Business Simulator

Table of contents

Analysis

Introduction:

The purpose of this simulation is to help educate Business studies students, economics students and anyone else who is interested and give them some form of Idea of what it may be like to run their own business. I believe this would be beneficial to students (or anyone who wants to learn) as it allows them to explore the inner workings of a business and experiment with certain parameters and see how changing those parameters may affect the business. Allowing the player to understand how and why businesses act the way they do. This may also help students when answering exam questions since with the simulation they can apply what they have learned and recognise parts of the course in the game. Then when answering questions, they can use what they've taken from the game to help with their answer by using an application. It's also a fun game to play and could be a unique way to revise. The aim of the simulation is to either reach a certain amount of revenue/profit or just see what the highest amount you can make. The simulation involves the player buying and selling certain types of goods in a business. These include frozen goods, chilled goods, fresh goods and ambient/regular goods. The player will purchase x amount of a good at a price which is ever changing. Goods will all also be classed on their type based on elasticity, those being Inferior goods (Demand goes down when income goes up), normal goods(Demand up when income up) and luxury goods(high income elasticity). What this does is it shows the player of the game the concept of elasticities, allowing them to understand why the prices of goods are what they are. Furthermore the player will have a better understanding of the law of supply and demand. The economic formula to calculate the Income elasticity (YED) of a good is the %change in quantity demanded / %change in income. For my game however I have set the elasticity myself based on what type of good my products are. For example, wine would be a luxury good as when income goes up demand will go up.

For a good to be deemed income elastic, it must have a YED > 1. So for my game there will be Strong Elastic goods where the YED will be 1.8, meaning if income increased by 10%, demand would increase 1.8x, meaning demand would increase by 18%. And there are also strong inelastic goods. These have a YED of 0.5. Meaning if income increased by 10%, demand would only increase by 0.5x meaning demand will increase by 5%. As I stated previously, I have set the YED of these products myself based on what they're like in the real world. The simulation will also give the player an option to view a Profit/Loss sheet allowing them to get a better understanding of how their business is performing. As such, this will allow them to figure out what are bad actions and what are good actions.

User Installation Procedure:

In order for a user to have this running/working properly, the user will need to first download all the files from the NEAProtoSave file. Once this is done, the user should then change the connection string on line 23 to match the location where the file is saved on their own computer. For example, my connection string is:
@"Data Source=C:\Users\sampr\OneDrive\Desktop\KAB6 Comp Sci\Comp Sci NEA\NEAProtoSave\NEAProtoSave\Files\NEAdataBaseTest.db;Version=3;";.
The user will need to modify everything before "NEAProtoSave" to match the location of where they've stored the files. To do this simply, the user could go to the solution explorer and copy the file path of the NEAdataBaseTest file and paste it into the Connection string. Optionally, the user can also install DB Browser for SQLite. This isn't essential for the program to be run but it'll allow the user to be able to see their data. Once they've installed this, they can view the data by clicking on "Open Database", then going to where they've saved the NEAProtoSave folder, open the folder, open the files folder and then double click on the NEAdataBaseTest file.

Requirements:

The following are all requirements that I deem essential for my game to work as intended.

PreGame requirements:

- Pre1: Program should display the options "New Game", "Load Game" or "Quit"
- Pre2: If the user enters quit the program should display a thank you message and end
- Pre3: If the user enters new game, the program should prompt the user to make a username and password.
- Pre4: Once the user has entered a Username and password, they should be saved to the SQL database.
- Pre5: The user should then be sent to the setup phase.
- Pre6: If the user enters load game, prompt them to enter a username.
- Pre7: If the username is recognised, load the appropriate data from the SQL database (cash, goods and upgrades) and put the user in the setup phase
- Pre8: Else ,display an error message

Setup phase requirements:

- Set1: The program should let the user know they're in the setup phase
- Set2: Display the amount of cash the user has
- Set3: Display the amount of goods in fresh, frozen, chilled and regular storage.
- Set4: Display the options to Simulate, purchase goods, view storage, view and purchase upgrades and to save and quit.
- Set5: If the user enters Sim, they should then be sent to the simulation phase.
- Set6: If the user enters Purchase, begin the purchase phase
- Set7: If the user enters View, move them to the view storage phase
- Set8: If the user enters Upgrades, move them to the upgrades phase
- Set9: If the user enters Save, save their current data to the SQL database (Cash, goods, Upgrades)

- Set10: If the user enters quit, display a thank you message then end the game
- Set11: Display the option to view finances

Simulation phase requirements:

- Sim1: The program should let the user know they're in the simulation phase
- Sim2: Display how many units of each good was sold.
- Sim3: Display the amount each good was sold for.
- Sim4: Display the total amount of revenue the good made
- Sim5: Display the amount of cash the user has (After sales)
- Sim6: Display the amount of units in each storage
- Sim7: Tell the user if any goods expired and how many of each
- Sim8: Prompt the user to begin the next cycle
- Sim9: If it is a 4th cycle, force the user to pay £500 toward bills
- Sim10: Deduce £500 from the user's current cash.
- Sim11: If when the game attempts to collect the bills the user cannot afford it, tell the user they cannot afford to pay bills and as such the game will end
- Sim12: End the game if the user cannot pay bills.
- Sim13: Every simulation, increase or decrease the price of goods
- Sim14: Every simulation, store the amount of inflows and outflows of the business

Purchase phase requirements:

- Pur1: The program should let the user know they're in the purchase phase
- Pur2: Prompt the user to enter the type of goods they'd like to purchase
- Pur3: Display the types of goods available
- Pur4: Display the option to go back
- Pur5: If the user enters back, send them back to the setup phase
- Pur6: If the user enters an invalid option, prompt them to try again
- Pur7: When the user enters a valid storage type, display the goods in that storage type (e.g. if user enters frozen display the frozen goods)

- Pur8: When the user enters a valid storage type, move them to that storage area
- Pur9: Display the names of the goods in that storage type
- Pur10: Display the current prices of the goods in that storage type
- Pur11: If the user has the elasticity insight upgrade, display whether the good is elastic or inelastic.
- Pur12: Prompt the user to enter the name of the good they'd like to buy
- Pur13: If the name is invalid, display and error message then prompt them to try again
- Pur14: If the user entered a valid good, prompt them to enter the amount of the good they'd like to buy
- Pur15: If the input is invalid, display and error message then make the user pick which good they want to purchase again
- Pur16: If the user enters a valid quantity, display a success message
- Pur17: Then prompt the user to enter the selling price
- Pur18: If the input is invalid, display and error message then make the user pick which good they want to purchase again
- Pur19 If they enter a valid selling price display a success message and then a final success message showing the price and quantity.
- Pur20: If the user has entered everything correctly but they do not have the storage or quantity, display an error message then prompt them to try again.
- Pur21: If the user does not want to purchase the good, give them the option to go back
- Pur22: If the user enters back, return them to the start of the purchase phase.

View storage phase requirements:

- Sto1: The program should let the user know they're in the view storage phase
- Sto2: Program should display all the different types of storages
- Sto3: Program should display the amount of units in each storage
- Sto4: Prompt the user to enter the name of the storage they want to view

- Sto5: Give the user the option to go back
- Sto6: When the user enters back, send them back to the setup phase
- Sto7: When the user enters a storage type, display the goods in that storage
- Sto8: Program should display: the name of the good, quantity held and selling price
- Sto9: Prompt the user to enter any key to continue
- Sto10: Ensure the user can only hold the maximum amount of goods possible (e.g. if cap is 100 they shouldn't be able to hold >100 goods of that category)

Upgrades phase requirements:

- Upg1: The program should let the user know they're in the upgrades phase
- Upg2: Display the name of available upgrades
- Upg3: Display the price of the upgrades
- Upg4: Display a brief description of what the upgrade does
- Upg5: Have a way of selecting and purchasing the upgrades
- Upg6: Have a way of going back
- Upg7: Display a message once the user has purchased the upgrade
- Upg8: Prevent the user from purchasing an upgrade they already own
    - As in prevent the user from wasting money
- Upg9: Display a message if the user tries to buy the same upgrade again
- Upg10: Apply the upgrade successfully

Finance requirements:

- Fin1: Upon entering the finance menu, user should have the options to view current P/L sheet, previous P/L sheet or a grand total P/L sheet
- Fin2: Upon entering the finance menu, user should have the option to go back
- Fin3: If the user enters C on week 1, display a message to let the user know there's no current data

- Fin4: If the user enters C on any other week display the profit/loss sheet for that week
- Fin5: The sheet should show the week number
- Fin6: The sheet should show sales revenue
- Fin7: The sheet should show expenditures (Bills, Cost of goods, Upgrades)
- Fin8: Tell the user how much profit they made for the week.
- Fin9: Display whether or not they made a profit (if not display loss)
- Fin10: Tell the user if they made more or less profit than the previous week
- Fin11: If the user enters P on week 1, display error message
- Fin12: Else, display the previous p/l sheet
- Fin13: If the user enters T on week 1, display an error message
- Fin14: The grand P/L sheet should display the total revenue the user has made since starting
- Fin15: Should also display all total expenditures since starting
- Fin16: Display the net income for the business
- Fin17: Tell the user if the business has overall been profitable or not.
- Fin18: When the user is done, they can press the enter button to go back to the setup phase.

Aspects within the game

Goods & Storages:

Within the game, there will be multiple different aspects which contribute to the completion of the game. The first of which would be the storage. The higher the storage the more of an item you can hold. There will be multiple different types of storage which all correlate with the different types of products. The first type of storage and products are frozen. Frozen goods range from ice-creams to pizzas and fish. The main benefit of frozen products is that they are long lasting. The next type of storage and products are fresh products. This would be the fruits, vegetables and dips/sauces. Main benefit of fresh goods is that they sell quickly. Following this, there is then chilled goods. This would be stuff like meat, juices or milk. Main benefit of chilled goods is the wide variety. The final type of goods are ambient/regular goods. These would be stuff like sweets, biscuits etc. Good

thing about these is that they're easy to store and stock. As mentioned in the introduction, goods are also assigned levels of Income elasticity.

Cash:

Within the game, the player will have a cash balance. Cash is integral. It's needed to buy goods, obtain upgrades and pay bills. The only way to earn cash will be from sales of goods. And as mentioned, cash can be spent on goods to sell, upgrades to improve your business or every 4 weeks at the end of the month on bills.

Upgrades:

Within the game, the player will have the options to buy upgrades. In order to purchase an upgrade you will need the correct amount of money and to have completed all prior prerequisites. There will be your basic upgrades like more storage space and shelf space. For example, in order to obtain "Frozen Upgrade II", you must first own "Frozen Upgrade I" and have sold 2000 total frozen goods. There will also be passive upgrades that can help you in other ways. For example "Money trees" will allow the player to earn a flat out 10% bonus on sales. Or "Experienced Entrepreneur" will allow the player to see elasticities of goods.

Example playthrough:

Player enter's New game, enters a name and password and is put into the Setup Phase. Player enters P to purchase goods, then F to purchase frozen goods. They buy 10 turkeys for £12 each and sell them for £15 each.
They then enter sim and all 10 sell. The user then enters Save and Quits. This would be the most basic/barebones level of a playthrough

Research

Who is this for?

Ultimately, this can be used by anyone. However the main people it is intended for are students who study business or economics. This would allow those students to experience/ get an idea of some of the concepts the would've been taught in their lessons and apply them in a suitable environment. Teachers could use this simulator to help showcase what they have taught and see how it is actually used in a business. Or it could be also used in some form of homework. As well as this, this can be used for people who just want to pass time/have fun.

Technology required:

I will be doing all the coding on my own personal laptop. The coding will be done in Microsoft Visual Studio in C#. In order to save and load data, I will be using SQLite and DBrowserForSQLite.

Research conducted:

A profit and loss statement (P&L), or income statement or statement of operations, is a financial report that provides a summary of a company's revenues, expenses, and profits/losses over a given period of time. The P&L statement shows a company's ability to generate sales, manage expenses, and create profits.

Investopedia:

Link: https://www.investopedia.com/terms/b/balancesheet.asp Date Visited 11.7.24 What the website gives: Explains the purpose of a balance sheet and the importance of having one Takeaways:

- A balance sheet is a financial statement that reports a company's assets, liabilities, and shareholder equity.
- The balance sheet is one of the three core financial statements that are used to evaluate a business.
- It provides a snapshot of a company's finances (what it owns and owes) as of the date of publication.

- The balance sheet adheres to an equation that equates assets with the sum of liabilities and shareholder equity.
- Fundamental analysts use balance sheets to calculate financial ratios.

Forbes

Link: https://www.forbes.com/advisor/business/balance-sheet/ Date Visited: 11.7.24 What the website gives: Explain the components of a balance sheet Takeaways:

On hand assets:

- Liquid assets: Cash and cash equivalents, such as certificates of deposit (CDs)
- Accounts receivable (A/R): Money owed to your company
- Marketable securities: Liquid assets that are readily convertible into cash (generally reported under cash and cash equivalents)
- Inventory: Any products you have available for sale
- Prepaid expenses: Rent, insurance and contracts with vendors

Long term assets:

- Investments or securities that can't be liquidated within the next year
- Fixed assets: Land, machinery and buildings
- Intangible assets: Intellectual property, brand awareness and company reputation

Shareholder Equity

- Money generated by a company
- Money put into the business by its owners and shareholders
- Any other capital put into the business

Corporate Finance Institute:

Link: https://corporatefinanceinstitute.com/resources/accounting/balance-sheet/ Date Visited: 11.7.24 What the website gives: A walkthrough of an example balance sheet Takeaways: The balance sheet on the website.

Wise

Link: https://wise.com/us/income-statement/profit-loss-statement Date Visited: 11.7.24 What the website gives: Quick explanation of a profit loss sheet and a model sheet Takeaways: The model sheet

NetSuite

Link: https://www.netsuite.co.uk/portal/uk/resource/articles/accounting/profit-an

d-loss-statement.shtml Date Visited: 11.7.24 What the website gives: An in depth explanation of the profit/loss sheet Takeaways:

- A profit and loss statement includes a business's total revenue, expenses, gains, and losses, arriving at net income for a specific accounting period.
- Management analyses a P&L to determine how to increase profitability by increasing revenue, lowering costs or both. A P&L is also a useful tool for lenders and investors that are evaluating a business for a loan or investment.
- A profit and loss statement is prepared using one of two accounting methods: cash or accrual.

Paychex

Link: https://www.paychex.com/articles/finance/how-to-create-a-profit-and-loss-statement-for-small-businesses Date Visited: 11.7.24 What the website gives: An example of a PL sheet and how to use one Takeaways: The example

Design:

My program is object oriented and makes use of 8 classes to ensure the game runs as intended without any game breaking bugs or issues.

Program Class & Databaseconfig class

Program is used to essentially start the game. Database Config creates a connection string. This string is used as a connection between the program and the SQL database. So whenever my program wants to read or write data to the database, it used the ConnectionString constant to know/find the location of the .db file containing the database

Game Class:

Is the biggest of all the classes and where the bulk of the mechanics take place. An explanation of what all the methods do are as follows(when I say

in-memory I am referring to the actual program memory as opposed to what's stored in the DB):

EnsureUsersTablesExists():

Creates the four SQLite tables (Users, Goods, Storage, Upgrades) if they don't already exist. Called at the start of a new game to make sure persistence is ready.

AddGoodsToStorage(string UserName, int Good_Id, string ProductName, int GoodType, int Quantity, decimal SellingPrice, int CyclePurchased):

Inserts a purchased item into the Storage DB table for that user.
Used anytime the user buys goods (in PurchaseGoods) or when saving the game to persist current inventory.

UpgradesMenu():

Displays the list of available upgrades, reads the player's choice, and dispatches to purchase one. Invoked from the Setup Phase when the player types "U" to buy or view upgrades.

SaveUpgrades(string userName, string upgradeName):

Inserts (or ignores if the player already owns the upgrade) a purchased upgrade into the Upgrades DB table. Called immediately after an upgrade is bought to ensure that choice is applied and constant.

List<string> LoadUpgrades(string userName):

Fetches all upgrade names the user has already bought from the DB. Run when the game is loaded to re-apply any past upgrades.

ApplyUpgrades(List<string> loadedupgrades)

Looks up each loaded upgrade in the players playthrough list and applies its effect to the Store. Ensures that saved upgrades actually modify gameplay after loading

Game():

Initializes the store (with £1,000), market (pointing at the DB), cycle counter, and the in-game list of possible upgrades. Sets up all core objects before showing the main menu.

InitialiseUpgrades():

Fills the availableUpgrades list with each upgrade's name, cost, description, and effect delegate. Called once in Game so the program knows what upgrades exist.

PurchasedUpgrades(Upgrades upgrade):

Checks if the player already owns this upgrade, whether they can afford it, deducts cash, records the expense, applies the effect, and saves it. Invoked from UpgradesMenu() when the player pricks one to buy.

MainMenu():

Shows the "New Game / Load Game / Quit" screen, reads player's choice, and hands off to the appropriate setup or load logic. The very first screen the player sees when the program is started.

SetupNewPlayer()

Asks for a business name and password, creates a new user record via CreateNewPlayer, and then calls Start(). Follows from "New Game" in the main menu.

Bool CreateNewPlayer(string UserName, string Password):

Inserts a new row into Users (username + password + initial cash). Returns success or failure (e.g. duplicate name). Underpins SetupNewPlayer to actually register the player into the DB.

Start():

The core game loop: increments the cycle count, runs SetupPhase(), pays monthly bills every 4 weeks, checks for expired goods, runs

SimulationPhase(), logs that week's finances, then repeats. Once the player either starts a new game or loads an old one, this drives week-by-week progression until they run out of money or quit.

SetupPhase():

Presents options ("Sim", "Purchase", "View", "Upgrades", "Finance", "Save", "Quit") and routes to the correct sub-routine or returns to start the simulation. Called at the top of every cycle from Start().

SaveGame():

Updates the user's cash in Users, clears out their old Storage entries, then re-inserts every item currently in memory. Can be called from the setup menu ("Save").

LoadGame(string userName):

Reads the user's cash from Users; if found, rebuilds their Store (cash + username), calls LoadPlayerGoods, then LoadUpgrades + ApplyUpgrades, and finally hands over to Start(). Follows from "Load Game" on the main menu.

LoadPlayerGoods(string userName):

Joins Storage with Goods in the DB to pull every stored product (name, quantity, price, type, cycle added) and re-creates them in the in-memory store. Ensures player's on-hand inventory is restored when loading a game

PurchasePhase():

Lets the player choose which category of goods to buy (Fresh/Chilled/Frozen/Regular), then calls PurchaseGoods for that category. Accessed via "P" in the setup menu.

PurchaseGoods(string category)

Lists market prices for every item in that category (showing elasticity if the player has the upgrade), prompts for name, quantity, and chosen selling price, then attempts the purchase and calls AddGoodsToStorage.
Handles the entire buy flow: display → input → cash/storage check → persistence.

ViewStoragePhase():

Displays how full each storage is, lets you pick one to inspect contents, and (work-in-progress) offers to remove items. Accessible via "V" in setup; lets the player audit and manage inventory.

SimulationPhase():

Clears the console, simulates a week's worth of sales on every item via Store.SimulateSales (printing units sold and revenue), updates cash, then calls market.UpdateMarketPrice() to randomize next week's prices. Called after the player finishes the setup phase each cycle.

FinanceMenu():

Offers "Current", "Previous", or "Total" Profit/Loss reports and routes to the corresponding sheet-display methods. Accessible via "F" in the setup phase.

DisplayPortfolio(int weekIndex):

Prints a detailed P&L report for an arbitrary week, including revenue, each expense line, net income, and % change vs. the prior week. Used by both PreviousSheets() and by selecting a specific week.

CurrentSheet():

Shows just the latest week's P&L summary, highlights profit vs. loss, and compares to the previous week's net income. Called when the player picks "Current" in FinanceMenu().

PreviousSheets():

If at least two weeks have passed, invokes DisplayPortfolio for the week before last; otherwise reports “No previous data.” Handles “Previous” in FinanceMenu().

TotalSheet():

Aggregates all weeks’ sales, purchases, bills, and upgrade expenses into grand totals, computes overall net income, and prints whether the player is profitable overall. Invoked for “Total” P&L in the finance menu.

StoreClass:

The Store class represents the player’s business, it’s the core of their in-game identity and manages all the operational aspects of running the store. is what makes the simulation interactive and strategic. While the Game class handles the flow, but the Store handles the mechanics, buying, selling, storing, and growing the business. As such, whatIt is responsible for is as follows:

- Tracking the player’s money (Cash)
- Managing inventory across four storage types (Fresh, Chilled, Frozen, Regular)
- Handling purchases and sales (e.g., can they afford a product? Do they have space?)
- Applying upgrades (like sales boosts or market insights)
- Managing time-sensitive goods (e.g., removing expired chilled products)
- Handling storage limits to enforce strategic decisions
- Recording ownership of upgrades to modify gameplay effects

And this is a brief explanation of what each method does:

Store(decimal initialCash, string userName)

Initializes a new store for the player, sets starting cash and username. creates empty lists for each storage type (Fresh, Chilled, Frozen, Regular). prepares

for any upgrades the player may buy. Called when the player starts a new game or load one, establishing the player's in-memory state.

AddProductToStorage(StorageType storageType, Product product):

Adds a Product object to the appropriate storage list in memory.
Used during game-load to rehydrate inventory and whenever the player buy goods (so that the in-memory store matches the DB).

AdjustCash(decimal Amount):

Adds (or subtracts, if negative) the given amount from Cash.
Used by upgrade effects or other routines that need to tweak the player's balance outside of buying/selling logic.

AddUpgrade(string upgradeName):

Records a purchased upgrade name in the ownedUpgrades set.
Called immediately after the player pays for an upgrade so subsequent checks (e.g. in purchase, display) know they own it.

Bool HasUpgrade(string upgradeName):

Returns whether the given upgrade is already owned.
Checks in multiple places: To prevent duplicate purchases in PurchasedUpgrades To decide whether to show elasticity info in PurchaseGoods.

Bool BuyProduct(Product product):

Attempts to buy the given product:

1. Verifies there's enough cash (Cash ≥ cost).
2. Verifies there's enough free storage (currentQty + product.Quantity ≤ MaxStorageCapacity).
3. If both pass, deducts cash and adds the product to the in-memory storage.

Drives the core purchase logic in Game.PurchaseGoods; returns success/failure for user feedback.

Int GetCurrentStorageQuantity(StorageType storageType):

Sums the quantities of products in the specified storage area.
Displays how full each storage is, and to enforce capacity when buying.

Decimal SimulateSales(Market market):

Iterates every product in every storage:

1. Calls market.SimulateProductSales(product) to get units sold.
2. Computes revenue (units × selling price), adds it to Cash, reduces inventory.
3. Logs each sale in the console.
   Core of the sales phase (Game.SimulationPhase), returns total revenue to record in finances.

DisplayCash():

Prints the current cash balance in green if ≥ £500, red if lower.
Called at key points (e.g. before purchases) to give the player an idea of their liquidity.

DisplayStatus():

Prints cash plus the fill-level of each storage type.
Shown every cycle in the setup phase (Game.SetupPhase), so players can decide what to buy or sell.

DiplsayStorageStatus():

Prints “Fresh: x/100 units”, etc., for each storage. Used in the “View Storage” menu (Game.ViewStoragePhase) to overview inventory levels.

DisplayStorage(StorageType storageType):

Lists every product in the chosen storage with quantity and selling price.

Drilled-into view when the player picks a specific storage type in the "View Storage" flow.

Bool PayBills(decimal amount)

If Cash ≥ amount, subtracts that amount, logs payment, and returns true; otherwise false. Called every four cycles in Game.Start to enforce fixed monthly costs and potentially trigger game over.

CheckForExpiredGoods(int currentCycle):

Scans chilled storage for any product that's been there ≥ 2 cycles, removes it, and notifies the player. Called right after bills in the monthly check, so expired goods can't be sold in subsequent simulations.

Product Class:

Acts as a simple data container that represents each individual good in the player's inventory. It holds all the key attributes the game needs in order to:

- Track what the item is (Name)
- Know how much it cost to buy (PurchasePrice) and how much it will sell for (SellingPrice)
- Record how many units you have (Quantity)
- Know where it belongs in storage (StorageType)
- Remember when it was stocked (CycleAdded) so perishables can expire
- Store its price-elasticity (Elasticity) for sales simulation logic .

Market Class:

The market class represents the SQLite DB which the in-memory store interacts with. It's responsible for:

- Initialising market data by loading every product's base purchase price from the SQLite Goods table
- Providing price information to both buying (so the player pays half the market price) and selling (to compare their selling price against "true" market price)
- Simulating dynamic market behavior via random price fluctuations each cycle

- Drives sales logic by calculating how many units of each product sell, based on price-elasticity and the relationship between the selling price and the market price
- Categorising goods into storage types and flagging whether they're elastic or inelastic when you have the right upgrade .

Market(string ConnectionString):

Saves the DB connection string, initializes the in-memory marketPrices dictionary, and immediately calls LoadGoodsFromDatabase().
When starting or loading a game, this status up all known products and their initial prices.

LoadGoodsFromDatabase():

Simply executes "SELECT GoodName, PurchasePrice FROM Goods;
And then fills marketPrices[GoodName] = PurchasePrice for each row.
Fills market with every product from DB so player can buy/sell them.

GetGoodId(string productName):

Similarly, it executes "SELECT Good_Id FROM Goods WHERE GoodName = @GoodName;
To look up the database ID for persistence.

Decimal GetMarketPrice(string productName):

Looks up the current price in marketPrices, rounds it to two decimal places. Shown to the player during purchase (PurchaseGoods) and used internally in SimulateProductSales to compare against their selling price.

UpdateMarketPrice():

Iterates every key in marketPrices, applies a random ±£0.01–0.50 change, and clamps it ≥ £0.01 so prices stay positive. Called at the end of each sim cycle.

Int SimulateProductSales(Product product):

1. Fetch the marketPrice for that product.

2. Computes a priceFactor = marketPrice / product.SellingPrice.
3. Estimates base sales = Quantity × priceFactor.
4. Adjusts for elasticity (StrongElastic, WeakElastic, WeakInelastic, StrongInelastic) with caps or fixed percentages.
5. Returns the lesser of that estimate and available Quantity.

Drives every sale in Store.SimulateSales, determining units sold and revenue added to the player's cash.

Dictionary<string, decimal> GetGoodsByCategory(string category):

Filters marketPrices by comparing GetStorageType(productName) to the given category (fresh/chilled/frozen/regular) and returns matching name and price. Used in Game.PurchaseGoods to display only the items you can buy for the category the player chose.

StorageType GetStorageType(string productName):

Returns a StorageType enum based on the switch of common product names. ENsures both purchase and load routines put items into the correct in-memory shelf

Bool IsElastic(string productName):

Returns true if the product is considered elastic based on a switch.

WeeklyFinances Class:

A simple data container that records everything needed for a single week's profit & loss statement. Each cycle through the game loop creates one WeeklyFinance instance, which then gets stored in the game's weeklyFinances list. Later, the various finance-report menus pull from these objects to show current, previous, or total P&L sheets

Int Week { get; set; }:

The week number this record corresponds to. Used in reports to label the week the player is viewing.

Decimal SalesRevenue { get; set; }:

Total income from sales during that week. Filled in Game.Start() from the accumulated currentWeekSalesRevenue. Shown under “Revenue” in every P&L report.

Decimal PurchaseExpenses{ get; set; }:

Total cost of goods bought that week, taken from currentWeekPurchaseExpenses in Game.Start(). Listed under “Expenditures” in weekly and grand-total sheets.

Decimal BillsExpenses { get; set; }

Sum paid toward the fixed £500 bills every 4 weeks in that cycle, comes from currentWeekBillsExpenses. Appears in the expenses section on the P&L sheet.

Decimal UpgradeExpenses { get; set; }:

Total spent on any upgrades that week.

Decimal NetIncome => SalesRevenue - (PurchaseExpenses + BillsExpenses + UpgradeExpenses);

A read only property that subtracts all expenses from revenues.

Upgrades Class:

Used to model a single purchasable upgrade.
What the upgrade is (Name)
Cost (Price)
What it does (Description)
What it actually does (Effect)
By centralizing these four elements, the Game class can treat every upgrade uniformly, displaying it, checking affordability, persisting ownership, and applying its in-game effect when purchased or re-loaded.

SQLite:

My program also utilises SQLite, in accordance with CRUD:

Create:

It creates tables if they don't already exist containing the goods, storages and users. As well as creating a new user in a new game run.

Retrieve:

Whenever the user is loading a new game, it will need to retrieve the data from the DB.

Update:

Similarly, whenever the user saves their game, it will need to update the current information in the database for the user with the in-memory information when the user enters save.

Delete:

If the user wishes to remove goods from their storage it will also remove it from the DB.

SQL to ensure tables exist:

The following SQL queries are used to make sure the tables required are all present. As without them the program will not be able to function fully. If the table already exists then nothing will happen but if for some reason the table doesn't exist, it will create them.

Users Table:

```
CREATE TABLE IF NOT EXISTS Users
(
    Id INTEGER PRIMARY KEY AUTOINCREMENT,
    Username TEXT NOT NULL UNIQUE,
    Password TEXT NOT NULL,  Cash REAL
)
```

Goods Table:

```
CREATE TABLE IF NOT EXISTS Goods
(
```

```
    Good_Id INTEGER PRIMARY KEY,
    GoodName TEXT NOT NULL,
    PurchasePrice REAL NOT NULL,
    GoodType INT NOT NULL,
    CycleExpires INT NOT NULL
)
```

Storage Table:

```
CREATE TABLE IF NOT EXISTS Storage
(
    Storage_Id INTEGER PRIMARY KEY,
    UserName TEXT NOT NULL,
    GoodName TEXT NOT NULL,
    Good_Id INT NOT NULL,
    Quantity INT NOT NULL,
    SellingPrice REAL NOT NULL,
    CyclePurchased INT NOT NULL,
    GoodType INT NOT NULL, --1 = Chilled, 2 = Fresh, etc.
    FOREIGN KEY(Good_Id) REFERENCES Goods(Good_Id)
)
```

Upgrades Table:

```
  CREATE TABLE IF NOT EXISTS Upgrades
  (
    UpgradeId INTEGER PRIMARY KEY AUTOINCREMENT,
    UserName TEXT NOT NULL,
    UpgradeName TEXT NOT NULL,
    UNIQUE(UserName, UpgradeName)
```

)

Inserting Data

The following SQL/code is used to save/insert data into the database table and load/retrieve data from the database. I decided that the best way for data to be saved for this project was by using a connection string to create a direct path to the database and where the data will be saved. This was done by using:

```
public const string ConnectionString
 = @"Data Source=C:\\Users\\sampr\\OneDrive\\Desktop\\KAB6 Comp
Sci\\Comp Sci NEA\\NEAProtoSave\\NEAProtoSave\\Files\\
NEAdataBaseTest.db;Version=3;";
```

To act as a way to open a file and read/write to the file

AddGoodsToStorage:

```
INSERT INTO Storage (UserName, Good_Id, GoodName, Quantity,
SellingPrice, CyclePurchased, GoodType)
VALUES (@UserName, @Good_Id, @GoodName, @Quantity, @SellingPrice,
@CyclePurchased, @GoodType);";

using (SQLiteConnection conn = new
SQLiteConnection(DataBaseConfig.ConnectionString))
{
    conn.Open();
    using (SQLiteCommand cmd = new SQLiteCommand(insertSQL, conn))
    {
        cmd.Parameters.AddWithValue("@UserName", UserName);
        cmd.Parameters.AddWithValue("@Good_Id", Good_Id);
        cmd.Parameters.AddWithValue("@GoodName", ProductName);
        cmd.Parameters.AddWithValue("@Quantity", Quantity);
        cmd.Parameters.AddWithValue("@SellingPrice", SellingPrice);
        cmd.Parameters.AddWithValue("@CyclePurchased", CyclePurchased);
        cmd.Parameters.AddWithValue("@GoodType", GoodType);
```

```
        cmd.ExecuteNonQuery();
    }
```

CreateNewPlayer:

```
string sql = "INSERT INTO Users (Username, Password, Cash) VALUES
(@UserName, @Password, @Cash);";
```

Saving Data

Save Upgrades

```
string SQL = @"
INSERT OR IGNORE INTO Upgrades (UserName, UpgradeName)
VALUES (@UserName, @UpgradeName);";

using (SQLiteConnection conn = new
SQLiteConnection(DataBaseConfig.ConnectionString))
{
    conn.Open();
    using (SQLiteCommand cmd = new SQLiteCommand(SQL, conn))
    {
        cmd.Parameters.AddWithValue("@UserName", userName);
        cmd.Parameters.AddWithValue("@UpgradeName", upgradeName);
        cmd.ExecuteNonQuery();
    }
}
```

Save Game:

```
UPDATE Users SET Cash = @Cash WHERE UserName = @UserName;

string ClearStorageSQL = "DELETE FROM Storage WHERE UserName =
@UserName;"; // clears the storage table and adds the new/updated
data(goods)
```

Removing Data

RemoveGoodsFromStorage:

This SQL checks that the goods are actually there and able to be removed/valid:

```
string checkSQL = @"
SELECT Quantity FROM Storage WHERE UserName = @UserName AND
Good_Id = @Good_Id AND GoodName = @GoodName
AND SellingPrice = @SellingPrice AND CyclePurchased = @CyclePurchased
AND GoodType = @GoodType;";
 int currentQuantity = 0;
```

This SQL then removes all the goods if the user wants them all removed:

```
string RemoveSQL = @"
DELETE FROM Storage WHERE UserName = @UserName AND Good_Id =
@Good_Id AND GoodName = @GoodName
AND SellingPrice = @SellingPrice AND CyclePurchased = @CyclePurchased
AND GoodType = @GoodType;";
```

And then this SQL only removes a certain amount, e.g. if the player has 50 milk and they just want to remove 10:

```
int newQuantity = currentQuantity - QuantityRemove;
   string UpdateSQL = @" UPDATE Storage SET Quantity = @Quantity
WHERE UserName = @UserName
   AND Good_Id = @Good_Id AND GoodName = @GoodName AND
SellingPrice = @SellingPrice
   AND CyclePurchased = @CyclePurchased AND GoodType =
@GoodType;";
```

Retrieving data:

LoadUpgrades:

```
string SQL = "SELECT UpgradeName FROM Upgrades WHERE UserName = @UserName;";
```

LoadGame:

```
string sql = "SELECT Cash FROM Users WHERE LOWER(Username) = LOWER(@Username);";
```

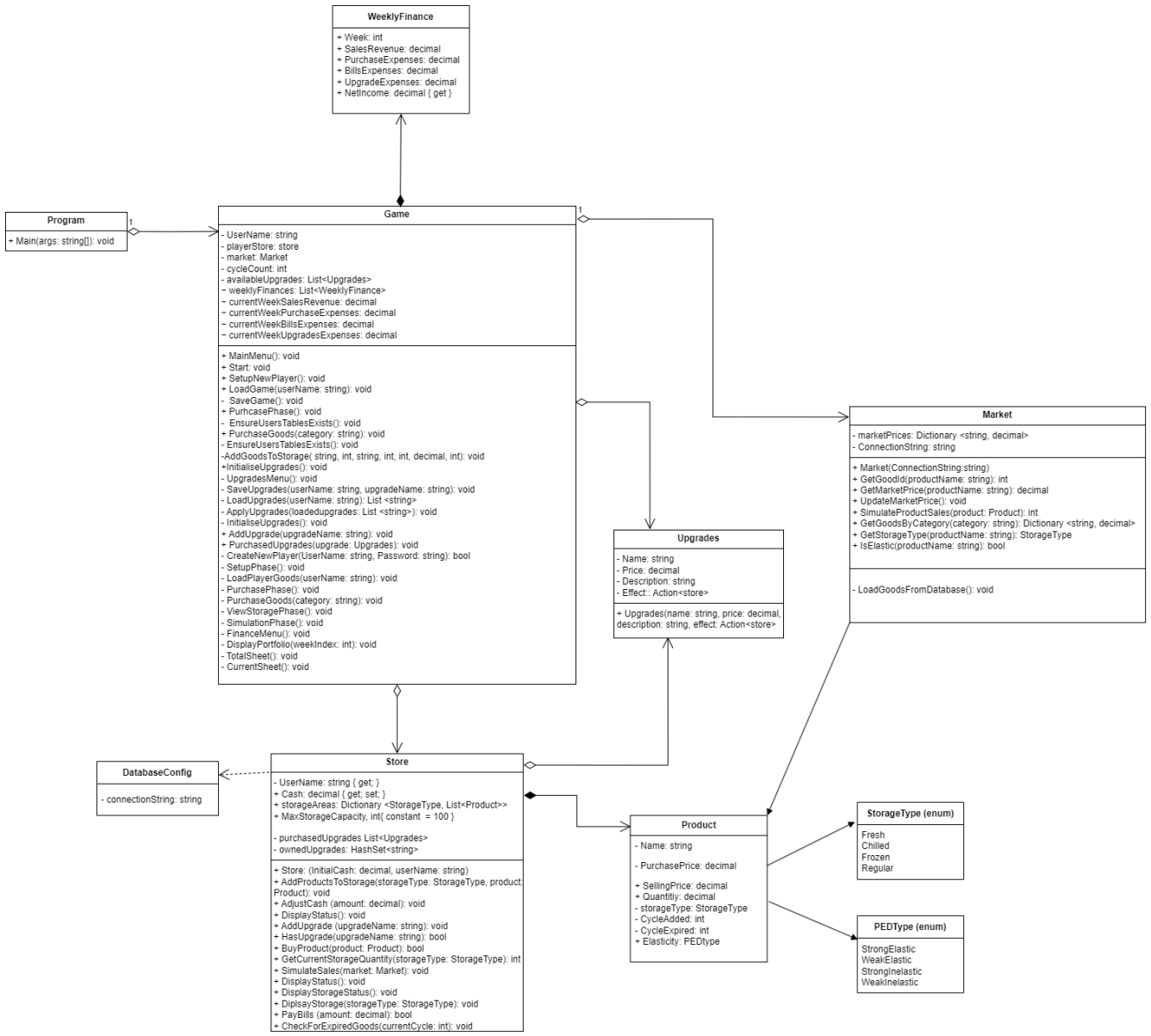
LoadPlayerGoods:

```
string sql = @" SELECT g.GoodName, s.Quantity, s.SellingPrice, s.GoodType, s.CyclePurchased
FROM Storage s
INNER JOIN Goods g ON s.Good_Id = g.Good_Id
WHERE s.UserName = @UserName;";
```

Images:

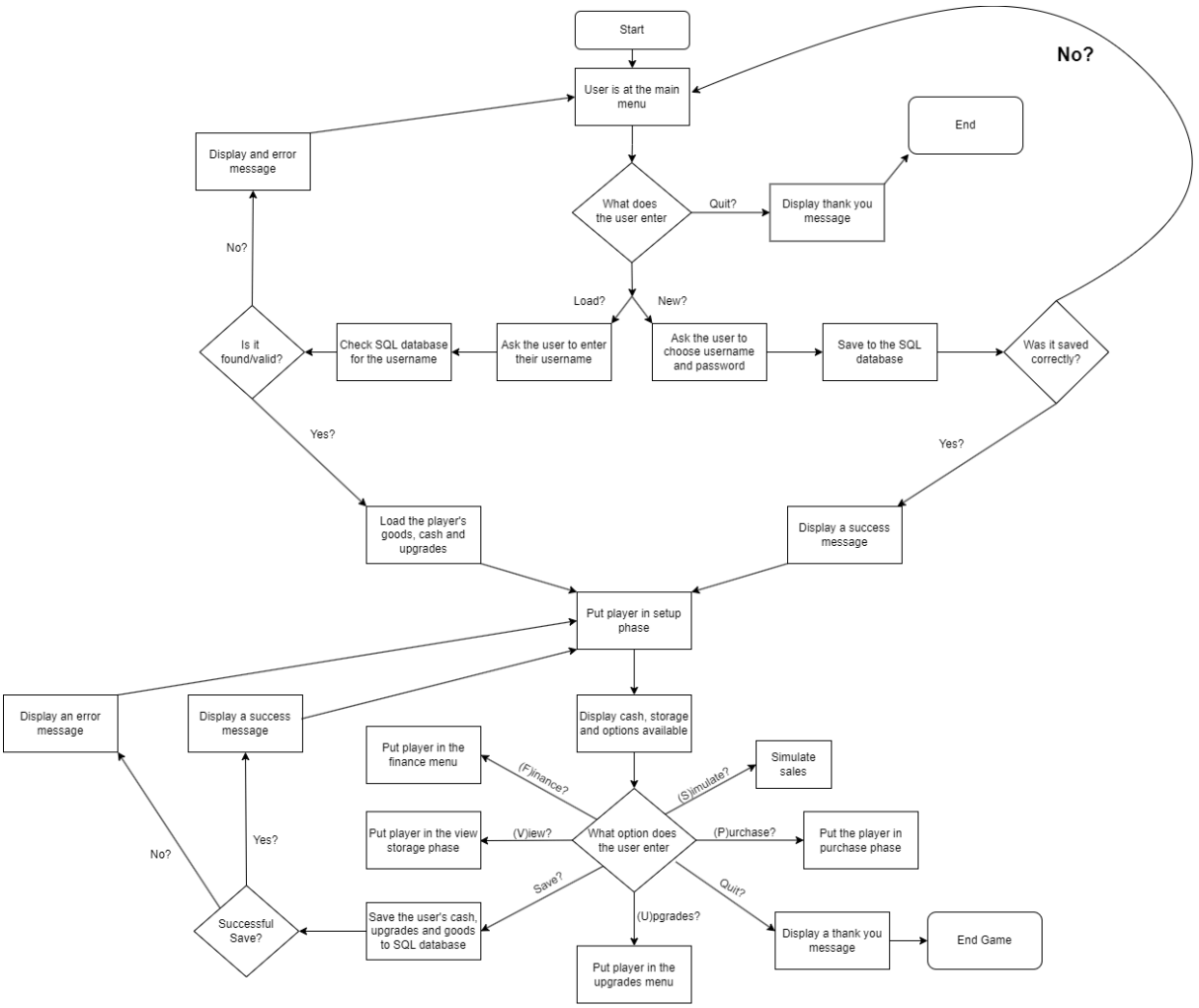
Class Diagram:

Below is a class diagram showcasing the classes in my program, meanwhile the flowcharts showcase the main stages of the game and what happens during these stages.



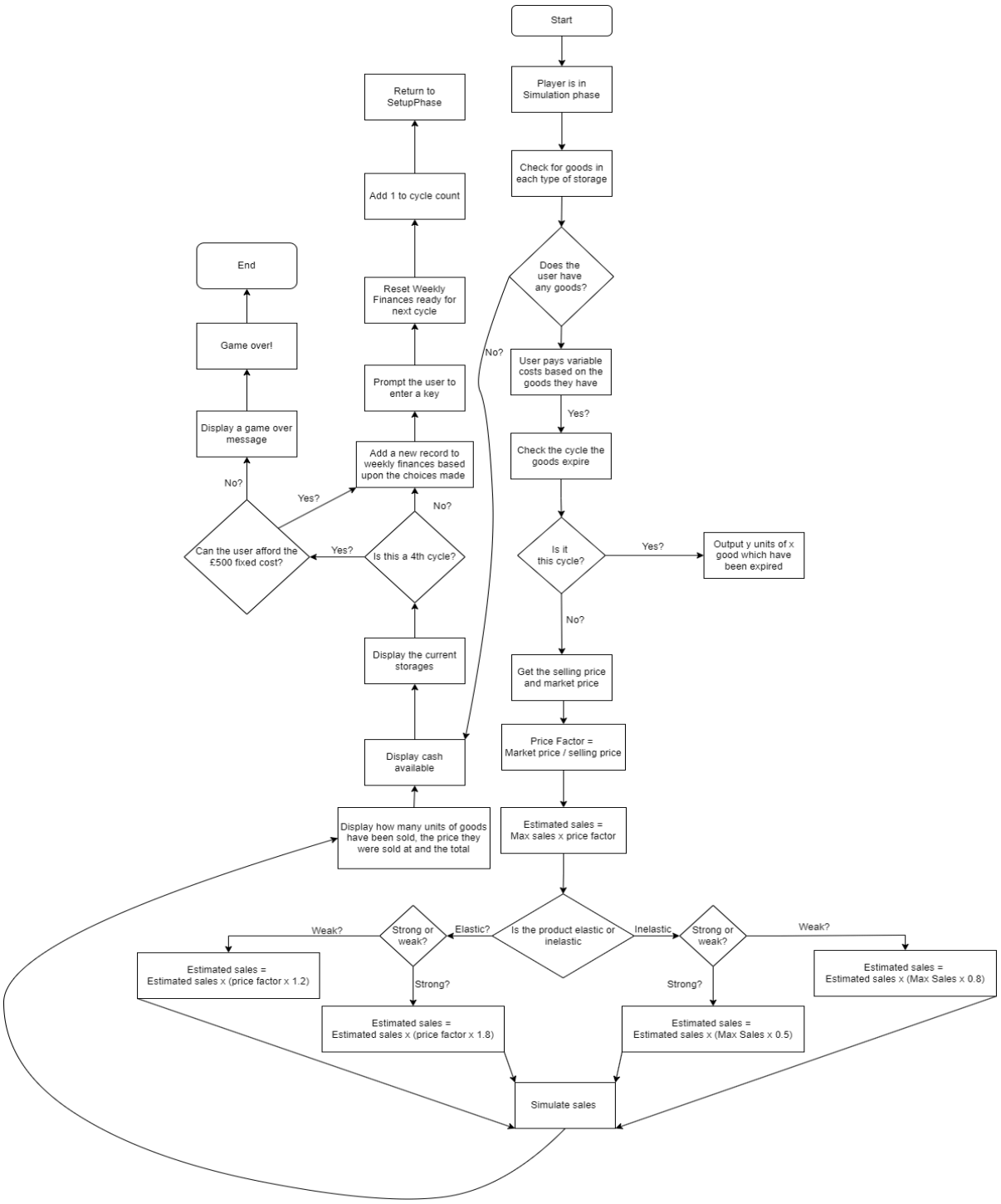
Setup Phase flowchart:

Below is a flowchart which depicts the logic/flow of the Setup phase. Showing all the options/paths available to the player in this phase.



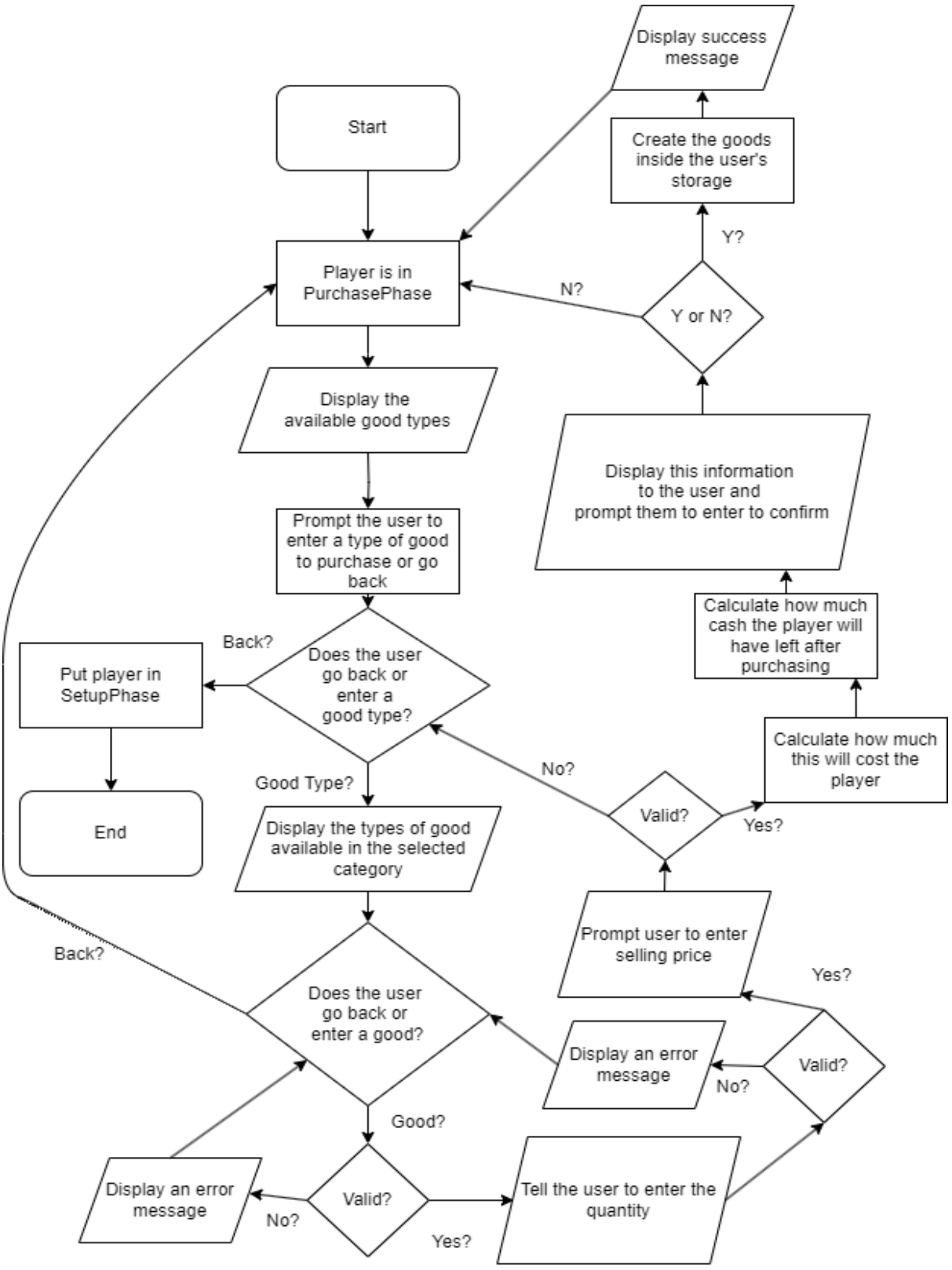
Simulation Phase flowchart:

The flowchart below depicts the logic/flow of the simulation phase. Most of the flowchart is “behind the scenes” as in the simulation phase the user only sees the result of their actions. While the flowchart shows how we arrive at that result.



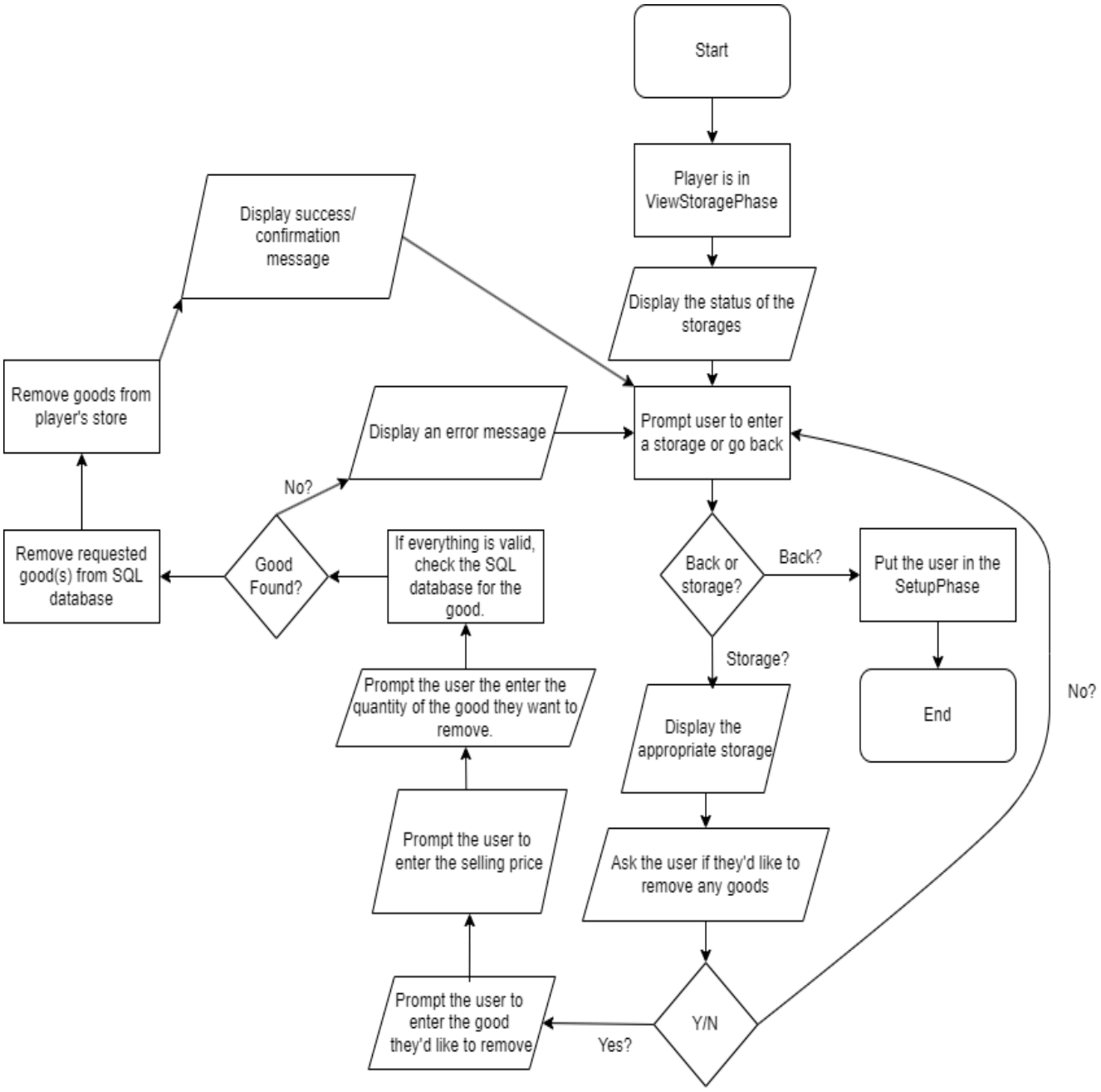
Purchase Phase flowchart:

The following flowchart then shows the logic behind the user purchasing goods.


Start
Player is in PurchasePhase
Display the available good types
Prompt the user to enter a type of good to purchase or go back
Does the user go back or enter a good type?
Back?
Put player in SetupPhase
End
Good Type?
Display the types of good available in the selected category
Does the user go back or enter a good?
Back?
Good?
Valid?
No?
Display an error message
Yes?
Tell the user to enter the quantity
Prompt user to enter selling price
Valid?
No?
Display an error message
Yes?
Calculate how much this will cost the player
Valid?
No?
Yes?
Calculate how much cash the player will have left after purchasing
Display this information to the user and prompt them to enter to confirm
Y or N?
N?
Y?
Create the goods inside the user's storage
Display success message

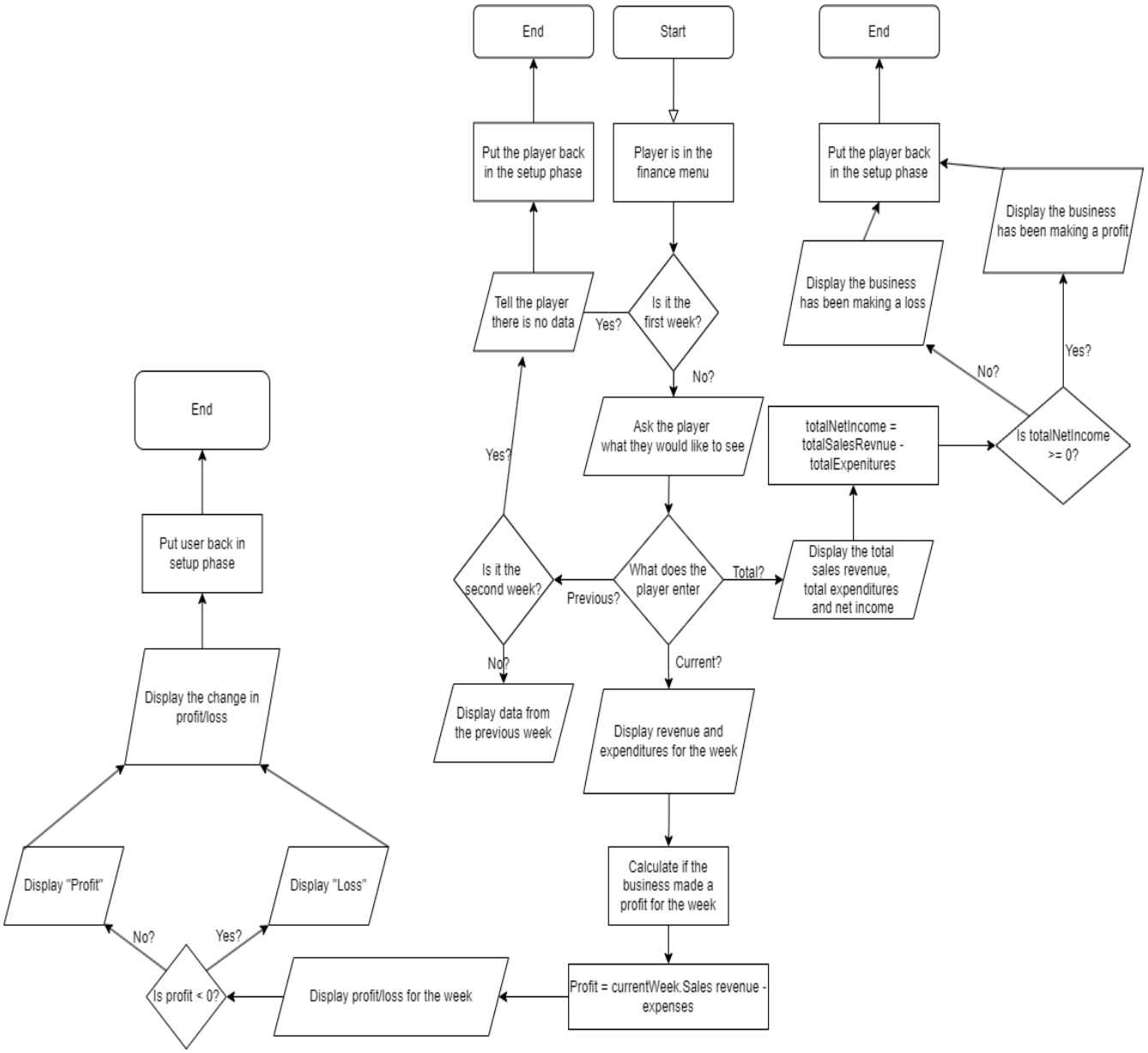
View Storage Phase flowchart:

Flowchart showing the logic of how the user may view their storage



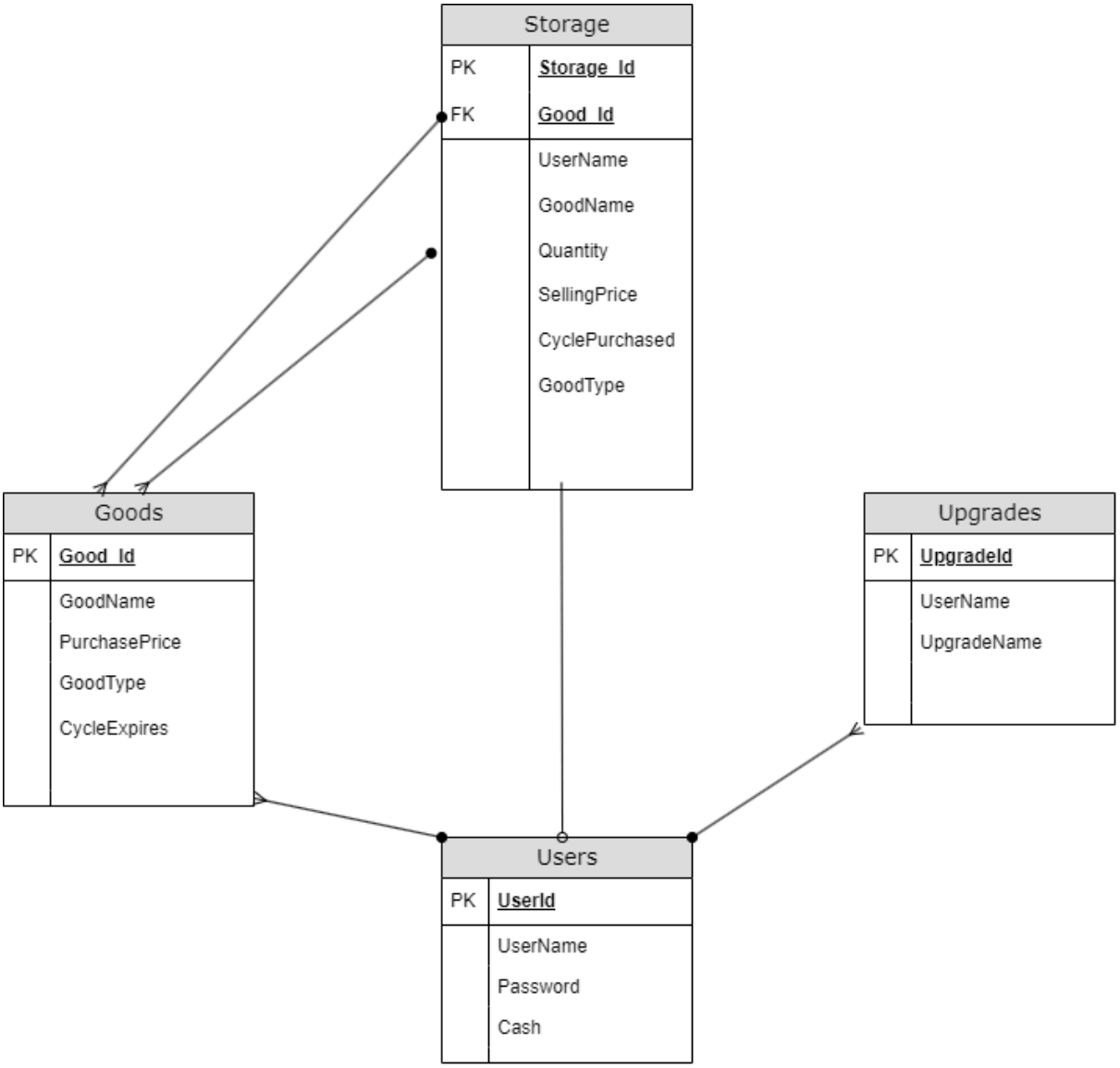
Finance Menu Flowchart:

Flowchart showing the overall flow of how the finance menu operates



Entity relationship diagram:

ERD showing how the tables within my DB relate/interact with each other:



Testing:

(Link to spreadsheet and videos is in the Github repository)
As well as playing the game to test specific requirements, I also gave it to 8 of my friends to get their feedback on it. 7 of my friends who played it also take business studies, 2 take both business and Economics and 3 study

economics. So this allowed me to properly see/determine if my game does help business studies students gain a better understanding. All of my friends agreed that the game is fun. The 7 friends who take business studies agreed it could help them pick up application marks when writing essays. So while I believe it could be more in depth and potentially have more functionalities, I can say it does help students answer questions better hence aligning with the base purpose to help educate business & economics students.s

Testing of requirements/objectives

Phase	Req. ID	Description	Test Method	Expected Result
PreGame	Pre1	Display the main menu options ("New Game", "Load Game", "Quit")	Start the program	Options "New Game", "Load Game" and "Quit" are shown
PreGame	Pre2	Quit the program from the main menu	Enter "Quit" at the main menu	Program ends with a "Thank you" message
PreGame	Pre3	New-game setup prompts	Enter "New Game" at the main menu	Prompt for username and password
PreGame	Pre4	Save new user credentials	Inspect the SQLite DB (e.g. via DB Browser)	Username and password record present
PreGame	Pre5	Transition into setup phase	Complete new-game signup	User is placed into the setup phase
PreGame	Pre6	Load existing game	Enter "Load Game" at the main menu	Previous user data (cash, goods, upgrades) is loaded
PreGame	Pre7	Verify data loading	Play through a session, then reload	Data is correctly reloaded
PreGame	Pre8	Handle invalid main-menu input	Enter an unrecognized command (e.g. "barry")	Error message displayed
Setup	Set1	Indicate setup phase	Enter any command to trigger setup	"Setup Phase" label appears
Setup	Set2	Display current cash balance	Enter setup phase	Cash amount is shown
Setup	Set3	Reflect purchases in storage	Buy goods, then "View Storage"	Purchased items appear in storage list

Setup	Set4	Present all setup-phase options	Observe setup menu	Options Simulate, Purchase, View, Upgrades, Save, Quit, Finance show
Setup	Set5	Enter simulation phase	Enter “Sim”	User enters the simulation phase
Setup	Set6	Enter purchase phase	Enter “Purchase”	User enters the purchase phase
Setup	Set7	Enter storage-view phase	Enter “View”	User enters the view storage phase
Setup	Set8	Enter upgrades phase	Enter “Upgrades”	User enters the upgrades phase
Setup	Set9	Save current game	Enter “Save”	Database is updated with current cash, goods, upgrades
Setup	Set10	Quit from setup	Enter “Quit”	Program ends with a “Thank you” message
Setup	Set11	Enter finance menu	Enter “Finance”	Finance menu options appear
Sim	Sim1	Indicate simulation phase	Enter a simulation cycle	“Simulation Phase” label appears
Sim	Sim2	Report units sold	Buy 10 units of milk at £12, sell at £15 each	Console shows how many units sold
Sim	Sim3	Report selling price	As above	Console shows the selling price of each unit sold
Sim	Sim4	Report revenue	As above	Console shows total revenue from the sale
Sim	Sim5	Update cash balance	Complete a sale cycle	Cash balance reflects post-sale amount
Sim	Sim6	Track remaining inventory	Buy goods in each category, then simulate	Remaining units in each storage are shown
Sim	Sim7	Expiry notification	Hold perishable goods past their expiry cycle	Notification of expired goods and quantities
Sim	Sim8	Prompt for next cycle	Finish a simulation	Prompt “Press any key to begin next cycle”
Sim	Sim9	Monthly bills deduction	Run four consecutive cycles	£500 bill is automatically deducted
Sim	Sim10	Handle insufficient funds for bills	Ensure cash < £500 at bill time	Game ends with “cannot afford bills” message
Sim	Sim11	No sales / game over	Set selling prices too high	No units sold, then bill time → game over

Sim	Sim12	Confirm game-over condition	As above	Game terminates
Sim	Sim13	Dynamic price updates	After a simulation cycle	Market prices differ from their initial values
Sim	Sim14	Show weekly finance summary	Enter finance menu post-simulation	Current P/L sheet displays revenues and expenditures
Purchase	Pur1	Indicate purchase phase	From setup menu enter “Purchase”	“Purchase Phase” heading is displayed
Purchase	Pur2	Prompt for storage type	In purchase phase observe first prompt	Prompt “Enter storage type to purchase (Fresh/Chilled/Frozen/Regular) or Back”
Purchase	Pur3	List available storage categories	In purchase phase	Fresh, Chilled, Frozen and Regular are listed
Purchase	Pur4	Show option to go back	In purchase menu	“Back” option is displayed
Purchase	Pur5	Back returns to setup	Enter “Back”	User is returned to the setup-phase menu
Purchase	Pur6	Invalid storage input → error + reprompt	Enter “Foo” as storage type	Error “Invalid option, try again” and re-prompt storage type
Purchase	Pur7	Valid storage input lists that category’s goods	Enter “Frozen”	All frozen products (names) are displayed
Purchase	Pur8	Entering valid storage moves into that sub-menu	Enter “Frozen”	Subsequent prompts relate to Frozen category
Purchase	Pur9	Display product names in chosen storage	In Frozen sub-menu	List shows each product’s name
Purchase	Pur10	Display current market prices	In product list	Each product shows its current purchase price
Purchase	Pur11	With elasticity upgrade, show elasticity info	First buy “Experienced Entrepreneur” upgrade, then re-enter Purchase	Each product line shows “Elastic” or “Inelastic”
Purchase	Pur12	Prompt to enter product name	After listing products	Prompt “Enter product name or Back”

Purchase	Pur13	Invalid product name → error + reprompt	Enter “NoSuchGood”	Error “Product not found, please try again” and re-prompt
Purchase	Pur14	Valid product name → prompt for quantity	Enter a listed product name	Prompt “Enter quantity to buy”
Purchase	Pur15	Invalid quantity → error + reprompt	Enter “abc” or negative	Error “Invalid quantity, please enter a positive integer” and re-prompt
Purchase	Pur16	Valid quantity → success message	Enter a valid integer within storage capacity	Message “You have selected X units”
Purchase	Pur17	Prompt for selling price	After valid quantity	Prompt “Enter your desired selling price per unit”
Purchase	Pur18	Invalid price → error + reprompt	Enter “-5” or “xyz”	Error “Invalid price, please enter a positive number” and re-prompt
Purchase	Pur19	Valid price → final confirmation	Enter a valid decimal price	Message “Purchased X units at £Y each. Added to storage.”
Purchase	Pur20	Insufficient cash or storage → error + reprompt	Try buying more than cash allows or over capacity	Error “Cannot afford that purchase” or “Not enough storage space” and re-prompt
Purchase	Pur21	Option to cancel purchase	At product-name or quantity prompt enter “Back”	Returns to storage-type selection prompt
Purchase	Pur22	“Back” at any stage returns to Purchase start	Enter “Back” at selling-price prompt	Returns to the initial Purchase menu (storage-type prompt)
Storage	Sto1	Indicate view-storage phase	From setup menu enter “View”	“View Storage Phase” heading is displayed
Storage	Sto2	List all storage types	In view-storage menu	Shows “Fresh, Chilled, Frozen, Regular” with unit capacity labels
Storage	Sto3	Show current units in each storage	In view menu	E.g. “Fresh: 12/100 units, Chilled: 5/50 units, …”
Storage	Sto4	Prompt to choose a storage to inspect	After listing storages	Prompt “Enter storage type to view or Back”

Storage	Sto5	Option to go back	In view-storage menu	“Back” option is displayed
Storage	Sto6	Back returns to setup	Enter “Back”	User is returned to the setup menu
Storage	Sto7	Valid storage input lists its contents	Enter “Fresh”	Lists each Fresh product in inventory
Storage	Sto8	Display product name, quantity held, and selling price	In storage contents display	Table or lines showing Name
Storage	Sto9	Prompt to continue after viewing	After listing contents	“Press any key to return to setup phase”
Storage	Sto10	Enforce max capacity	Fill storage to capacity, then view	Display shows no more than max capacity and correct count
Upgrades	Upg1	Indicate upgrades phase	From the setup menu enter “Upgrades”	“Upgrades Phase” heading is displayed
Upgrades	Upg2	Display the names of available upgrades	In the upgrades menu	A list of all upgrade names is shown
Upgrades	Upg3	Display the price of each upgrade	In the upgrades menu	Each upgrade name has its cost shown next to it
Upgrades	Upg4	Display a brief description of what each upgrade does	In the upgrades menu	A descriptive blurb appears under or beside each upgrade
Upgrades	Upg5	Provide a way to select and purchase an upgrade	Choose an upgrade number or code and confirm purchase	Upgrade is bought, cash is deducted, and upgrade added to inventory
Upgrades	Upg6	Offer a “Back” option to return to setup	In the upgrades menu enter “Back”	Returns to the main setup-phase menu
Upgrades	Upg7	Display confirmation message once an upgrade is purchased	After completing an upgrade purchase	Message “Upgrade ‘X’ purchased!” appears
Upgrades	Upg8	Prevent purchasing an upgrade already owned	Attempt to buy the same upgrade twice	Purchase is blocked; no cash deducted

Upgrades	Upg9	Display message if user tries to buy an owned upgrade again	Select an owned upgrade and confirm	Message “You already own this upgrade” appears
Upgrades	Upg10	Apply the upgrade’s effect successfully	Buy an upgrade (e.g. “Experienced Entrepreneur”), then trigger its effect	The corresponding feature appears in-game (e.g. elasticity info)
Finance	Fin1	Display finance options: Current, Previous, Total, Back	Enter “Finance” from setup	All 4 options shown
Finance	Fin2	Allow user to go back from finance menu	Enter “B” or “Back”	Returns to setup
Finance	Fin3	Show no data message on week 1 current sheet	Start new game → Finance → “C”	Message “No current P/L data”
Finance	Fin4	Show current P/L sheet after at least one week	Complete a week then enter Finance → “C”	Displays current week’s P/L summary
Finance	Fin5	Sheet includes week number	Any sheet view	Line “Week: X” is visible
Finance	Fin6	Show sales revenue	View any P/L	Line “Sales Revenue: £X” is present
Finance	Fin7	Show expenditures (bills, goods, upgrades)	View any P/L	Separate lines for bills, purchases, and upgrade costs
Finance	Fin8	Show net income	View any P/L	“Net Income: £X” is calculated
Finance	Fin9	Indicate profit or loss	View any P/L	Message “You made a profit” or “You incurred a loss”
Finance	Fin10	Compare current week to previous	View current sheet	Shows change in income vs previous week
Finance	Fin11	Show error for “Previous” on week 1	Start game → Finance → “P”	Message “No previous data”
Finance	Fin12	Show correct sheet for	Week 2+, then Finance → “P”	Shows week 1’s summary

		"Previous" from week 2+		
Finance	Fin13	Error message if "Total" selected on week 1	Week 1 → Finance → "T"	"No P/L history available yet."
Finance	Fin14	Display total revenue after multiple weeks	Finance → "T"	"Total Revenue: £X" shown
Finance	Fin15	Display total expenditures (bills + goods + upgrades)	Finance → "T"	Lists total of each category
Finance	Fin16	Show total net income	Finance → "T"	"Net Income Overall: £X" shown
Finance	Fin17	Show profitability assessment	Finance → "T"	Message "Business has been profitable" or "Loss of £X"
Finance	Fin18	Return to setup after viewing any finance sheet	After viewing a sheet, press Enter	Returns to setup menu

Evaluation

Upon looking at my finished product I can say that it has been successful. All requirements were met except from:
Upgrade requirement 7 (Upg7): Display confirmation message once an upgrade is purchased
Upgrade requirement 9 (Upg9): Display message if user tries to buy an owned upgrade again.
While a message is technically displayed, it's not on the screen long enough for the player to properly see and understand. Following this after doing my testing by giving my game to friends to play, they all agreed that after playing it they felt like they were more confident when it came to answering an exam question. The main two reasons being they understand better how elasticities work, and they have a better idea of how to pick up analysis and

application marks. Which meets the original purpose of my project which was “to help educate Business studies students, economics students and anyone else who is interested and give them some form of Idea of what it may be like to run their own business.” Furthermore, during testing of this project, one of my friends gave me the idea of removing goods from storage. While playing it they purchased some goods and set them at a selling price too high. Since my friend set the price too high, this meant the goods would not sell and they were just taking up storage space. This was quite annoying/frustrating and my friend suggested there should be a way to remove goods. And so that’s what I did. While writing the code to ensure the goods were removed in game, I also had to make sure the goods would be removed from the DB if the user had saved.

Conclusion:

Overall while I can confidently say my project was successful, I believe that it could’ve been done better. I feel as though I could’ve spent more time planning it out to make it a bit more smooth and nice to play. Furthermore, I could’ve also applied some models and theories from economics and Business studies to make this more educational and beneficial. On top of this, I could’ve had these “events” within the game where the player was given some data/information and they had multiple choices to choose from, With the correct choice leading to the player enjoying more benefits and the incorrect ones leading to consequences. For example, in business studies there is a theory about employee motivation, one of the factors linking to employee motivation is the standards of the workplace. A question would have been. “You’ve made an additional £500 of profit this month, however you notice employee motivation is also down by 5%. According to Maslow’s hierarchy of needs, what would be the best choice to spend this additional profit on?: 1. More storage. 2. Renovations. 3. More goods. 4. A faster supplier. “ While all of these are technically good choices, according to the theory the correct choice would be B. This could’ve helped players better answer exam multiple choice questions while also improving their general knowledge of the subject and memory of the theory. So ultimately, while I am happy with my finished product, I do believe and recognise I could’ve done better in some areas.

Code:

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using BusinessSimulator;
using System.Data.SQLite;
using System.Reflection.Metadata;
using static System.Net.Mime.MediaTypeNames;
using System.ComponentModel.Design;

namespace BusinessSimulator
{
    class Program
    {
        static void Main(string[] args)
        {
            Game game = new Game(); // Create a new instance of the Game
class.
            game.MainMenu(); // Start the game.

        }
    }
    // class used to allow the connection string to be used in multiple classes
so that data can be saved to and loaded from sql database
    public static class DataBaseConfig
    {

        //public const string ConnectionString = @"Data
Source=.\Files\NEAdataBaseTest.db;Version=3";
        public const string ConnectionString = @"Data Source =
C:\Users\sampr\OneDrive\Desktop\KAB6 Comp Sci\Comp Sci
NEA\NEAProtoSave\NEAProtoSave\Files\NEAdataBaseTest.db;Version=3";
```

```
    }
    public class Game
    {


        private string UserName; // Declares UserName at class level
        private Store playerStore; // Represents the player's store.
        private Market market; // Represents the market where prices are set.
        private int cycleCount; // Tracks the number of cycles completed.
        private List<Upgrades> availableUpgrades;
        private List<WeeklyFinance> weeklyFinances = new
List<WeeklyFinance>(); // list used to store weekly finances
        private decimal currentWeekSalesRevenue = 0; // tracks the sales
revenue for the current week
        private decimal currentWeekPurchaseExpenses = 0;// tracks the
purchase expenses for the current week
        private decimal currentWeekBillsExpenses = 0; // tracks the bills
expenses for the current week(if any/possible)
        private decimal currentWeekUpgradesExpenses = 0; // again, tracks
the upgrade expenses for the current week(if any/possible)

        // creates all required tables if they're not found in sql database
        private void EnsureUsersTablesExists()

{
    string createUsersTableSQL = @"
        CREATE TABLE IF NOT EXISTS Users (
            Id INTEGER PRIMARY KEY AUTOINCREMENT,
            Username TEXT NOT NULL UNIQUE,
            Password TEXT NOT NULL,
            Cash REAL
        );";

    string createGoodsTableSQL = @"
        CREATE TABLE IF NOT EXISTS Goods (
            Good_Id INTEGER PRIMARY KEY,
            GoodName TEXT NOT NULL,
```

```
            PurchasePrice REAL NOT NULL,
            GoodType INT NOT NULL,
            CycleExpires INT NOT NULL
        );";

            string createStorageTableSQL = @"
            CREATE TABLE IF NOT EXISTS Storage(
        Storage_Id INTEGER PRIMARY KEY,
        UserName TEXT NOT NULL,
        GoodName TEXT NOT NULL,
        Good_Id INT NOT NULL,
        Quantity INT NOT NULL,
        SellingPrice REAL NOT NULL,
        CyclePurchased INT NOT NULL,
        GoodType INT NOT NULL, --1 = Chilled, 2 = Fresh, etc.
        FOREIGN KEY(Good_Id) REFERENCES Goods(Good_Id)
        );";
            string createUpgradesTableSQL = @"
             CREATE TABLE IF NOT EXISTS Upgrades(
        UpgradeId INTEGER PRIMARY KEY AUTOINCREMENT,
        UserName TEXT NOT NULL,
        UpgradeName TEXT NOT NULL,
        UNIQUE(UserName, UpgradeName)
        );";
            // code to actually create the tables
            using (SQLiteConnection conn = new
SQLiteConnection(DataBaseConfig.ConnectionString))
            {

        conn.Open();
        using (SQLiteCommand cmd = new
SQLiteCommand(createUsersTableSQL, conn))
        {
            cmd.ExecuteNonQuery();
        }
        using(SQLiteCommand cmd = new
SQLiteCommand(createGoodsTableSQL, conn))
```

```
        {
            cmd.ExecuteNonQuery();
        }
        using (SQLiteCommand cmd = new
SQLiteCommand(createStorageTableSQL, conn))
        {
           cmd.ExecuteNonQuery();
        }
        using (SQLiteCommand cmd = new
SQLiteCommand(createUpgradesTableSQL, conn))
        {
            cmd.ExecuteNonQuery();
        }
    }
}
        // all the relevant data needed to add goods to storage
        private void AddGoodsToStorage(string UserName, int Good_Id, string
ProductName, int GoodType, int Quantity, decimal SellingPrice, int
CyclePurchased)
        {
            //Console.WriteLine($"DEBUG: Attempting to add '{ProductName}'
(Good_Id: {Good_Id}) to storage.");

            if (Good_Id == -1)
            {
                Console.WriteLine($"ERROR: Product '{ProductName}' not found in
Goods table.");
                return; // Exit if the product ID is invalid
            }
            // sql to insert goods into storage
            string insertSQL = @"
            INSERT INTO Storage (UserName, Good_Id, GoodName, Quantity,
SellingPrice, CyclePurchased, GoodType)
            VALUES (@UserName, @Good_Id, @GoodName, @Quantity,
@SellingPrice, @CyclePurchased, @GoodType);";
```

```
        using (SQLiteConnection conn = new
SQLiteConnection(DataBaseConfig.ConnectionString))
        {
            conn.Open();
            using (SQLiteCommand cmd = new SQLiteCommand(insertSQL,
conn))
            {
                cmd.Parameters.AddWithValue("@UserName", UserName);//
adds the relevant data to the sql command
                cmd.Parameters.AddWithValue("@Good_Id", Good_Id);
                cmd.Parameters.AddWithValue("@GoodName", ProductName);
                cmd.Parameters.AddWithValue("@Quantity", Quantity);
                cmd.Parameters.AddWithValue("@SellingPrice", SellingPrice);
                cmd.Parameters.AddWithValue("@CyclePurchased",
CyclePurchased);
                cmd.Parameters.AddWithValue("@GoodType", GoodType);

                cmd.ExecuteNonQuery();
            }
        }

        //Console.WriteLine($"DEBUG: Successfully added '{ProductName}'
(Good_Id: {Good_Id}) to storage.");
    }
    private void UpgradesMenu()
    {
        Console.Clear();
        Console.WriteLine("=== Upgrades ===");
        for (int i = 0; i < availableUpgrades.Count; i++) // logic to display list
of avaialble upgrades
        {
            var upgrade = availableUpgrades[i]; //wrties the name and price
of upgrade
            Console.WriteLine($"{i + 1}. {upgrade.Name} - £{upgrade.Price}");
            Console.WriteLine($" {upgrade.Description}"); // short description
of the upgrade
        }
```

```
            Console.WriteLine("Enter the number of the upgrade you'd like to
purchase or enter 0 to go back");
            if(int.TryParse(Console.ReadLine(), out int choice) && choice > 0 &&
choice <= availableUpgrades.Count)
            {
                PurchasedUpgrades(availableUpgrades[choice - 1]);
            }
            else
            {
                Console.WriteLine("Invalid choice");
            }
        }
        private void SaveUpgrades(string userName, string upgradeName) //
logic to save upgrades to the database
        {
            string SQL = @"
            INSERT OR IGNORE INTO Upgrades (UserName, UpgradeName)
            VALUES (@UserName, @UpgradeName);";

            using (SQLiteConnection conn = new
SQLiteConnection(DataBaseConfig.ConnectionString))
            {
                conn.Open();
                using (SQLiteCommand cmd = new SQLiteCommand(SQL, conn))
                {
                    cmd.Parameters.AddWithValue("@UserName", userName);
                    cmd.Parameters.AddWithValue("@UpgradeName",
upgradeName);
                    cmd.ExecuteNonQuery();
                }
            }
           // Console.WriteLine($"DEBUG: Upgrade '{upgradeName}' saved for
user '{userName}'.");
        }
```

```
        private List<string> LoadUpgrades(string userName) // logic to load
upgrades from the database
        {
            string SQL = "SELECT UpgradeName FROM Upgrades WHERE
UserName = @UserName;";
            List<string> upgrades = new List<string>();

            using (SQLiteConnection conn = new
SQLiteConnection(DataBaseConfig.ConnectionString))
            {
                conn.Open();
                using (SQLiteCommand cmd = new SQLiteCommand(SQL, conn))
                {
                    cmd.Parameters.AddWithValue(@"UserName", userName);
                    using (SQLiteDataReader reader = cmd.ExecuteReader())
                    {
                        while (reader.Read())
                        {
                            upgrades.Add(reader.GetString(0));
                        }
                    }
                }
            }
            //Console.WriteLine($"DEBUG: Loaded {upgrades.Count} upgrades
for user '{userName}'");
            return upgrades;
        }
        private void ApplyUpgrades(List<string> loadedupgrades) // logic to
actually apply upgrades to the store when loading upgrades
        {
            foreach (string upgradeName in loadedupgrades)
            {
                Upgrades upgrade = availableUpgrades.FirstOrDefault(u =>
u.Name == upgradeName); // find matching upgrade

                if (upgrade != null)
                {
```

```
                //Console.WriteLine($"DEBUG: Found upgrade
'{upgrade.Name}'. Re-applying effect");
                upgrade.Effect(playerStore);
                //Console.WriteLine($"DEBUG: Re-applied '{upgrade.Name}'
upgrade: {upgrade.Description}");
            }
            else
            {
                Console.WriteLine($"WARNING: Loaded upgrade
'{upgradeName}' doesn't match any upgrade");
            }
        }
    }


    public Game()
    {
        string ConnectionString = @"Data
Source=C:\\Users\\sampr\\OneDrive\\Desktop\\KAB6 Comp Sci\\Comp Sci
NEA\\NEAProtoSave\\NEAProtoSave\\Files\\NEAdataBaseTest.db;Version=3;";
        playerStore = new Store(1000, UserName); // Initialize the store
with £1000.
        market = new Market(ConnectionString); // Initialize the market.
        cycleCount = 0; // Start the cycle count at 0.
        InitialiseUpgrades();
    }
    private void InitialiseUpgrades()
    {
        availableUpgrades = new List<Upgrades>
        {
            new Upgrades("Sales Boost", 200, "Increases sales by 5%
regardless of elasticity",
            Store =>
            {
                if (!Store.HasUpgrade("Sales Boost"))
                {
                    Store.AdjustCash(0); //testing
```

```
                    //Console.WriteLine("DEBUG: Applying sales boost effect");
                }
            }),
            new Upgrades("Elasticity Insight", 500, "Reveals whether a
product is elastic or inelastic",
            Store =>
            {
                //Console.WriteLine("DEBUG: Applying elasticity insight effect");
            }),
        };
    }
    public void PurchasedUpgrades(Upgrades upgrade)
    {
        if (playerStore.HasUpgrade(upgrade.Name))
        {
            // prevents the player from buying the same upgrade multiple
times (wasting money)
            Console.WriteLine($"You already own the '{upgrade.Name}'
upgrade.");
            return;
        }

        if (playerStore.Cash >= upgrade.Price) // checks if the player has
enough cash to buy the upgrade
        {
            playerStore.Cash -= upgrade.Price; // deduct the cost of the
upgrade from the player's cash
            currentWeekUpgradesExpenses += upgrade.Price; // Add the cost
of the upgrade to the weekly total.
            playerStore.AddUpgrade(upgrade.Name); // Add the upgrade to
the player's list of upgrades
            upgrade.Effect(playerStore); // Apply the effect of the upgrade
            SaveUpgrades(playerStore.UserName, upgrade.Name); // Save the
upgrade to the database

            Console.WriteLine($"'{upgrade.Name}' purchased successfully!
{upgrade.Description}");
```

```
            }
            else
            {
                Console.WriteLine("Not enough cash to purchase this upgrade.");
            }
        }


        public void MainMenu()
        {
            while (true)
            {
                Console.Clear();
                Console.ForegroundColor = ConsoleColor.Yellow;
                Console.WriteLine("=== Welcome to the Business Simulator ===");
                Console.ResetColor();
                Console.WriteLine("(N)ew Game");
                Console.WriteLine("(L)oad Game");
                Console.WriteLine("(Q)uit");

                string choice = Console.ReadLine().Trim().ToLower(); // Convert
input to lowercase for consistent comparison

                switch (choice)
                {
                    case "n":
                    case "new game":
                        SetupNewPlayer(); // Start a new game
                        return;
                    case "l":
                    case "load game":
                        Console.WriteLine("Enter your business name to load game");
                        string username = Console.ReadLine();
                        LoadGame(username);
                        return;
                    case "q":
                    case "quit":
```

```
                Console.WriteLine("Thank you for playing!");
                Environment.Exit(0); // Exit the program
                break;
            default:
                Console.WriteLine("Invalid option, please try again.");
                break;
        }
    }
}
    public void SetupNewPlayer()
    {
        EnsureUsersTablesExists(); // first checks if the tables exist in the
database
        Console.Clear();
        Console.ForegroundColor = ConsoleColor.Yellow;
        Console.WriteLine("===New Game===");
        Console.ResetColor();

        Console.WriteLine("Please enter a name for your business ");
        string UserName = Console.ReadLine().Trim();

        Console.WriteLine("Please select a password");
        string Password = Console.ReadLine().Trim();

        if (CreateNewPlayer(UserName, Password))
        {
            Console.WriteLine("Player created successfully, starting game");
            playerStore = new Store(1000, UserName);
            Start();
        }
        else
        {
            Console.WriteLine("Failed to create user, please try again");
            Console.ReadKey();

        }
    }
```

```
        private bool CreateNewPlayer(string UserName, string Password)
        {
            // creates a new profile for the player
            string sql = "INSERT INTO Users (Username, Password, Cash) VALUES
(@UserName, @Password, @Cash);";
            try
            {
                using (SQLiteConnection conn = new
SQLiteConnection(DataBaseConfig.ConnectionString))
                {
                    conn.Open();
                    Console.WriteLine("Database connection opened.");
                    using (SQLiteCommand cmd = new SQLiteCommand(sql, conn))
                    {
                        cmd.Parameters.AddWithValue("@UserName", UserName);
                        cmd.Parameters.AddWithValue("@Password", Password);
                        cmd.Parameters.AddWithValue("@Cash", 1000.00);
                        cmd.ExecuteNonQuery();
                        Console.WriteLine("User inserted.");
                        Console.WriteLine("Enter any key to continue");
                        Console.ReadKey();
                    }
                }
                return true;
            }
            catch (SQLiteException ex)
            {
                if ((SQLiteErrorCode)ex.ErrorCode ==
SQLiteErrorCode.Constraint)
                {
                    Console.WriteLine("Error: Username already exists, please
choose another");
                }
                else
                {
                    Console.WriteLine($"Database error: {ex.Message}");
```

```
            }
            return false;
        }

    }


    public void Start()
    {
        while (true)
        {
            cycleCount++; // Increment the cycle count at the start of each
loop.

            SetupPhase(); // Enter the setup phase where the player makes
decisions.

            // At the end of each month/ every 4 cycles attempt to pay bills.
            if (cycleCount % 4 == 0)
            {
                bool canPayBills = playerStore.PayBills(500); // Attempt to pay
£500 in bills.
                if (!canPayBills)
                {
                    Console.WriteLine("You cannot afford to pay the bills. Game
over!");
                    break; // End the game if bills cannot be paid.
                }
                else
                {
                    Console.WriteLine("You have paid £500 towards bills.");
                    currentWeekBillsExpenses += 500; // Add the bill payment
to the weekly total
                }

                // Check for any expired chilled goods.
                playerStore.CheckForExpiredGoods(cycleCount);
```

```
            }

            SimulationPhase(); // Simulate the sales for this cycle.

            WeeklyFinance wf = new WeeklyFinance() // this is used to keep
track of the weekly finances for the p/l sheet
            {
                Week = cycleCount,
                SalesRevenue = currentWeekSalesRevenue,// sets the sales
revenue for the week
                PurchaseExpenses = currentWeekPurchaseExpenses, // sets the
purchase expenses for the week
                BillsExpenses = currentWeekBillsExpenses, // sets the bills
expenses for the week
                UpgradeExpenses = currentWeekUpgradesExpenses // sets the
upgrade expenses for the week
            };
            weeklyFinances.Add(wf);

            //resets ready for the next week
            currentWeekBillsExpenses = 0;
            currentWeekPurchaseExpenses = 0;
            currentWeekSalesRevenue = 0;
            currentWeekUpgradesExpenses = 0;

            Console.WriteLine("Press any key to start the next cycle...");
            Console.ReadKey(); // Wait for player input to proceed.
        }

    }




    private void SetupPhase()
    {
        while (true)
```

```
        {
            Console.Clear();
            Console.ForegroundColor = ConsoleColor.Yellow;
            Console.WriteLine("=== Setup Phase ===");
            Console.ResetColor();
            Console.ForegroundColor = ConsoleColor.DarkCyan;
            playerStore.DisplayStatus(); // Display current cash and inventory
status.
            Console.ResetColor();

            Console.WriteLine("Enter '(S)im' to simulate the next week.");
            Console.WriteLine("Enter '(P)urchase' to buy goods.");
            Console.WriteLine("Enter '(V)iew' to view your storage.");
            Console.WriteLine("Enter '(U)pgrades' to view and buy upgrades.");
            Console.WriteLine("Enter '(F)inance' to view your finances.");
            Console.WriteLine("Enter 'Save' to save your game");
            Console.WriteLine("Or enter (Q)uit to quit the game");
            Console.WriteLine("Helpful Hint: You will pay £500 in bills every 4
weeks (This is a fixed cost)");

            string choice = Console.ReadLine().Trim().ToLower();

            switch (choice)
            {
                case "sim":
                case "s":
                    return; // Exit the setup phase and proceed to simulation.
                case "purchase":
                case "p":
                    PurchasePhase(); // Proceed to the purchase phase.
                    break;
                case "view":
                case "v":
                    ViewStoragePhase(); // Proceed to view storage.
                    break;
                case "save":
                    SaveGame();
```

```
                    break;
                case "finance":
                case "f":
                    FinanceMenu(); // takes player to finance menu
                    break;
                case "upgrades":
                case "u":
                    UpgradesMenu(); // Show the upgrades page.
                    break;
                case "quit":
                case "q":
                    Console.WriteLine("Thank you for playing!");
                    Environment.Exit(0); // Quit the program.
                    break;
                default:
                    Console.WriteLine("Invalid option. Please try again.(Enter any
key)");
                    Console.ReadKey();
                    break;
            }
        }
    }
    private void SaveGame()
    {
        string sql = "UPDATE Users SET Cash = @Cash WHERE UserName =
@UserName;";
        try
        {
            using (SQLiteConnection conn = new
SQLiteConnection(DataBaseConfig.ConnectionString))
            {
                conn.Open();
                //Console.WriteLine("Database connection established
successfully.");
                using (SQLiteCommand cmd = new SQLiteCommand(sql, conn))
                {
```

```
                    Console.WriteLine($"Saving for user: {playerStore.UserName},
Cash: {playerStore.Cash}");
                    cmd.Parameters.AddWithValue("@Cash", playerStore.Cash);
                    cmd.Parameters.AddWithValue("@UserName",
playerStore.UserName);
                    int rowsAffected = cmd.ExecuteNonQuery();
                   // Console.WriteLine($"Rows affected: {rowsAffected}");

                    if (rowsAffected > 0)
                    {
                        Console.WriteLine("Save successful");
                    }
                    else
                    {
                        Console.WriteLine("No data saved");
                    }
                }
                string ClearStorageSQL = "DELETE FROM Storage WHERE
UserName = @UserName;"; // clears the storage table and adds the
new/updated data(goods)
                using (SQLiteCommand clearCmd = new
SQLiteCommand(ClearStorageSQL, conn))
                {
                    clearCmd.Parameters.AddWithValue("@UserName",
playerStore.UserName);
                    clearCmd.ExecuteNonQuery();
                }

                foreach (var storageArea in playerStore.storageAreas)
                {
                    foreach (var product in storageArea.Value) // adds the
goods to the storage table
                    {
                        AddGoodsToStorage(
                            playerStore.UserName,
                            market.GetGoodId(product.Name),
                            product.Name,
```

```
                        (int)product.StorageType,
                        product.Quantity,
                        product.SellingPrice,
                        product.CycleAdded
                    );
                }
            }

            Console.WriteLine("Game saved successfully");
            Console.ReadKey();
        }
        }
        catch (SQLiteException ex)
        {
            Console.WriteLine($"Error saving game: {ex.Message}");
        }

    }
    public void LoadGame(string userName)
    {
        //Console.WriteLine($"DEBUG: Attempting to load game for user
'{userName}'.");

        string sql = "SELECT Cash FROM Users WHERE LOWER(Username) =
LOWER(@Username);";

        try
        {
            using (SQLiteConnection conn = new
SQLiteConnection(DataBaseConfig.ConnectionString))
            {
                conn.Open();
                using (SQLiteCommand cmd = new SQLiteCommand(sql, conn))
                {
                    cmd.Parameters.AddWithValue("@Username", userName);
                    object result = cmd.ExecuteScalar(); // Execute the query
and get the result
```

```
                    if (result != null)
                    {
                        decimal loadedCash = Convert.ToDecimal(result);
                        //Console.WriteLine($"DEBUG: Successfully loaded cash
(£{loadedCash:O.OO}) for user '{userName}'.");

                        playerStore = new Store(loadedCash, userName);
                        LoadPlayerGoods(userName);

                        List<string> upgrades = LoadUpgrades(userName);
                        ApplyUpgrades(upgrades);

                        Start(); // Begin game loop after loading data
                    }
                    else
                    {
                        Console.WriteLine($"ERROR: No user found with username
'{userName}' please try again or quit.");
                        Console.ReadLine();
                        MainMenu();
                    }
                }
            }
        }
        catch (SQLiteException ex)
        {
            Console.WriteLine($"ERROR: Failed to load game. {ex.Message}");
        }

    }
    private void LoadPlayerGoods(string userName)
    {
        //Console.WriteLine($"DEBUG: Loading goods for user '{userName}'.");

        string sql = @"
```

```
        SELECT g.GoodName, s.Quantity, s.SellingPrice, s.GoodType,
s.CyclePurchased
        FROM Storage s
        INNER JOIN Goods g ON s.Good_Id = g.Good_Id
        WHERE s.UserName = @UserName;";

        using (SQLiteConnection conn = new
SQLiteConnection(DataBaseConfig.ConnectionString))
        {
            conn.Open();
            using (SQLiteCommand cmd = new SQLiteCommand(sql, conn))
            {
                cmd.Parameters.AddWithValue("@UserName", userName); //
Add the username parameter to the query

                using (SQLiteDataReader reader = cmd.ExecuteReader()) //
Execute the query and read the results
                {
                    while (reader.Read()) // Loop through each row of the result
set
                    {
                        string goodName = reader.GetString(0);
                        int quantity = reader.GetInt32(1);
                        decimal sellingPrice = reader.GetDecimal(2);
                        StorageType goodType = (StorageType)reader.GetInt32(3);
                        int cyclePurchased = reader.GetInt32(4);

                        Product product = new Product(goodName, 0,
sellingPrice, quantity, goodType, cyclePurchased);
                        playerStore.AddProductToStorage(goodType, product);

                        //Console.WriteLine($"DEBUG: Loaded product
'{goodName}' with quantity {quantity}.");
                    }
                }
            }
        }
```

```
        //Console.WriteLine("DEBUG: Finished loading player goods.");
    }



    private void PurchasePhase()
    {
        while (true)
        {
            Console.Clear(); // displays the main options available
            Console.ForegroundColor = ConsoleColor.Yellow;
            Console.WriteLine("=== Purchase Phase ===");
            Console.ResetColor();
            Console.WriteLine("Choose the type of goods to purchase:");
            Console.ForegroundColor = ConsoleColor.Cyan;
            Console.Write("Enter '(F)rozen', ");
            Console.ResetColor();
            Console.ForegroundColor = ConsoleColor.Magenta;
            Console.Write("'(r)egular', ");
            Console.ResetColor();
            Console.ForegroundColor = ConsoleColor.Blue;
            Console.Write("'(c)hilled', ");
            Console.ResetColor();
            Console.ForegroundColor = ConsoleColor.DarkGreen;
            Console.Write("or '(fr)esh'.");
            Console.ResetColor();
            Console.WriteLine("\nOr enter '(B)ack' to return to the main
menu.");

            string choice = Console.ReadLine().Trim().ToLower();

            if (choice == "back")
            {
                break; // Return to the main setup menu.
            }
```

```
            else if (choice == "b")
            {
                break ;
            }

            switch (choice) // Show available goods in the chosen category
            {
                case "frozen":
                case "f":
                    PurchaseGoods("frozen");
                    break;
                case "chilled":
                case "c":
                    PurchaseGoods("chilled");
                    break;
                case "regular":
                case "r":
                    PurchaseGoods("regular");
                    break;
                case "fresh":
                case "fr":
                    PurchaseGoods("fresh");
                    break;
                default:
                    Console.WriteLine("Invalid option. Please try again.");
                    Console.ReadKey();
                    break;
            }
        }
    }
    private void PurchaseGoods(string category)
    {
        Console.Clear();
        if (category == "frozen" || category == "f")
        {
            Console.ForegroundColor = ConsoleColor.Cyan;
        }
```

```
        else if (category == "chilled" || category == "c")
        {
            Console.ForegroundColor = ConsoleColor.Blue;
        }
        else if (category == "regular" || category == "r")
        {
            Console.ForegroundColor = ConsoleColor.Magenta;
        }
        else if (category == "fresh" || category == "fr")
        {
            Console.ForegroundColor = ConsoleColor.Green;
        }
        Console.WriteLine($"=== {category} Goods ===");
        Console.ResetColor();

        // Fetch available goods for the specified category
        var availableGoods = market.GetGoodsByCategory(category);

        if (availableGoods.Count == 0)
        {
            Console.WriteLine("No goods available in this category.");
            Console.ReadKey();
            return;
        }

        // Check if the player has the "Elasticity Insight" upgrade
        bool hasElasticityInsight = playerStore.HasUpgrade("Elasticity
Insight");

        // Display goods with elasticity info if the upgrade is purchased
        foreach (var good in availableGoods)
        {
            string elasticityInfo = hasElasticityInsight
                ? (market.IsElastic(good.Key) ? "(Elastic)" : "(Inelastic)")
                : "";
```

```
            Console.WriteLine($"{good.Key} - Market Price:
£{good.Value:0.00} {elasticityInfo}");
        }
        playerStore.DisplayCash();
        Console.Write("Enter the name of the good to purchase or enter
(b)ack to go back: ");
        string goodName = Console.ReadLine().Trim().ToLower();

        if (goodName == "back" || goodName == "b")
        {

        }
        else if (!availableGoods.ContainsKey(goodName)) // Check if the
entered good is available
        {
            Console.WriteLine("Good not recognized. Please try again.");
            Console.ReadKey();
            PurchaseGoods(category);
        }
        else
        {


            decimal purchasePrice = market.GetMarketPrice(goodName) / 2;
// Set the purchase price to half the market price

            Console.Write($"Enter the quantity of {goodName} to buy: ");
            string prequantity = Console.ReadLine();
            int quantity;
            if (int.TryParse(prequantity, out quantity)) // trys to parse the
input to an integer
            {
                Console.WriteLine($"Successfully purchased {quantity} of
{goodName}.");
            }
            else
            {
```

```
                Console.WriteLine("Invalid Input, please try again");
                Console.ReadLine();
                PurchaseGoods(category);
            }


            Console.Write($"Enter the selling price for {goodName}: ");
            string preprice = Console.ReadLine();
            decimal sellingPrice;
            if (decimal.TryParse(preprice, out sellingPrice)) // trys to parse the
input to a decimal
            {
                if (sellingPrice == 0)
                {
                    Console.WriteLine("Invalid price, please try again");
                    Console.ReadLine();
                    PurchaseGoods(category);
                }
                else if (sellingPrice != 0)
                {
                    Console.WriteLine($"Successfully selling {goodName} for
{sellingPrice} each.");
                }
            }
            else
            {
                Console.WriteLine("Invalid Input, please try again");
                Console.ReadLine();
                PurchaseGoods(category);
            }

            // Create and buy the product
            Product product = new Product(goodName, purchasePrice,
sellingPrice, quantity, market.GetStorageType(goodName), cycleCount);

            if (playerStore.BuyProduct(product)) // if the player has enough
cash and storage space
```

```
            {
                decimal purchaseCost = purchasePrice * quantity; // Calculate
the total cost of the purchase
                currentWeekPurchaseExpenses += purchaseCost; // Add the
purchase cost to the weekly total

                Console.WriteLine($"Successfully purchased {quantity}
{goodName}, they will be sold for £{sellingPrice} each.");
                // Add the goods to the storage table
                AddGoodsToStorage(playerStore.UserName,
market.GetGoodId(goodName), goodName,
(int)market.GetStorageType(goodName), quantity, sellingPrice, cycleCount);
            }
            else
            {
                Console.WriteLine("Purchase failed due to lack of storage or
insufficient funds.");
            }

            Console.ReadKey();
        }
    }



    private void ViewStoragePhase()
    {
        while (true)
        {   // Menu to display storage status
            Console.Clear();
            Console.ForegroundColor = ConsoleColor.Yellow;
            Console.WriteLine("=== View Storage ===");
            Console.ResetColor();
            playerStore.DisplayStorageStatus();

            Console.WriteLine("Enter the name of the storage to view specific
goods (e.g., 'chilled').");
```

```
            Console.WriteLine("Enter '(b)ack' to return to the previous
menu.");

            string storageChoice = Console.ReadLine().Trim().ToLower();

            if (storageChoice == "back" || storageChoice == "b")
            {
                break; // Return to the main setup menu.
            }

            // Check if the entered storage type is valid.
            if (Enum.TryParse(storageChoice, true, out StorageType
storageType))
            {
                playerStore.DisplayStorage(storageType); // Display goods in
the selected storage.
                Console.WriteLine("Would you like to remove any goods from
this storage? Y/N"); // working on this
                string yesorno = Console.ReadLine().Trim().ToLower();
                if (yesorno == "y")
                {
                    Console.WriteLine("Please enter the name of the good you'd
like to remove");
                    string remove = Console.ReadLine().Trim().ToLower();

                }
                else if (yesorno  == "n")
                {
                    ViewStoragePhase();
                }
                else
                {
                    Console.WriteLine("Invalid option, please enter Y to remove
items from storage or N to not");
                    Console.ReadKey();
                }
            }
```

```
            else
            {
                Console.WriteLine("Invalid storage type. Please try again.");
                Console.ReadKey();
            }

            Console.WriteLine("Press any key to continue...");
            Console.ReadKey(); // Wait for player input before returning.
        }
    }



    private void SimulationPhase()
    {
        Console.Clear();
        Console.ForegroundColor = ConsoleColor.Yellow;
        Console.WriteLine("=== Simulation Phase ===");
        Console.ResetColor();

        decimal revenueThisCycle = playerStore.SimulateSales(market); //
Simulate sales.
        currentWeekSalesRevenue += revenueThisCycle; // Add the revenue
to the weekly total.
        playerStore.DisplayStatus(); // Display cash and inventory
        market.UpdateMarketPrice(); // Update market prices for the next
cycle.
    }
    private void FinanceMenu()
    {
        Console.Clear();
        Console.ForegroundColor = ConsoleColor.Yellow;
        Console.WriteLine("=== Finance Menu ===");
        Console.ResetColor();
        Console.WriteLine("What would you like to see?");
        Console.WriteLine("(C)urrent Profit/Loss sheet");
```

```
        Console.WriteLine("(P)revious Profit/Loss sheet"); // will maybe try and allow the player to select which week they want to see
        Console.WriteLine("(T)otal Profit/Loss sheet");
        Console.WriteLine("Or enter (B)ack to go back");

        string sheet = Console.ReadLine().Trim().ToLower();

        switch(sheet)
        {
            case "current":
            case "c":
                CurrentSheet();
                break;
            case "previous":
            case "p":
                PreviousSheets();
                break;
            case "total":
            case "t":
                TotalSheet();
                break;
            case "back":
            case "b":
                SetupPhase();
                break;
            default:
                Console.WriteLine("Invalid option, please try again");
                Console.ReadKey();
                break;
        }
    }
    private void DisplayPortfolio(int weekIndex)
    {
        if (weekIndex < 0 || weekIndex >= weeklyFinances.Count) // checks if the week index is less than 0 or greater than the number of weeks
        {
            Console.WriteLine("No data available for the requested week.");
```

```
            Console.ReadKey();
            return;
        }

        WeeklyFinance weekData = weeklyFinances[weekIndex];
        Console.Clear();
        Console.ForegroundColor = ConsoleColor.Yellow;
        Console.WriteLine($"=== Profit/Loss Report for Week
{weekData.Week} ===");
        Console.ResetColor();
        Console.ForegroundColor = ConsoleColor.Green;
        Console.WriteLine("Revenue:");
        Console.ResetColor();
        Console.WriteLine($"Sales: £{weekData.SalesRevenue:0.00}");
        Console.ForegroundColor = ConsoleColor.Red;
        Console.WriteLine("Expenditures:");
        Console.ResetColor();
        Console.WriteLine($"  Purchases:
£{weekData.PurchaseExpenses:0.00}");
        Console.WriteLine($"  Bills: £{weekData.BillsExpenses:0.00}");
        Console.WriteLine($"  Upgrades:
£{weekData.UpgradeExpenses:0.00}");
        Console.WriteLine($"Net Income: £{weekData.NetIncome:0.00}");

        if (weekIndex > 0)
        {
            decimal prevNetIncome = weeklyFinances[weekIndex -
1].NetIncome; // gets the net income from the previous week
            if (prevNetIncome != 0)
            {
                decimal percentageChange = ((weekData.NetIncome -
prevNetIncome) / Math.Abs(prevNetIncome)) * 100; // calculates the
percentage change
                Console.WriteLine($"Change from previous week:
{percentageChange:+0.00;-0.00}%");
            }
            else
```

```
            {
                Console.WriteLine("Change from previous week: N/A (previous
net income was £0.00)");
            }
        }
        Console.WriteLine("\nPress any key to return...");
        Console.ReadKey();
    }

    private void CurrentSheet()
    {
        Console.Clear();
        Console.ForegroundColor = ConsoleColor.Yellow;
        Console.WriteLine("=== Current Profit/Loss sheet ===");
        Console.ResetColor();
        if (weeklyFinances.Count == 0)
        {
            Console.WriteLine("No data to display");
        }
        else
        {
            WeeklyFinance currentWeek = weeklyFinances.Last();
            Console.WriteLine($"Week: {currentWeek.Week}");
            Console.ForegroundColor = ConsoleColor.Green;
            Console.WriteLine("Revenue: ");
            Console.ResetColor();
            Console.WriteLine($"Sales: £{currentWeek.SalesRevenue:0.00}");
            Console.ForegroundColor = ConsoleColor.Red;
            Console.WriteLine("Expenditures:");
            Console.ResetColor();
            Console.WriteLine($"Purchases:
£{currentWeek.PurchaseExpenses:0.00}");
            Console.WriteLine($"Bills: £{currentWeek.BillsExpenses:0.00}");
            Console.WriteLine($"Upgrades:
£{currentWeek.UpgradeExpenses:0.00}");
            decimal profit = currentWeek.SalesRevenue -
(currentWeek.PurchaseExpenses + currentWeek.BillsExpenses);
```

```
            Console.ForegroundColor = ConsoleColor.Yellow;
            Console.WriteLine($"Overall profit/loss for the week:
£{profit:0.00}");
            Console.ResetColor();
            if (profit > 0)
            {
                Console.ForegroundColor = ConsoleColor.Green;
                Console.WriteLine("Profit");
                Console.ResetColor();
            }
            else
            {
                Console.ForegroundColor = ConsoleColor.Red;
                Console.WriteLine("Loss");
                Console.ResetColor();
            }
            if (weeklyFinances.Count > 1)
            {
                WeeklyFinance previousWeek =
weeklyFinances[weeklyFinances.Count - 2];
                if (previousWeek.NetIncome != 0)
                {
                    decimal percentChange = ((profit -
previousWeek.NetIncome) / previousWeek.NetIncome) * 100;
                    if (percentChange > 0)
                    {
                        Console.ForegroundColor = ConsoleColor.Green;
                        Console.WriteLine($"Profit change from previous week:
{percentChange:0.00}%");
                        Console.ResetColor();
                    }
                    else
                    {
                        Console.ForegroundColor = ConsoleColor.Red;
                        Console.WriteLine($"Loss change from previous week:
{percentChange:0.00}%");
                        Console.ResetColor();
```

```
                }

            }
            else
            {
                Console.WriteLine("No previous data to compare to");
            }

        }
    }
    Console.ReadKey();
}
private void PreviousSheets()
{
    Console.Clear();
    if (weeklyFinances.Count > 1 )
    {
        DisplayPortfolio(weeklyFinances.Count - 2);
    }
    else
    {
        Console.WriteLine("No previous data to display");
        Console.ReadKey();
    }
}
private void TotalSheet()
{
    Console.Clear();
    Console.ForegroundColor = ConsoleColor.Yellow;
    Console.WriteLine("=== Grand Total Profit/Loss Sheet ===");
    Console.ResetColor();

    if (weeklyFinances.Count == 0)
    {
        Console.WriteLine("No financial data available.");
        Console.ReadKey();
        return;
```

```
        }

        // Calculate overall totals from all weekly records.
        decimal totalSales = weeklyFinances.Sum(w => w.SalesRevenue);
        decimal totalPurchases = weeklyFinances.Sum(w =>
w.PurchaseExpenses);
        decimal totalBills = weeklyFinances.Sum(w => w.BillsExpenses);
        decimal totalUpgrades = weeklyFinances.Sum(w =>
w.UpgradeExpenses);
        decimal totalNetIncome = totalSales - (totalPurchases + totalBills +
totalUpgrades);

        // Display cumulative totals.
        Console.ForegroundColor = ConsoleColor.Green;
        Console.WriteLine("Total Revenue: ");
        Console.ResetColor();
        Console.WriteLine($"Total Sales Revenue: £{totalSales:0.00}");
        Console.ForegroundColor = ConsoleColor.Red;
        Console.WriteLine("Total Expenditures: ");
        Console.ResetColor();
        Console.WriteLine($"Total Purchase Expenses:
£{totalPurchases:0.00}");
        Console.WriteLine($"Total Bills Expenses: £{totalBills:0.00}");
        Console.WriteLine($"Total Upgrade Expenses:
£{totalUpgrades:0.00}");
        Console.WriteLine($"Overall Net Income: £{totalNetIncome:0.00}");

        // Display overall profit or loss
        if (totalNetIncome >= 0)
        {
            Console.ForegroundColor = ConsoleColor.Green;
            Console.WriteLine("The business has been making an overall
profit.");
        }
        else
        {
            Console.ForegroundColor = ConsoleColor.Red;
```

```
                Console.WriteLine("The business has been making an overall loss.");
            }
            Console.ResetColor();
            Console.WriteLine("\nPress any key to return...");
            Console.ReadKey();
        }
    }

    public class Upgrades // loigc to make upgrades function
    {
        public string Name { get; private set; } // gets the name of the upgrade
        public decimal Price { get; private set; } // gets the price of the
upgrade
        public string Description { get; private set; } // gets the description of
the upgrade
        public Action<Store> Effect { get; private set; } // gets the effect of the
upgrade



        public Upgrades(string name, decimal price, string description,
Action<Store> effect)
        {
            Name = name; // sets the name of the upgrade
            Price = price; // sets the price of the upgrade
            Description = description; // sets the description of the upgrade
            Effect = effect; // sets the effect of the upgrade
        }

    }
    public class Store
    {
        public string UserName {  get; private set; } // gets the username
        public decimal Cash { get;  set; } // gets the amount of cash available

        public Dictionary<StorageType, List<Product>> storageAreas; //
dictionary to store the storage areas
```

```
    public const int MaxStorageCapacity = 100; // sets the max storage
capacity (might change this with an upgrade)

    private List<Upgrades> purchasedUpgrades;
    private HashSet<string> ownedUpgrades = new HashSet<string>();




    public Store(decimal initialCash, string userName)
    {
        Cash = initialCash;
        UserName = userName;
        storageAreas = new Dictionary<StorageType, List<Product>>();
        purchasedUpgrades = new List<Upgrades>();

        // Ensures all storage types are initialized when a new store is
created
        storageAreas[StorageType.Fresh] = new List<Product>();
        storageAreas[StorageType.Chilled] = new List<Product>();
        storageAreas[StorageType.Frozen] = new List<Product>();
        storageAreas[StorageType.Regular] = new List<Product>();
    }
    public void AddProductToStorage(StorageType storageType, Product
product)
    {
        if (!storageAreas.ContainsKey(storageType))
        {
            storageAreas[storageType] = new List<Product>();
        }

        storageAreas[storageType].Add(product);

        //Console.WriteLine($"DEBUG: Added product '{product.Name}' to
{storageType} storage.");
    }
```

```
    public void AdjustCash(decimal Amount)
    {
        Cash += Amount;
    }
    public void AddUpgrade(string upgradeName)
    {
        if (!ownedUpgrades.Contains(upgradeName))
        {
            ownedUpgrades.Add(upgradeName);
        }
    }


    public bool HasUpgrade(string upgradeName)
    {
        return ownedUpgrades.Contains(upgradeName);
    }
    public bool BuyProduct(Product product)
    {
        // Ensure the storage exists and the storage type is correct
        if (!storageAreas.ContainsKey(product.StorageType))
        {
            Console.WriteLine($"Error: Storage type {product.StorageType}
not found.");
            return false;
        }

        decimal totalCost = product.PurchasePrice * product.Quantity;
        int currentStorageQuantity =
GetCurrentStorageQuantity(product.StorageType);

        // Check if there's enough cash and space in storage
        if (Cash >= totalCost && (currentStorageQuantity +
product.Quantity) <= MaxStorageCapacity)
        {
            Cash -= totalCost; // Deduct the purchase cost
```

```
            storageAreas[product.StorageType].Add(product); // Add the
product to storage
            return true; // Purchase successful
        }

        Console.WriteLine("Purchase failed: Not enough cash or storage
space.");
        return false; // Purchase failed
    }


    public int GetCurrentStorageQuantity(StorageType storageType)
    {
        int totalQuantity = 0;
        foreach (var product in storageAreas[storageType])
        {
            totalQuantity += product.Quantity; // Sum up all quantities in the
storage.
        }
        return totalQuantity;
    }

    public decimal SimulateSales(Market market)
    {
        decimal totalRevenue = 0;
        foreach (var storageArea in storageAreas)
        {
            // Check if the storage area exists before iterating over it
            if (storageAreas.ContainsKey(storageArea.Key))
            {
                foreach (var product in storageArea.Value)
                {
                    int sold = market.SimulateProductSales(product);
                    decimal revenue = sold * product.SellingPrice;

                    Console.WriteLine($"{sold} units of {product.Name} sold at
£{product.SellingPrice} each. Total: £{revenue}");
```

```
                    Cash += revenue; // Add the revenue to the cash.
                    product.Quantity -= sold; // Reduce the quantity in storage.
                    totalRevenue += revenue; // Add the revenue to the total.
                }
            }
            else
            {
                Console.WriteLine($"Storage type {storageArea.Key} does not
exist.");
            }
        }
        return totalRevenue;
    }

    public void DisplayCash() // used for select circumstances
    {
        if (Cash < 500)
        {
            Console.ForegroundColor = ConsoleColor.Red;
            Console.WriteLine($"Cash: £{Cash}");
            Console.ResetColor();
        }
        else
        {
            Console.ForegroundColor = ConsoleColor.Green;
            Console.WriteLine($"Cash: £{Cash}");
            Console.ResetColor();
        }
    }
    public void DisplayStatus() // used to show the user what is in each
storage/how much
    {
        if (Cash < 500)
        {
            Console.ForegroundColor = ConsoleColor.Red;
            Console.WriteLine($"Cash: £{Cash}");
            Console.ResetColor();
```

```
            }
            else
            {
                Console.ForegroundColor = ConsoleColor.Green;
                Console.WriteLine($"Cash: £{Cash}");
                Console.ResetColor();
            }
            Console.ForegroundColor = ConsoleColor.Blue;
            Console.WriteLine($"Chilled Storage:
{GetCurrentStorageQuantity(StorageType.Chilled)} units");
            Console.ResetColor();
            Console.ForegroundColor = ConsoleColor.Cyan;
            Console.WriteLine($"Frozen Storage:
{GetCurrentStorageQuantity(StorageType.Frozen)} units");
            Console.ResetColor();
            Console.ForegroundColor = ConsoleColor.Magenta;
            Console.WriteLine($"Regular Storage:
{GetCurrentStorageQuantity(StorageType.Regular)} units");
            Console.ResetColor();
            Console.ForegroundColor = ConsoleColor.DarkGreen;
            Console.WriteLine($"Fresh Storage:
{GetCurrentStorageQuantity(StorageType.Fresh)} units");
            Console.ResetColor();
        }
        public void DisplayStorageStatus()
        {
            // Check if each storage type exists in the dictionary before
displaying
            if (storageAreas.ContainsKey(StorageType.Fresh))
                Console.WriteLine($"Fresh:
{GetCurrentStorageQuantity(StorageType.Fresh)} / {MaxStorageCapacity}
units");
            else
                Console.WriteLine("Fresh storage area does not exist.");

            if (storageAreas.ContainsKey(StorageType.Chilled))
```

```
            Console.WriteLine($"Chilled:
{GetCurrentStorageQuantity(StorageType.Chilled)} / {MaxStorageCapacity}
units");
        else
            Console.WriteLine("Chilled storage area does not exist.");

        if (storageAreas.ContainsKey(StorageType.Frozen))
            Console.WriteLine($"Frozen:
{GetCurrentStorageQuantity(StorageType.Frozen)} / {MaxStorageCapacity}
units");
        else
            Console.WriteLine("Frozen storage area does not exist.");

        if (storageAreas.ContainsKey(StorageType.Regular))
            Console.WriteLine($"Regular:
{GetCurrentStorageQuantity(StorageType.Regular)} / {MaxStorageCapacity}
units");
        else
            Console.WriteLine("Regular storage area does not exist.");
    }

    public void DisplayStorage(StorageType storageType)
    {
        // Check if the storage type exists in the dictionary before trying to
access it
        if (storageAreas.ContainsKey(storageType))
        {
          if (storageType == StorageType.Fresh)
          {
              Console.ForegroundColor = ConsoleColor.DarkGreen;
          }
          else if (storageType == StorageType.Chilled)
          {
             Console.ForegroundColor = ConsoleColor.Blue;
          }
          else if (storageType == StorageType.Frozen)
          {
```

```
                Console.ForegroundColor = ConsoleColor.Cyan;
            }
            else if (storageType == StorageType.Regular)
            {
                Console.ForegroundColor = ConsoleColor.Magenta;
            }
            Console.WriteLine($"=== {storageType} Storage ===");
            Console.ResetColor();
            foreach (var product in storageAreas[storageType])
            {
                Console.WriteLine($"{product.Name} - Quantity:
{product.Quantity}, Selling Price: £{product.SellingPrice}");
            }
        }
        else
        {
            // Handle the case where the storage type doesn't exist in the
dictionary
            Console.WriteLine($"Error: Storage type {storageType} does not
exist.");
        }

        Console.ReadKey();
    }


    public bool PayBills(decimal amount)
    {
        if (Cash >= amount)
        {
            Cash -= amount; // Deduct the bill amount from cash.
            Console.WriteLine($"You have paid £{amount} towards bills."); //
Inform the player about bill payment.
            Console.ReadKey();
            return true; // Bills paid successfully.
        }
        return false; // Not enough cash to pay bills.
```

```
        }

        public void CheckForExpiredGoods(int currentCycle)
        {
            var expiredProducts = new List<Product>();

            foreach (var product in storageAreas[StorageType.Chilled])
            {
                // Check if the product has been in storage for more than 2
cycles.
                if (currentCycle - product.CycleAdded >= 2)
                {
                    expiredProducts.Add(product); // Mark the product as expired.
                    Console.WriteLine($"{product.Quantity} units of
{product.Name} have expired."); // Notify the player.
                    Console.ReadKey();
                }
            }

            // Remove all expired products from the chilled storage.
            foreach (var expiredProduct in expiredProducts)
            {
                storageAreas[StorageType.Chilled].Remove(expiredProduct);
            }
        }
    }

    public class Product //used to store information on the various
goods/products in the game
    {
        public string Name { get; private set; }
        public decimal PurchasePrice { get; private set; }
        public decimal SellingPrice { get; set; }
        public int Quantity { get; set; }
        public StorageType StorageType { get; private set; }
        public int CycleAdded { get; private set; }
        public int CycleExpired { get; private set; }
```

```
        public PEDtype Elasticity { get; set; }

        public Product() { }
        public Product(string name, decimal purchasePrice, decimal sellingPrice,
int quantity, StorageType storageType, int cycleAdded)
        {
            Name = name;
            PurchasePrice = purchasePrice;
            SellingPrice = sellingPrice;
            Quantity = quantity;
            StorageType = storageType;
            CycleAdded = cycleAdded;
        }
    }

    public enum StorageType
    {
        Fresh,
        Chilled,
        Frozen,
        Regular
    }
    public enum PEDtype
    {
        StrongElastic,
        WeakElastic,
        StrongInelastic,
        WeakInelastic
    }
    public class Market
    {
        private Dictionary<string, decimal> marketPrices;
        private readonly string ConnectionString;

        public Market(string ConnectionString)
        {
            this.ConnectionString = ConnectionString;
```

```
            marketPrices = new Dictionary<string, decimal>();
            LoadGoodsFromDatabase(); // Fetch goods from the database
        }

        private void LoadGoodsFromDatabase()
        {
            string sql = "SELECT GoodName, PurchasePrice FROM Goods;";

            using (SQLiteConnection conn = new
SQLiteConnection(DataBaseConfig.ConnectionString))
            {
                conn.Open();
                using (SQLiteCommand cmd = new SQLiteCommand(sql, conn))
                {
                    using (SQLiteDataReader reader = cmd.ExecuteReader())
                    {
                        while (reader.Read())
                        {
                            string goodName = reader.GetString(0);
                            decimal purchasePrice = reader.GetDecimal(1);

                            if (!marketPrices.ContainsKey(goodName))
                            {
                                marketPrices[goodName] = purchasePrice;
                            }
                        }
                    }
                }
            }
        }
        public int GetGoodId(string productName)
        {
            string sql = "SELECT Good_Id FROM Goods WHERE GoodName =
@GoodName;";

            using (SQLiteConnection conn = new
SQLiteConnection(DataBaseConfig.ConnectionString))
```

```
        {
            conn.Open();
            using (SQLiteCommand cmd = new SQLiteCommand(sql, conn))
            {
                cmd.Parameters.AddWithValue("@GoodName", productName);
                object result = cmd.ExecuteScalar();

                if (result != null)
                {
                    int goodId = Convert.ToInt32(result);
                    //Console.WriteLine($"DEBUG: Found Good_Id {goodId} for
product '{productName}'.");
                    return goodId;
                }
                else
                {
                    Console.WriteLine($"ERROR: No Good_Id found for product
name '{productName}'.");
                    return -1;
                }
            }
        }
    }



    public decimal GetMarketPrice(string productName)
    {
        return Math.Round(marketPrices.ContainsKey(productName) ?
marketPrices[productName] : 0, 2);

    }

    public void UpdateMarketPrice()
    {
        Random random = new Random();
        var productNames = new List<string>(marketPrices.Keys);
```

```
            foreach (var productName in productNames)
            {
                // Generate a random price change between £0.01 and £0.50
                decimal priceChange =
Math.Round((decimal)(random.NextDouble() * 0.49 + 0.01), 2);

                // Randomly decide whether to increase or decrease the price
                bool increase = random.Next(2) == 0; // 50% chance to increase
or decrease

                if (increase)
                {
                    marketPrices[productName] += priceChange;
                }
                else
                {
                    marketPrices[productName] -= priceChange;

                    // Ensure the price doesn't drop below £0.01
                    if (marketPrices[productName] < 0.01m)
                    {
                        marketPrices[productName] = 0.01m;
                        marketPrices[productName] =
Math.Round(marketPrices[productName], 2);
                    }
                }
            }
        }
        public int SimulateProductSales(Product product)
        {
            decimal marketPrice = GetMarketPrice(product.Name);// Get the
current marketprice of the product by its name
            if (marketPrice == 0) return 0;
            // Calculates price factor (higher selling price compared to market
price means lower price factor)
            double priceFactor = (double)(marketPrice / product.SellingPrice);
```

```
        int maxSales = product.Quantity;// maximum amount of sales possible based on the amount of the product available
        int estimatedSales = (int)(maxSales * priceFactor);// Estimates sales based on the price factor and max quantity
        switch (product.Elasticity)// Changes the logic based on the elasticity of the product
        {
            case PEDtype.StrongElastic: // StrongElastic means the lower the price the higher the sales
                estimatedSales = (int)(estimatedSales * Math.Min(priceFactor, 1.8));// Maximum 80% increase
                break;
            case PEDtype.WeakElastic:// Weak elastic means there will still be more sales if the price is lower but not a massive amount more
                estimatedSales = (int)(estimatedSales * Math.Min(priceFactor, 1.2)); // Maximum 20% increase
                break;
            case PEDtype.WeakInelastic:// Weak inelastic means the price can be put slightly higher and sales will remain similiar but there will be a slight decrease
                estimatedSales = (int)(maxSales * 0.8); // Sets estimated to 80% of the maximum quantity, regardless of price
                break;
            case PEDtype.StrongInelastic: // Strong inelastic means price can be put higher and sales will remain generally unaffected
                estimatedSales = (int)((maxSales * 0.5));// Sets stimated sales to 50% of the maximum quantity
                break;
        }


        return Math.Min(estimatedSales, product.Quantity);
    }
```

```
    public Dictionary<string, decimal> GetGoodsByCategory(string
category) // go through all products, check their storage type and returns
the neccessarry ones
    {
        var availableGoods = new Dictionary<string, decimal>();

        foreach (var product in marketPrices)
        {
            if (GetStorageType(product.Key).ToString().ToLower() ==
category.ToLower())
            {
                availableGoods[product.Key] = product.Value;
            }
        }

        return availableGoods;
    }

    public StorageType GetStorageType(string productName) // defines
what type of storage the good belongs in
    {
        switch (productName.ToLower())
        {
            case "milk":
            case "yoghurt":
            case "steak":
            case "chicken":
            case "bacon":
                return StorageType.Chilled;
            case "strawberries":
            case "carrots":
            case "bananas":
            case "cabbage":
            case "mangos":
                return StorageType.Fresh;
            case "magnums":
            case "cornettos":
```

```
            case "pizza":
            case "turkey":
            case "peas":
                return StorageType.Frozen;
            case "sweets":
            case "chocolate":
            case "crips":
            case "sandwich":
            case "wine":
                return StorageType.Regular;
            default:
                return StorageType.Regular;



        }
    }
    public bool IsElastic(string productName)
    {
        switch (productName.ToLower())
        {
            case "milk":
            case "carrots":
            case "bananas":
            case "cabbage":
            case "mangos":
            case "strawberries":
            case "peas":
                return false; // ineslastic
            case "yoghurt":
            case "steak":
            case "chicken":
            case "bacon":
            case "magnums":
            case "cornettos":
            case "pizza":
            case "turkey":
```

```
                case "sweets":
                case "chocolate":
                case "crips":
                case "sandwich":
                case "wine":
                    return true; //elastic
                default: return false;
            }
        }
    }
    public class WeeklyFinance
    {
        public int Week { get; set; }
        public decimal SalesRevenue { get; set; }
        public decimal PurchaseExpenses { get; set; }
        public decimal BillsExpenses { get; set; }
        public decimal UpgradeExpenses { get; set; }
        public decimal NetIncome => SalesRevenue - (PurchaseExpenses +
BillsExpenses + UpgradeExpenses);
    }

}
```