```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Data.SQLite;
using Microsoft.Identity.Client.Extensions.Msal;
using System.ComponentModel.Design;

namespace BusinessSimulator
{
    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)
        private decimal currentWeekStorageExpenses = 0; // tracks cost of holding goods

        // 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
tablxe.");
        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 RemoveGoods(string UserName, int Good_Id, string ProductName, int
GoodType, int QuantityRemove, decimal SellingPrice, int cyclePurchased)
    {
     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;

        using (SQLiteConnection conn = new
SQLiteConnection(DataBaseConfig.ConnectionString))
        {
          conn.Open();
          using (SQLiteCommand checkCmd = new SQLiteCommand(checkSQL, conn))
          {
            checkCmd.Parameters.AddWithValue("@UserName", UserName);
            checkCmd.Parameters.AddWithValue("@Good_Id", Good_Id);
            checkCmd.Parameters.AddWithValue("@GoodName", ProductName);
            checkCmd.Parameters.AddWithValue("@SellingPrice", SellingPrice);
            checkCmd.Parameters.AddWithValue("@CyclePurchased", cyclePurchased);
            checkCmd.Parameters.AddWithValue("@GoodType", GoodType);

            object result = checkCmd.ExecuteScalar();
            if (result != null)
            {
              currentQuantity = Convert.ToInt32(result);
            }
            else
            {
              Console.WriteLine("ERROR: Product not found in database.");
              return;
            }
          }
          if (QuantityRemove > currentQuantity)
          {
            Console.WriteLine("ERROR: Removal quantity must exceed or be equal to
quantity in storage.");
          }
            if (QuantityRemove == currentQuantity)
            {
              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;";

              using (SQLiteCommand removeCmd = new SQLiteCommand(RemoveSQL,
conn))
              {
                removeCmd.Parameters.AddWithValue("@UserName", UserName);
                removeCmd.Parameters.AddWithValue("@Good_Id", Good_Id);
```

```
                    removeCmd.Parameters.AddWithValue("@GoodName", ProductName);
                    removeCmd.Parameters.AddWithValue("@SellingPrice", SellingPrice);
                    removeCmd.Parameters.AddWithValue("@CyclePurchased",
cyclePurchased);
                    removeCmd.Parameters.AddWithValue("@GoodType", GoodType);
                    removeCmd.ExecuteNonQuery();
                    int rowsAffected = removeCmd.ExecuteNonQuery();
                    if (rowsAffected > 0)
                    {
                       Console.WriteLine($"Successfully removed {QuantityRemove} units of
{ProductName} from storage.");
                    }
                    else
                    {
                    Console.WriteLine("ERROR: Failed to remove product from storage.");
                    }
                 }


              }
            else
            {
                 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;";

              using (SQLiteCommand updateCmd = new SQLiteCommand(UpdateSQL,
conn))
              {
                 updateCmd.Parameters.AddWithValue("@Quantity", newQuantity);
                 updateCmd.Parameters.AddWithValue("@UserName", UserName);
                 updateCmd.Parameters.AddWithValue("@Good_Id", Good_Id);
                 updateCmd.Parameters.AddWithValue("@GoodName", ProductName);
                 updateCmd.Parameters.AddWithValue("@SellingPrice", SellingPrice);
                 updateCmd.Parameters.AddWithValue("@CyclePurchased",
cyclePurchased);
                 updateCmd.Parameters.AddWithValue("@GoodType", GoodType);
                 int rowsAffected = updateCmd.ExecuteNonQuery();
                 if (rowsAffected > 0)
                 {
                    Console.WriteLine($"Successfully removed {QuantityRemove} units of
{ProductName} from storage.");
                 }
                 else
                 {
                    Console.WriteLine("ERROR: Failed to remove product from storage.");
                }
            }
        }
    }
}
private void UpgradesMenu()
{
    Console.Clear();
    Console.WriteLine(" === Upgrades ===");
    for (int i = 0; i < availableUpgrades.Count; i++) // logic to display list of avaialble
upgrades
    {
        var upgrade = availableUpgrades[i]; //wrties the name and price of upgrade
        Console.WriteLine($"{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
                StorageExpenses = currentWeekStorageExpenses // sets storage expenses for
the week
            };
            weeklyFinances.Add(wf);

            //resets ready for the next week
            currentWeekBillsExpenses = 0;
            currentWeekPurchaseExpenses = 0;
            currentWeekSalesRevenue = 0;
            currentWeekUpgradesExpenses = 0;
            currentWeekStorageExpenses = 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)
    {
        while (true) // Loop to restart the purchase process if necessary.
        {
            Console.Clear();
            // Set the color based on category
            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 for upgrade for elasticity insight.
            bool hasElasticityInsight = playerStore.HasUpgrade("Elasticity Insight");
            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();

            // Prompt for the good name.
            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")
            {
                return; // Go back to the previous menu.
            }
            if (!availableGoods.ContainsKey(goodName))
            {
                Console.WriteLine("Good not recognized. Please try again.");
```

```
                    Console.ReadKey();
                    continue; // Restart the loop.
                }

                // Retrieve the market purchase price for the selected good.
                decimal regPurchasePrice = market.GetMarketPrice(goodName);
                decimal purchasePrice = regPurchasePrice;

                // Prompt for quantity.
                Console.Write($"Enter the quantity of {goodName} to buy (or type 'back' to
cancel): ");
                string preQuantity = Console.ReadLine().Trim().ToLower();
                if (preQuantity == "back" || preQuantity == "b")
                {
                    continue; // Restart the process.
                }
                if (!int.TryParse(preQuantity, out int quantity))
                {
                    Console.WriteLine("Invalid quantity input. Please try again.");
                    Console.ReadKey();
                    continue; // Restart the process.
                }
                Console.WriteLine($"{quantity} {goodName} selected.");

                // Prompt for selling price.
                Console.Write($"Enter the selling price for {goodName} (or type 'back' to cancel):
");
                string prePrice = Console.ReadLine().Trim().ToLower();
                if (prePrice == "back" || prePrice == "b")
                {
                    continue; // Restart the process.
                }
                if (!decimal.TryParse(prePrice, out decimal sellingPrice))
                {
                    Console.WriteLine("Invalid price input. Please try again.");
                    Console.ReadKey();
                    continue; // Restart the process.
                }
                if (sellingPrice == 0)
                {
                    Console.WriteLine("Selling price cannot be zero. Please try again.");
                    Console.ReadKey();
                    continue; // Restart the process.
                }

                // Calculate the total purchase cost.
                decimal purchaseCost = purchasePrice * quantity;
```

```
            // Confirm purchase with the user.
            Console.WriteLine($"You are about to sell {goodName} for £{sellingPrice} each.");
            Console.WriteLine($"This will cost you £{purchaseCost}.");
            Console.WriteLine($"After the purchase you will have £{playerStore.Cash -
purchaseCost} remaining.");
            Console.Write("Press Enter to confirm or type 'n' to cancel: ");
            string confirmInput = Console.ReadLine().Trim().ToLower();
            if (confirmInput == "n")
            {
               // User canceled the purchase. Reset and start over.
               continue;
            }

            // Create the product with the given details.
            Product product = new Product(goodName, regPurchasePrice, sellingPrice,
quantity, market.GetStorageType(goodName), cycleCount);
            if (playerStore.BuyProduct(product))
            {
               currentWeekPurchaseExpenses += purchaseCost; // Update the weekly
purchase expenses.
               Console.WriteLine($"Successfully purchased {quantity} {goodName} to be sold
at £{sellingPrice} each.");
               // Add the purchased goods to storage.
               AddGoodsToStorage(playerStore.UserName, market.GetGoodId(goodName),
goodName, (int)market.GetStorageType(goodName), quantity, sellingPrice, cycleCount);
            }
            else
            {
               Console.WriteLine("Purchase failed due to insufficient funds or storage space.");
            }
            Console.ReadKey();
            return; // End the method after a purchase attempt.
         }
      }

      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 == "n")
                {
                    continue;
                }
                else if (yesorno == "y")
                {
                    Console.WriteLine("Please enter the name of the good you'd like to remove");
                    string GoodRemove = Console.ReadLine().ToLower();
                    Console.WriteLine("Please enter the selling price of the good you'd like to
remove");
                    string prePrice = Console.ReadLine().Trim();
                    if (!decimal.TryParse(prePrice, out decimal sellingPrice))
                    {
                        Console.WriteLine("Invalid price, please try again");
                        Console.ReadKey();
                        return;
                    }
                    Console.WriteLine("Please enter the quantity of the good you'd like to
remove");
                    string QuantityRemove = Console.ReadLine().ToLower();
                    if (!int.TryParse(QuantityRemove, out int quantityRemove))
                    {
                        Console.WriteLine("Invalid quantity, please try again");
                        Console.ReadKey();
                        return;
                    }
                    if (playerStore.storageAreas.ContainsKey(storageType))
                    {
                        var product = playerStore.storageAreas[storageType].FirstOrDefault(p =>
p.Name.ToLower() == GoodRemove && p.SellingPrice == sellingPrice);
                        if (product != null)
                        {
```

```
                        if (quantityRemove > product.Quantity)
                        {
                            Console.WriteLine("Removal quantity must exceed or be equal to
quantity in storage");
                            Console.ReadKey();
                        }
                        else
                        {
                            if (quantityRemove == product.Quantity)
                            {
                                playerStore.storageAreas[storageType].Remove(product);

                            }
                            else
                            {
                                product.Quantity -= quantityRemove;
                            }
                            int goodId = market.GetGoodId(product.Name);
                            RemoveGoods(playerStore.UserName, goodId, product.Name,
(int)product.StorageType, quantityRemove, sellingPrice, product.CycleAdded);
                        }
                    }
                    else
                    {
                        Console.WriteLine("Product not found in storage");
                        Console.ReadKey();
                    }


                }
                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.
            decimal storageExpenses = playerStore.CalculateStorageExpenses();
            playerStore.AdjustCash(-storageExpenses); // deduct the current cost of holding
goods
            currentWeekStorageExpenses += storageExpenses; // add this for the weekly
balance sheet
            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($"Upgrades: £{weekData.UpgradeExpenses:0.00}");
         Console.WriteLine($"Cost of holding goods: £{weekData.StorageExpenses: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($"Cost of holding goods:
£{currentWeek.StorageExpenses:0.00}");
            Console.WriteLine($"Upgrades: £{currentWeek.UpgradeExpenses:0.00}");

            decimal expenses = (currentWeek.PurchaseExpenses +
currentWeek.BillsExpenses + currentWeek.UpgradeExpenses +
currentWeek.StorageExpenses);
            decimal profit = currentWeek.SalesRevenue - expenses;
            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 totalStorageCosts = weeklyFinances.Sum(w => w.StorageExpenses);
            decimal totalNetIncome = totalSales - (totalPurchases + totalBills + totalUpgrades +
totalStorageCosts);


            // 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($"Total Upgrade Expenses: £{totalUpgrades:0.00}");
            Console.WriteLine($"Total Storage Expenses: £{totalStorageCosts: 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();
        }
    }
```