```
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 databa
    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; // Decalres 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; // trakcs 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:0.00}) 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) * 10;
                    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);
}
```

```
}
```