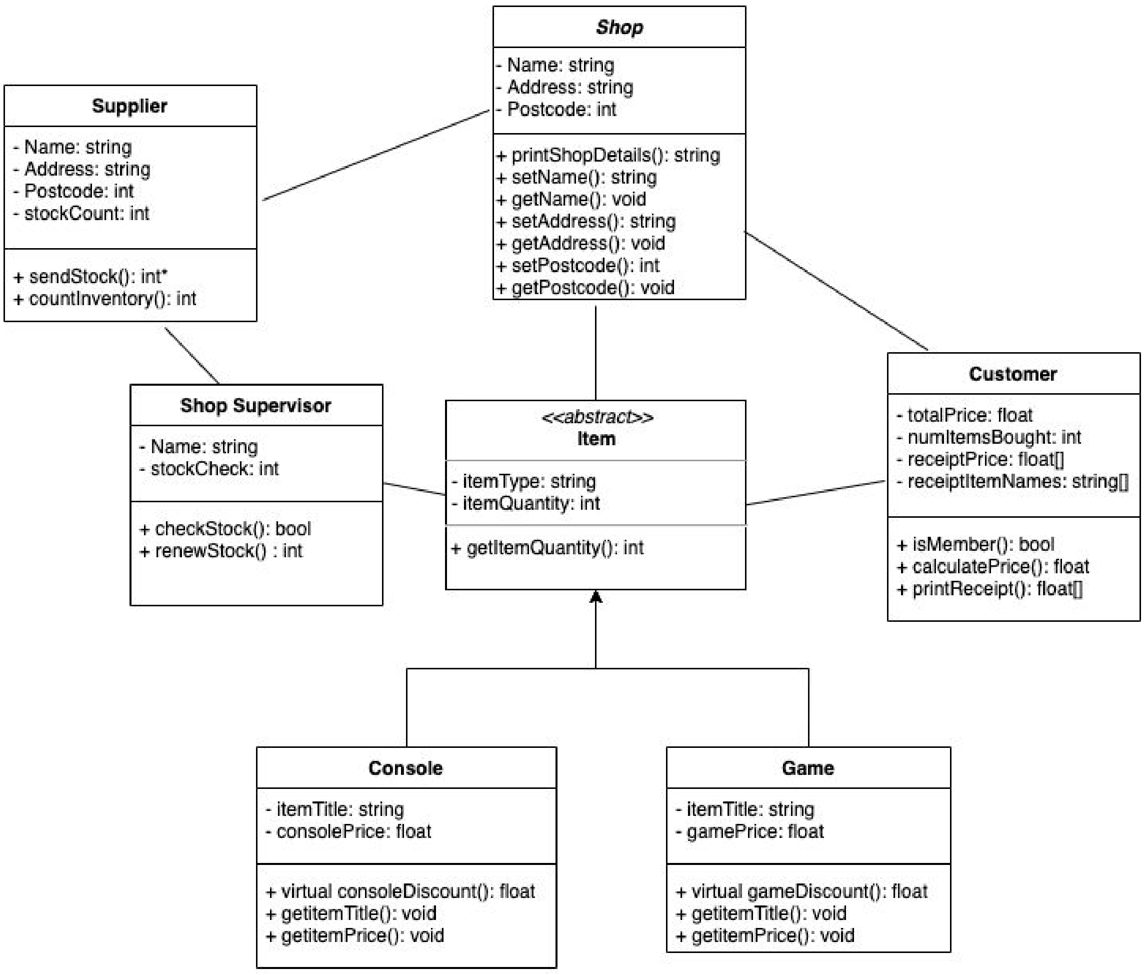
Inheritance: The console and game classes inherit the attributes from the item class since the attributes between games and consoles in a shop are shared. An item in the store must have a type and quantity. However, each console and game in the console and game class respectively has a unique console price and title.

Polymorphism and Abstract Classes: In our project, we will have an item class as an abstract class. The item class will have polymorphic behaviour within the consoleDiscount() and gameDiscount() methods, as a console and game will have different member discounts (10% and 5% respectively).

Use of the concept “Testing”

A series of unit, integration, regression and automated tests will be conducted to ensure that all expected inputs and outputs are displayed correctly. More detail is provided later in the document.

Class Diagram:



Class Descriptions

Shop

- The shop class is a class that defines a single shop with a name, address, postcode and shop ID. This class prints the details of the shop to notify the user where they are shopping from.

Item

- The item class is an abstract base class for the console and game classes. Defines variables present in both the console and game classes.

Console

- The console class inherits the attributes from the item class to indicate the type and quantity. The console also has its own unique attributes such as the item title and price.

Game

- The game class also inherits the attributes (type and quantity from the item class) to indicate what console the game is played on and how many units of the same game exist in the shop.

Customer

- The customer class is a subclass of the item class. The customer class will interact with the terminal in order to purchase a console or game.

Supplier

- The supplier class is responsible for sending stock when supply is low/under capacity

Shop Supervisor

- The shop supervisor class will check the stock and contact the supplier to renew stock when the stock is low. There is an association between the shop supervisor and the supplier

User Interface:
We will have output embedded in our program that provides instructions to the user. Some examples are listed below.

```
Hi. Welcome to EB Games *display name, address and postcode of
shop.*

---- Starting purchase ----
Would you like to purchase a game or a console?

---- Choosing a game----
Please look through the list of games available and …

---- Choosing a console----
Please look through the list of consoles available and …

---- Selecting item ----
You have selected *console/game title* for *price*. Press Y to
proceed to purchase or N to go back.

---- Making purchase ----
Are you a member with us? Press Y to indicate 'YES' or N to
indicate 'NO'

If 'YES', a 10% discount will be applied to your purchase! :)

Can you please confirm that you wish to purchase *console/game
title* for *price*. Select Y to confirm or N to change.

---- Purchase summary ----
Congratulations! You have purchased…. We hope you return soon!
```

Code Style
Each class will have its own header file containing variable and function definitions and a c++ file containing the implementations of these functions. Each class definition will also contain a comment describing the purpose of each class' functions, and what these functions will do. Each function will have a comment describing its purpose, the variable being returned, and what its parameters are.

Testing Plan

- **Unit Testing**
  - Use text files for input e.g. input.txt
  - E.g. diff input01.txt output01.txt
  - E.g.2. ./a.out < input01.txt | diff - output01.txt
  - Test each method/function with simple inputs and expected outputs
  - Use edge cases, boundary cases, unexpected input e.g. empty strings/arrays, test for 0, negative numbers, different types e.g. float for type int

- **Regression Testing**
  - Test previous code made for main/class.cpp files after adding any new functions
    - E.g. Repeat old tests for previous code whilst adding new code

- **Automated Testing**
  - Use of makefile to conduct automated tests
  - Makefile will work with different input files and test each input given

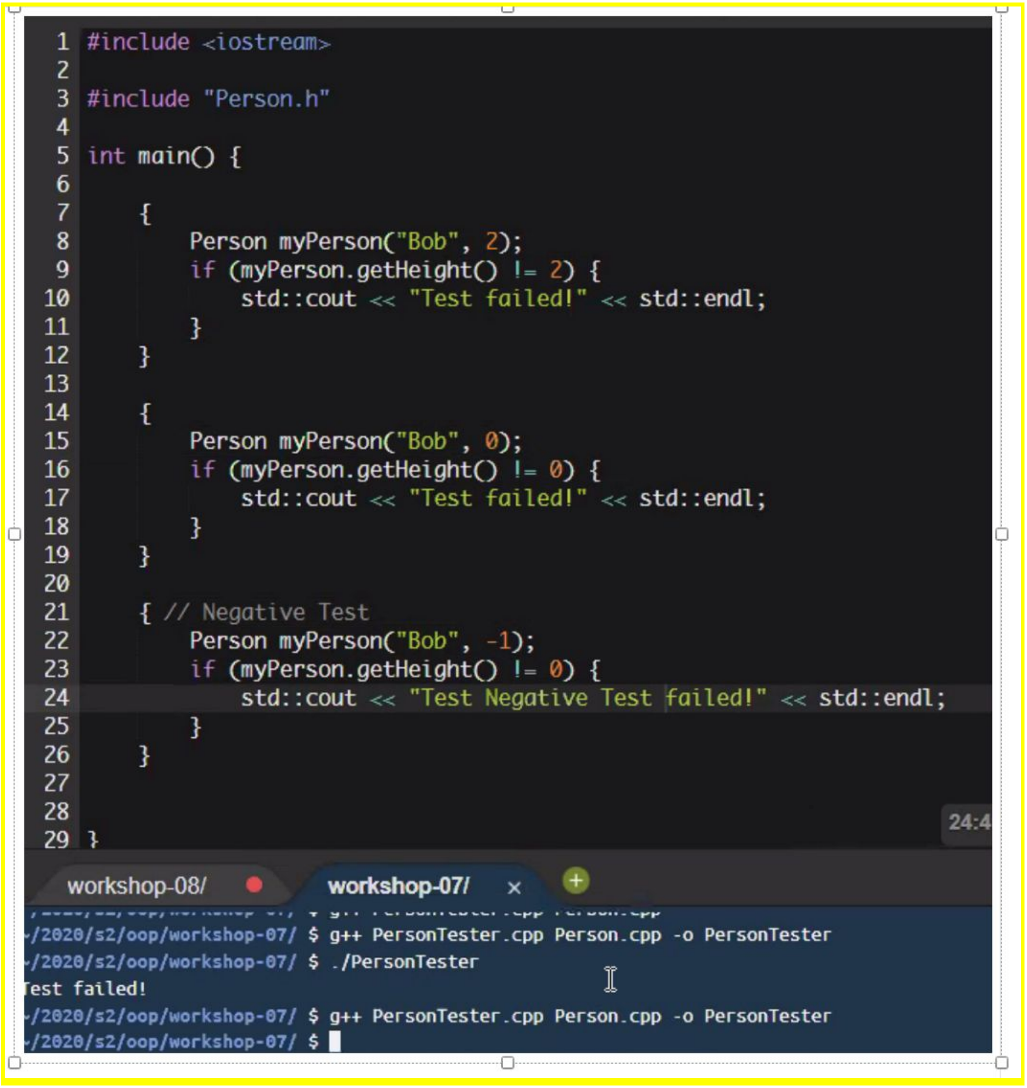
- **Integrated Testing**
  - Test the interaction between classes. i.e. any classes that require to communicate between classes to test the system functionality
  - Testing class dependencies - ensure code is rigid

Schedule Plan

Week	*Things to accomplish*
Break Week 1	1. Finish planning document (concurrent) ~~2. Create group on SVN~~ ~~3. Define implementations of **polymorphism, abstract classes and inheritance**~~ ~~4. Delegate code to write for each person~~ ~~5. Begin coding~~
Break Week 2	~~Tasks to complete (due: Friday, 2 Oct 2020)~~ ~~• Item, console and game classes (create makefile): Adam~~ ~~• Shop, customer and main class for UI: Antony~~

	<ul><li>~~Shop supervisor and supplier: Aniza~~</li><li>~~Discuss any changes/issues on Discord~~</li></ul>Meet again to discuss code<ol><li>~~Outline your plan for the next two weeks. (1 mark for each week's plan, to a total of 2.)~~</li><li>~~Show a Makefile that compiles the skeleton of your code. (1 mark)~~</li></ol>
Week 9	Tasks to complete<ol><li>~~Dynamic memory (stack + heap) - within function for multiple items i.e. customer buys console + game, customer loops through game/console selection in UI_main.cpp, etc. (all attempt this)~~</li><li>More testing of each function/class with multiple tests for different (3-4 tests per funct. - consider input validation)</li><li>~~Add Shop Supervisor and Supplier classes to end of UI_main.cpp to highlight their implementations after product(s) have been purchased~~</li><li>Consider/implement more examples of inheritance/polymorphism/abstract classes (see below)</li><li>~~Add more comments e.g. this function does this…~~</li><li>#ifndef, #define, #endif for all classes</li><li>Pointers? (possibly make object pointers)</li><li>~~Add destructors for consistency (Antony)~~</li><li>Distinguish between public, private and protected</li><li>~~Testing: unit and boundary tests - e.g. (post on Piazza about input validation)~~</li><li>~~Use of Makefiles - do we need a Makefile for each group member, one for each series of tests? (post on Piazza)~~</li><li>~~Unit, integration, regression and automated tests (how do we implement regression tests? - post on Piazza)~~</li><li>Planning doc: Update class diagram (for our reference) and UI explanation (to see changes made)</li><li>Update Google Drive with files as you go - use Discord for any issues</li><li>~~Store in SVN (before Week 9 prac session)~~</li></ol>~~Assessment:~~<ul><li>~~Present your idea. (1 mark)~~</li><li>~~Show a Makefile that compiles the skeleton of your code. (1 mark)~~</li><li>~~Outline your testing strategy (1 mark)~~</li><li>~~Outline your plan for the next two weeks. (1 mark for each week's plan, to a total of 2.)~~</li></ul>~~TOTAL POSSIBLE MARKS: 5~~
Week 10	Tasks to complete<ol><li>Dynamic memory (stack + heap) - within function for multiple items i.e. customer buys console + game, customer loops through game/console selection in UI_main.cpp, etc. (polish existing code to implement this feature)</li><li>Implement more examples of dynamic memory use, pointers, polymorphism and abstract classes by adding in the functionality suggested below.</li><li>Multiple shops (s1, s2) - one shop has more games and less consoles, one shop has no discounts, less stock, etc. (as a result: two shop supervisors)</li></ol>

	4. Multiple suppliers (supply1, supply2) - each supplier has unique state and behaviour. a. Abstract Class: Supplier → void sendStock(Item&) = 0 (works differently depending on which supplier) b. Class: Supply1 → only stocks consoles (maxStockCount = 400) i.e. only sends stock for consoles c. Class: Supply2 → only stocks games (maxStockCount = 1600) i.e. only sends stock for games 5. More item types e.g. accessories (not available at particular stores) 6. Add extra tests to updated customer class - Antony Assessment 1. Demonstrate an early version of your program showing at least two of the features listed. (2 marks) 2. Show that your code is inside the SVN system. (1 mark) 3. Be able to build it from your Makefile. (1 mark) 4. Be able to describe (and preferably demonstrate) two test situations. (2 marks) 5. Outline your plan for finishing work, with allocated tasks to the group. (1 mark for plan, 1 for allocation) 6. TOTAL POSSIBLE MARKS: 8.
Week 11	Tasks to complete and allocation of tasks to group members 1. Ensure all code has been exhaustively tested with unit, boundary, regression and automated testing - all group members 2. Check commenting, indentation, clarity and readability of code across all classes and files - Antony 3. Add latest code to one SVN system - compile and run on one computer - Adam 4. Update unit tests to reflect multiple shops and suppliers - all group members 5. Polish and refine supplier abstract class, supply1 and supply 2 classes to ensure they meet all tests - Aniza Final Assessment and Presentation of Project



```
#include <iostream>

#include "Person.h"

int main() {

    {
        Person myPerson("Bob", 2);
        if (myPerson.getHeight() != 2) {
            std::cout << "Test failed!" << std::endl;
        }
    }

    {
        Person myPerson("Bob", 0);
        if (myPerson.getHeight() != 0) {
            std::cout << "Test failed!" << std::endl;
        }
    }

    { // Negative Test
        Person myPerson("Bob", -1);
        if (myPerson.getHeight() != 0) {
            std::cout << "Test Negative Test failed!" << std::endl;
        }
    }


}
```

```
~/2020/s2/oop/workshop-07/ $ g++ PersonTester.cpp Person.cpp -o PersonTester
~/2020/s2/oop/workshop-07/ $ ./PersonTester
Test failed!
~/2020/s2/oop/workshop-07/ $ g++ PersonTester.cpp Person.cpp -o PersonTester
~/2020/s2/oop/workshop-07/ $ 
```

Don't write unit tests that break your program
Don't need to test getters and setters (test them anyway)
Unit = class e.g. CustomerTester.cpp, ShopTester.cpp
Test file is main file - TestMain.cpp for integration testing (running multiple classes at once)

Implementing dynamic memory

- Each item in the dynamic array corresponds to “game” or “console” - std::string* []
- Customer purchases game: addItem(std::string*, std::string itemType) - add to receipt price
- Initialise dynamic array in header file, constructor and UI_main.cpp
- Could add statement at end of UI_main.cpp showing receipt (summary of purchase details)
- The total price is calculated and updated based on the number of games/consoles purchased
- Int numGames and int numConsoles (increment in function as customer selects another game/console

Implementing more OOP concepts

- Multiple shops (s1, s2) - one shop has more games and less consoles, one shop has no discounts, less stock, etc.
- Multiple suppliers (supply1, supply2) - each supplier has unique state and behaviour e.g. one supplier only stocks consoles
- More item types e.g. accessories (not available at particular stores)