Eric Gathinji

Programming in C# CST-150-0500

Grand Canyon University

17th July 2025

Milestone 6

Github link: https://github.com/Ericgathinji444/Inventory-App-6
Video link: https://youtu.be/Xjt_vp6DnXQ?si=ZLT3_BQqUYcBdYx3

UPDATED FLOWCHART

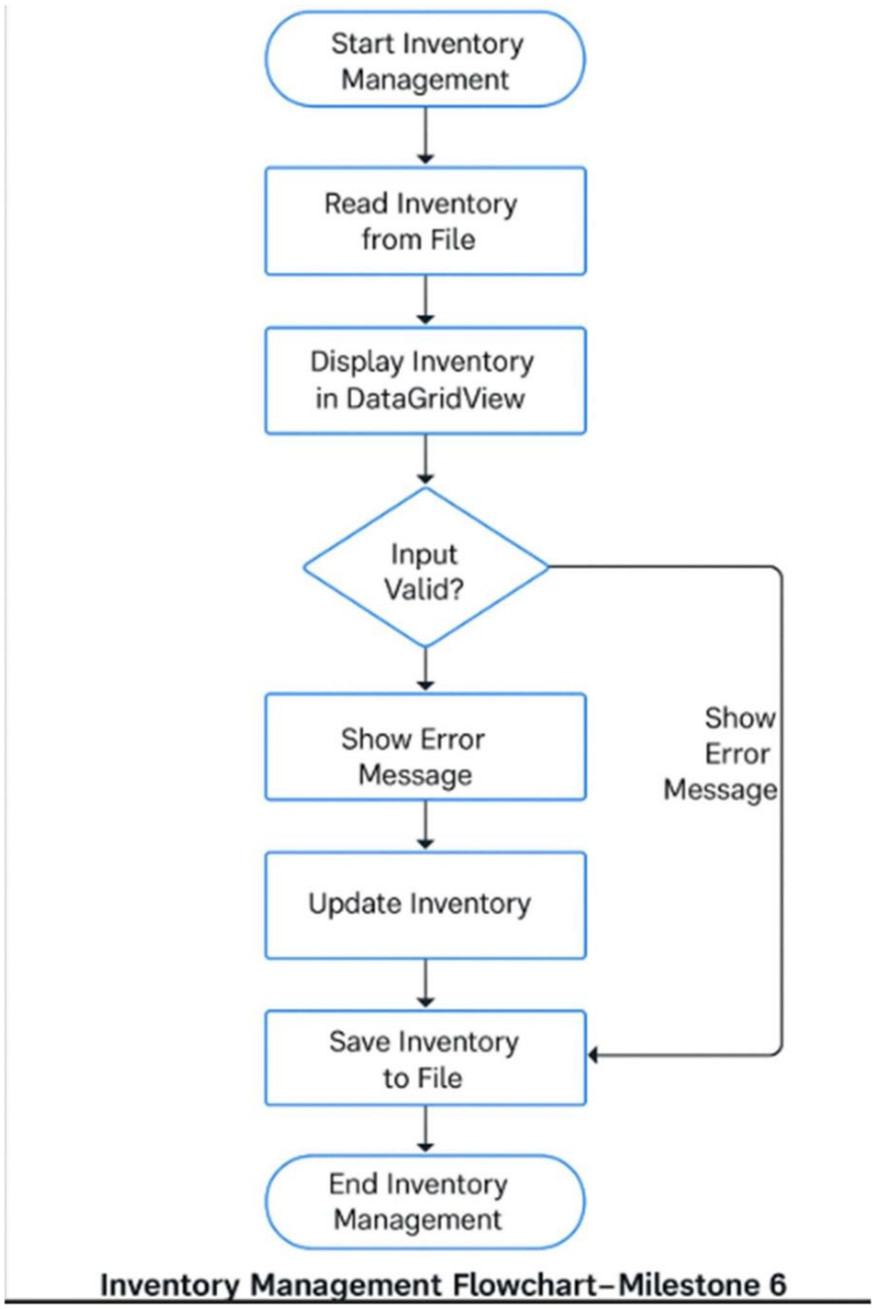


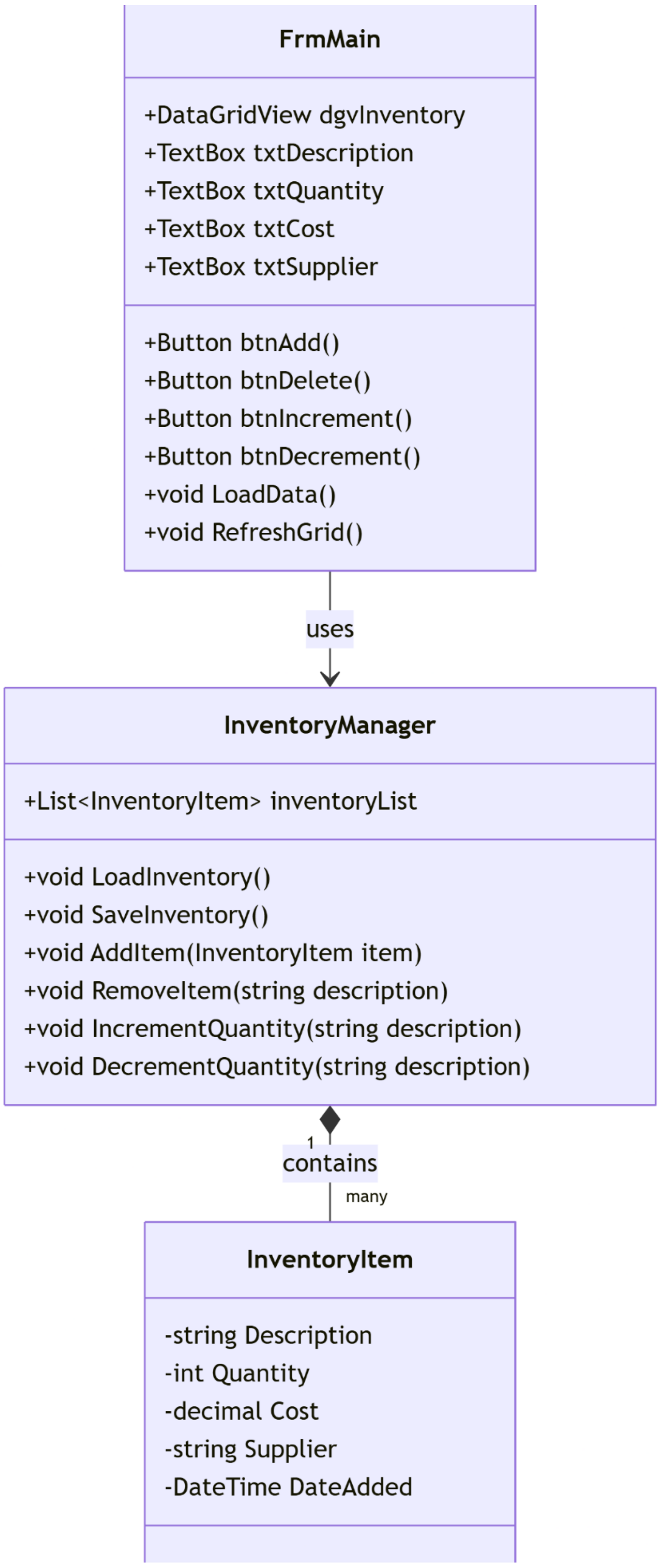
Figure 1: Updated flowchart

This flowchart outlines the logic of the Inventory Management application, capturing the steps of reading, displaying, modifying, and persisting inventory data using a List<T> and a DataGridView

EXPLANATION;

The flowchart begins with reading the Inventory.txt file into a List<InventoryItem>. Then, it populates the DataGridView to display all items. The user can choose to add, delete, increment, or decrement an inventory item using the corresponding buttons. After every operation, the List and file are updated to reflect changes. This logical sequence ensures data integrity and a user-friendly interaction loop.

UML CLASS DIAGRAM



UML CLASS DIAGRAM

Figure 2: Inventory Management System Class Diagram

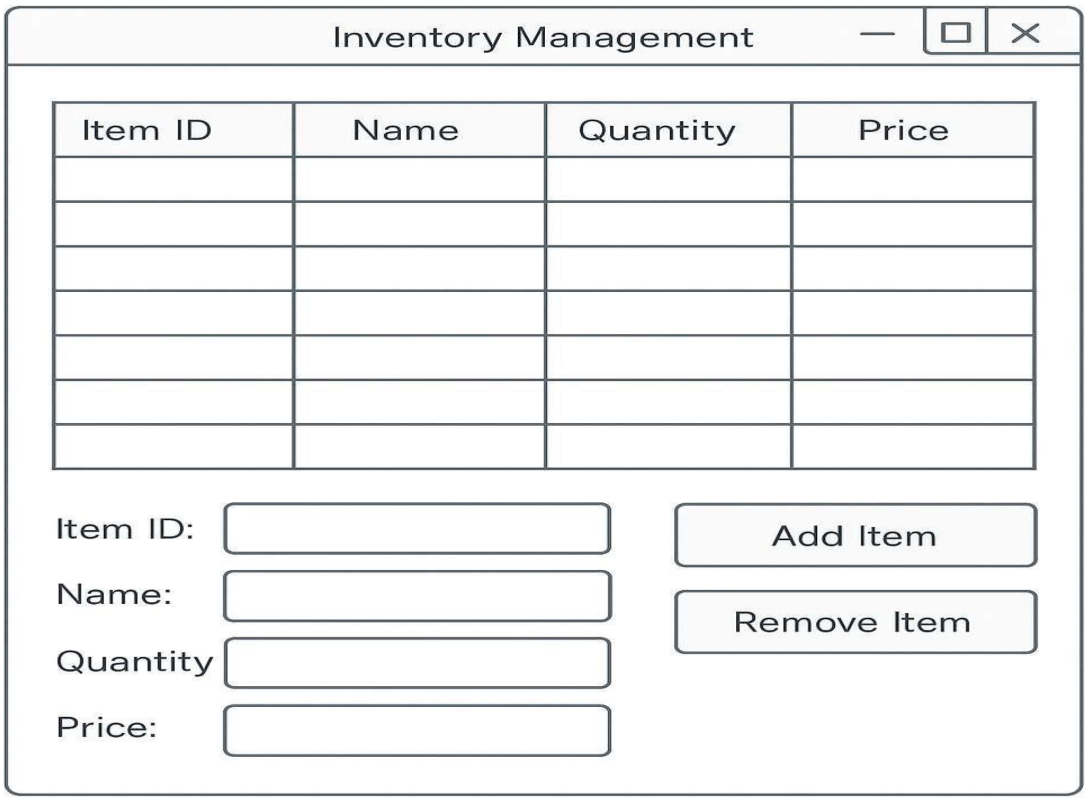
The three primary parts of an inventory management system are depicted in this diagram:

FrmMain: The user interface for displaying inventory items that includes buttons, input forms, and a data grid.

The main logic class for adding, editing, loading, and saving inventory items is called InventoryManager.

A single inventory entry is represented by the data class InventoryItem, which has attributes like cost, quantity, and description.

UPDATED WIREFRAME



Item ID	Name	Quantity	Price

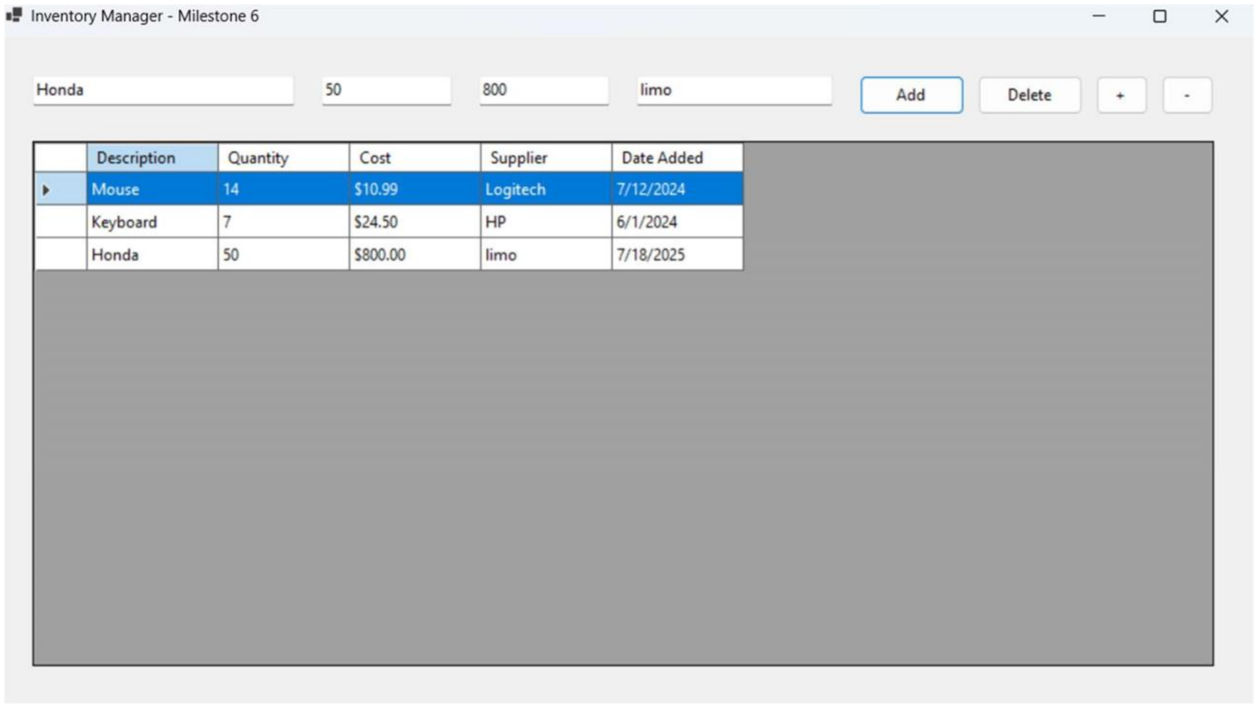
Inventory Management Application Wireframe

Figure 3:This wireframe represents the UI layout of the Inventory Management system using Windows Forms and DataGridView.

EXPLANATION;

The design includes four labeled text boxes (Descrip on, Quan ty, Cost, Supplier) for adding new inventory. There are four ac on bu ons: Add, Delete, Increment, and Decrement. Below is a wellstructured DataGridView to display five columns: Descrip on, Quan ty, Cost, Supplier, and DateAdded. The design allows intui ve interac on with minimal user confusion, mee ng the usability goals for Milestone 6.

INVENTORY DISPLAY



	Description	Quantity	Cost	Supplier	Date Added
▸	Mouse	14	$10.99	Logitech	7/12/2024
	Keyboard	7	$24.50	HP	6/1/2024
	Honda	50	$800.00	limo	7/18/2025

Figure 4:The DataGridView displays the inventory list with properly formatted column headers: Description, Quan ty, Cost, Supplier, and DateAdded.

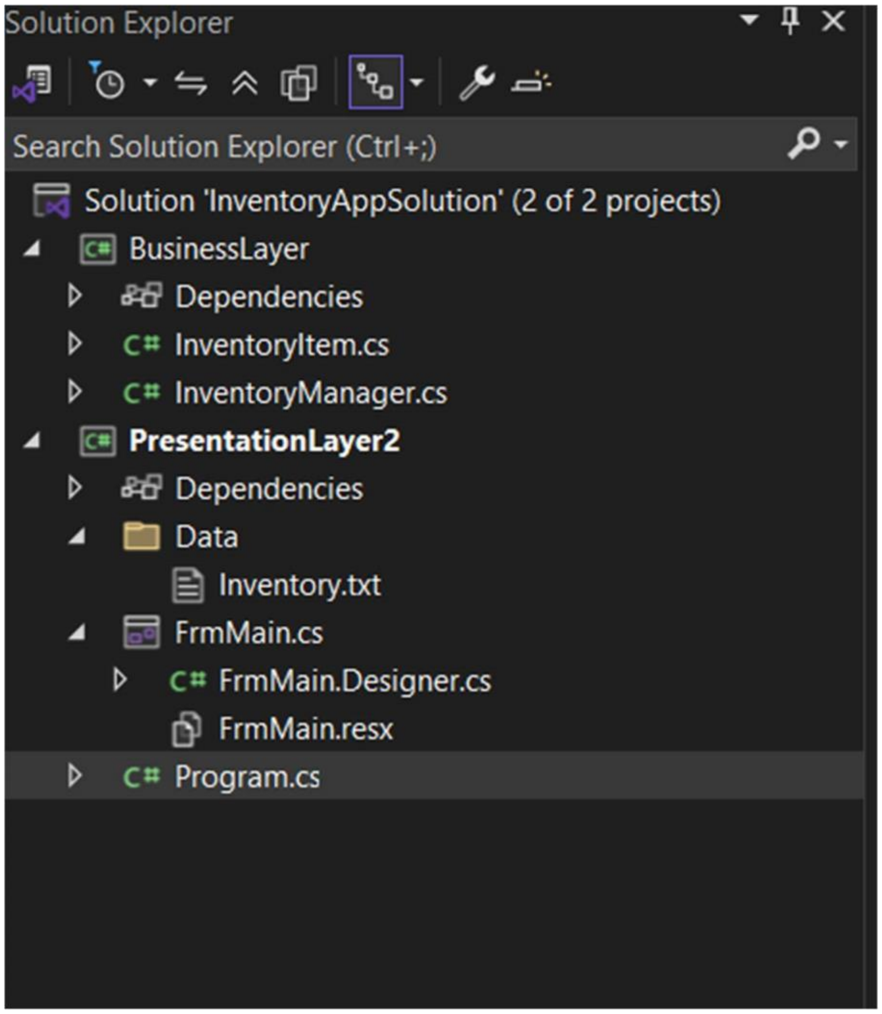
EXPLANATION;

Each inventory item is shown in its row with values properly formatted.

Numbers are left-justified, and the Cost field displays a dollar sign with two decimal places (e.g., $12.50).

This clean formatting ensures users can clearly understand inventory data and its financial impact at a glance.

SCREENSHOT OF THE SOLUTION EXPLORER



```
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'InventoryAppSolution' (2 of 2 projects)
  BusinessLayer
    Dependencies
    InventoryItem.cs
    InventoryManager.cs
  PresentationLayer2
    Dependencies
    Data
      Inventory.txt
    FrmMain.cs
      FrmMain.Designer.cs
      FrmMain.resx
    Program.cs
```

Figure 5:The solution is organized into an N-Layer structure with separate projects for Business Logic and Presentation.

EXPLANATION;

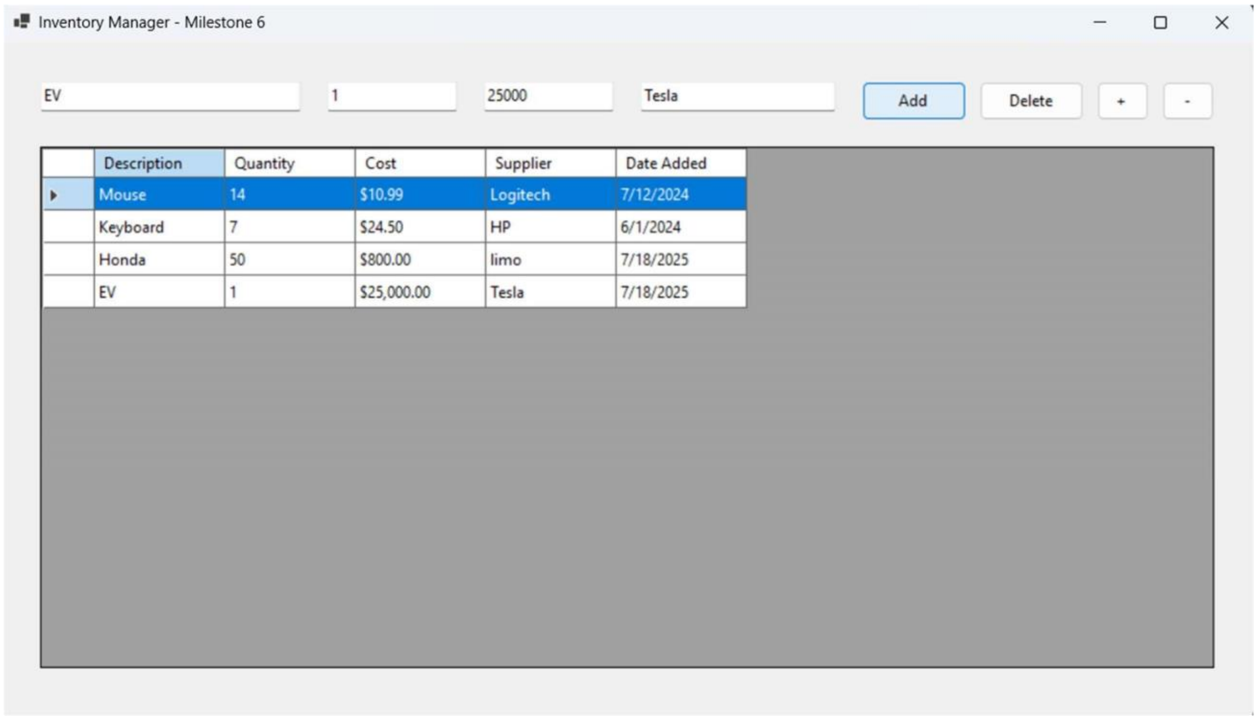
The InventoryAppSolu on contains:

BusinessLayer – Contains InventoryManager.cs and InventoryItem.cs

Presenta onLayer – Contains FrmMain.cs, FrmMain.Designer.cs, and Program.cs

A Data folder under PresentationLayer holds the Inventory.txt file.
This separation improves maintainability and aligns with best practices in layered architecture.

SCREENSHOT OF THE FORM AFTER BEING POPULATED



EV	1	25000	Tesla

Description	Quantity	Cost	Supplier	Date Added
Mouse	14	$10.99	Logitech	7/12/2024
Keyboard	7	$24.50	HP	6/1/2024
Honda	50	$800.00	limo	7/18/2025
EV	1	$25,000.00	Tesla	7/18/2025

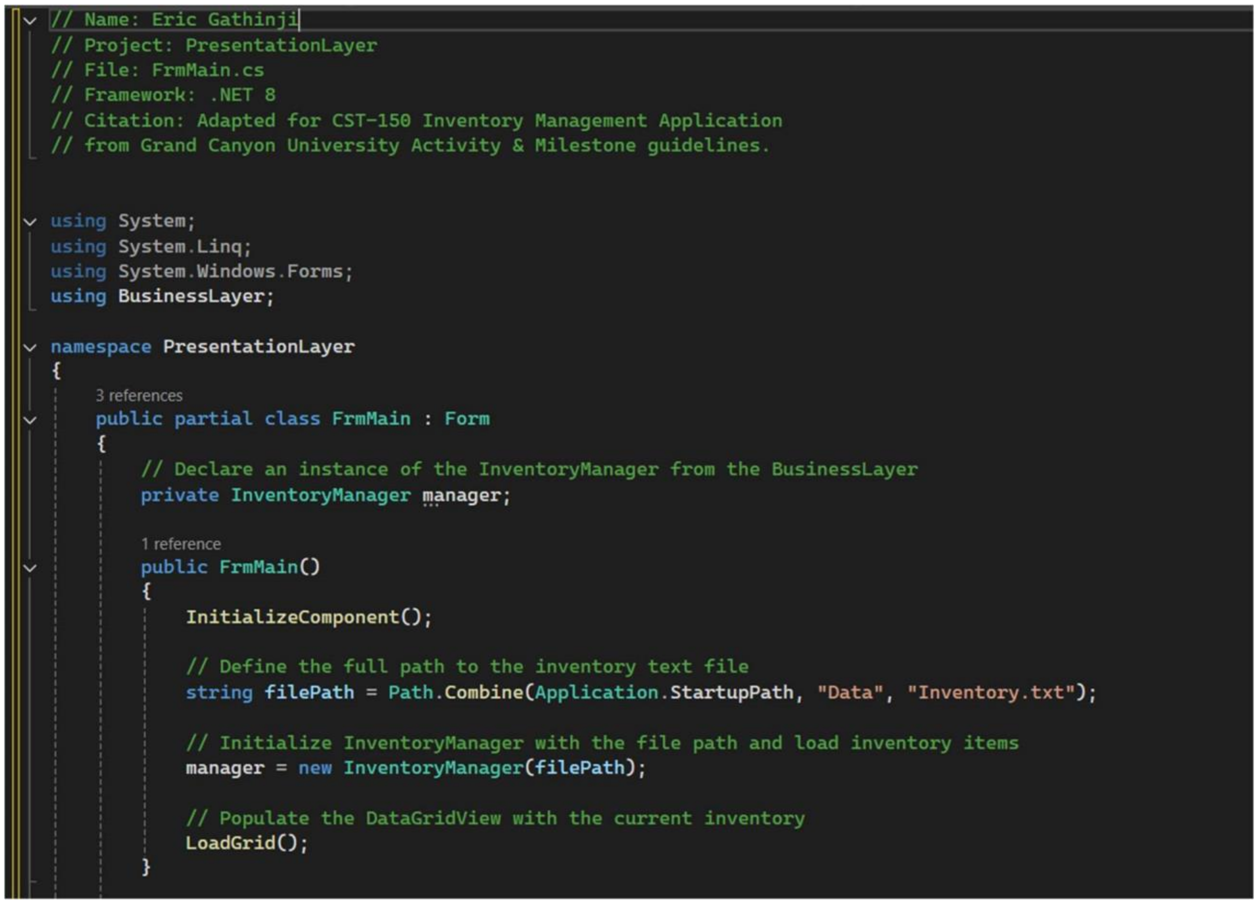
Figure 6: The form displays all inventory items after the application reads and loads the Inventory.txt data into the DataGridView.

EXPLANATION;

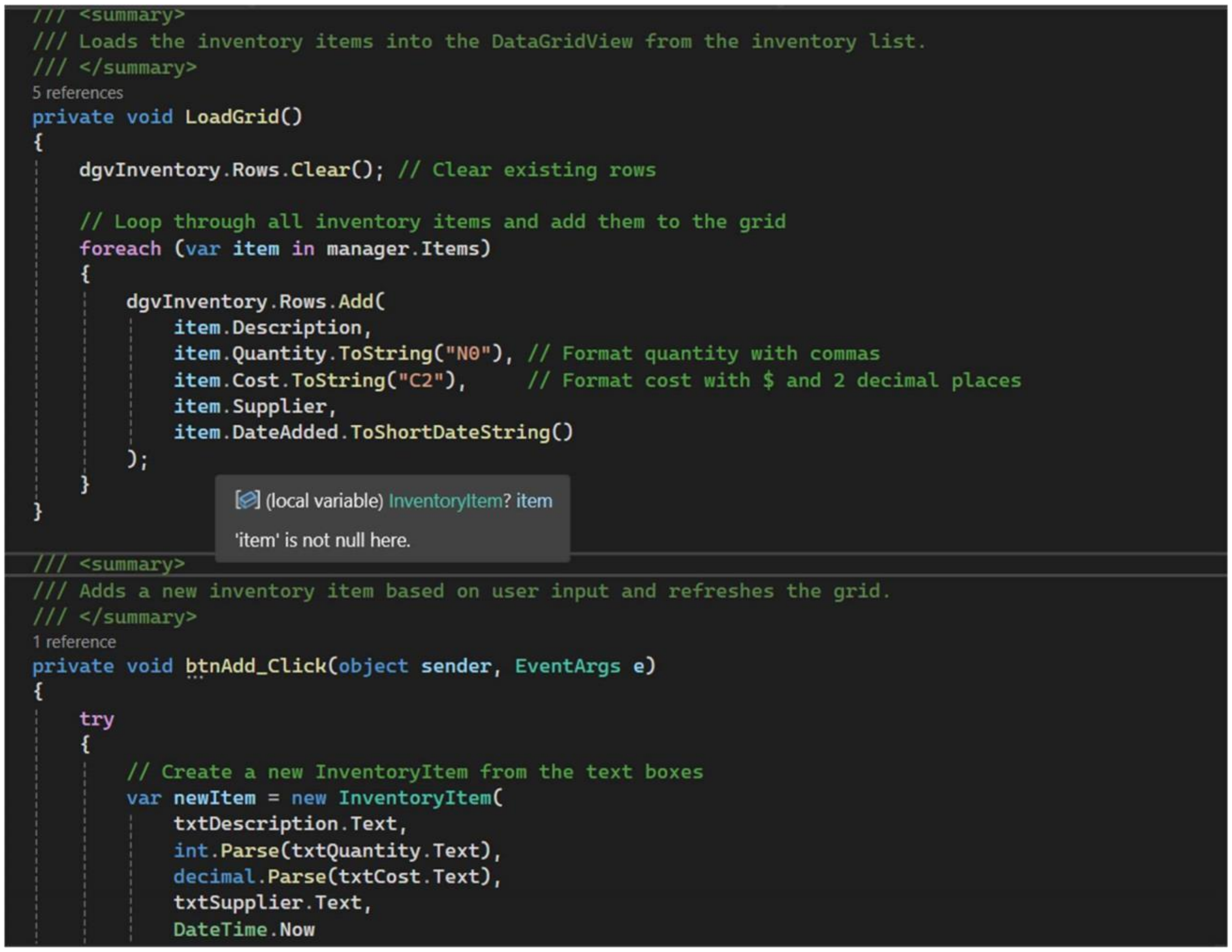
A er startup or clicking Add, the DataGridView gets populated with data from the list.

This visual output proves the inventory is being read from the text file and displayed correctly. When ac ons like Add, Delete, Inc, or Dec are triggered, the form immediately reflects these changes.

SCREENSHOT OF THE CODE BEHIND



```
// Name: Eric Gathinji
// Project: PresentationLayer
// File: FrmMain.cs
// Framework: .NET 8
// Citation: Adapted for CST-150 Inventory Management Application
// from Grand Canyon University Activity & Milestone guidelines.


using System;
using System.Linq;
using System.Windows.Forms;
using BusinessLayer;

namespace PresentationLayer
{
    3 references
    public partial class FrmMain : Form
    {
        // Declare an instance of the InventoryManager from the BusinessLayer
        private InventoryManager manager;

        1 reference
        public FrmMain()
        {
            InitializeComponent();

            // Define the full path to the inventory text file
            string filePath = Path.Combine(Application.StartupPath, "Data", "Inventory.txt");

            // Initialize InventoryManager with the file path and load inventory items
            manager = new InventoryManager(filePath);

            // Populate the DataGridView with the current inventory
            LoadGrid();
        }
```

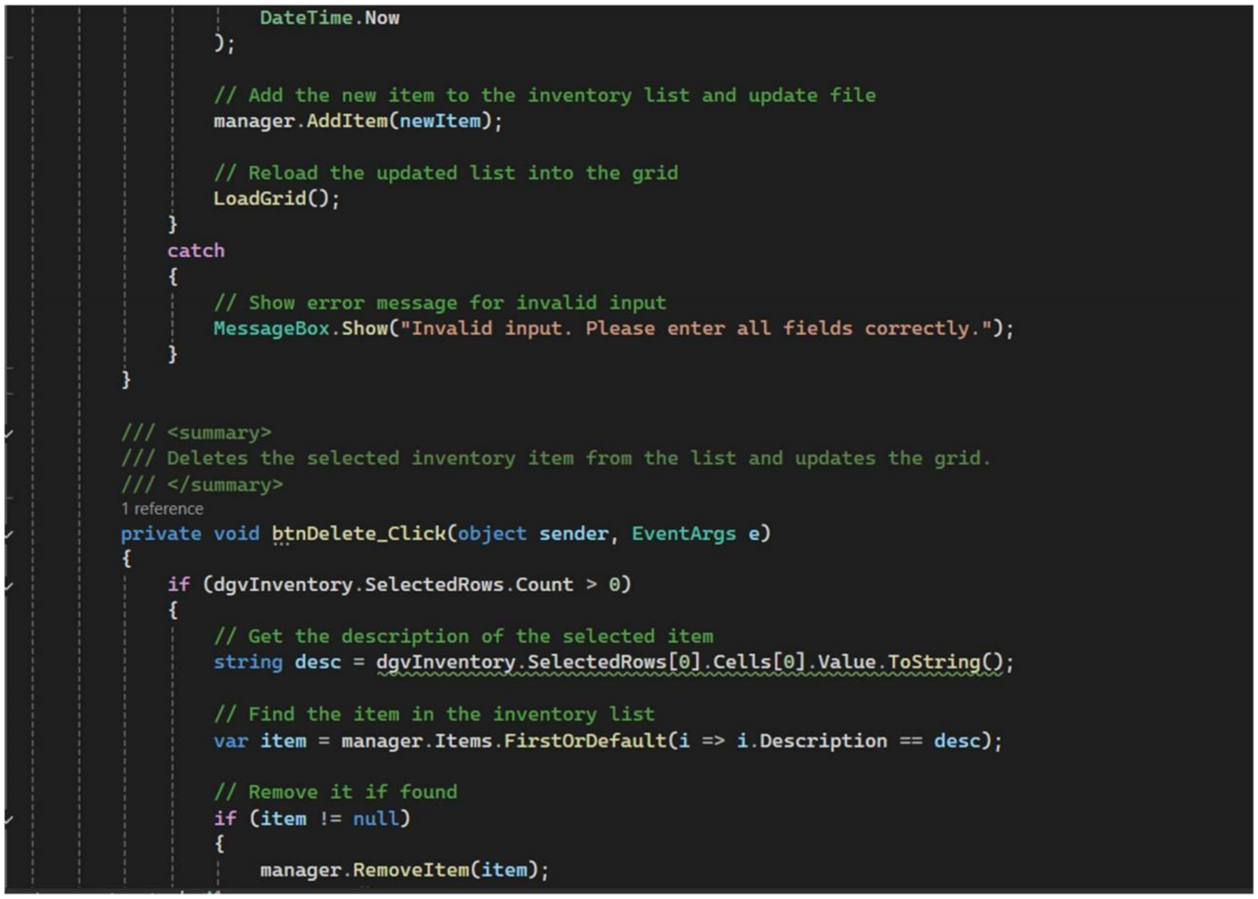


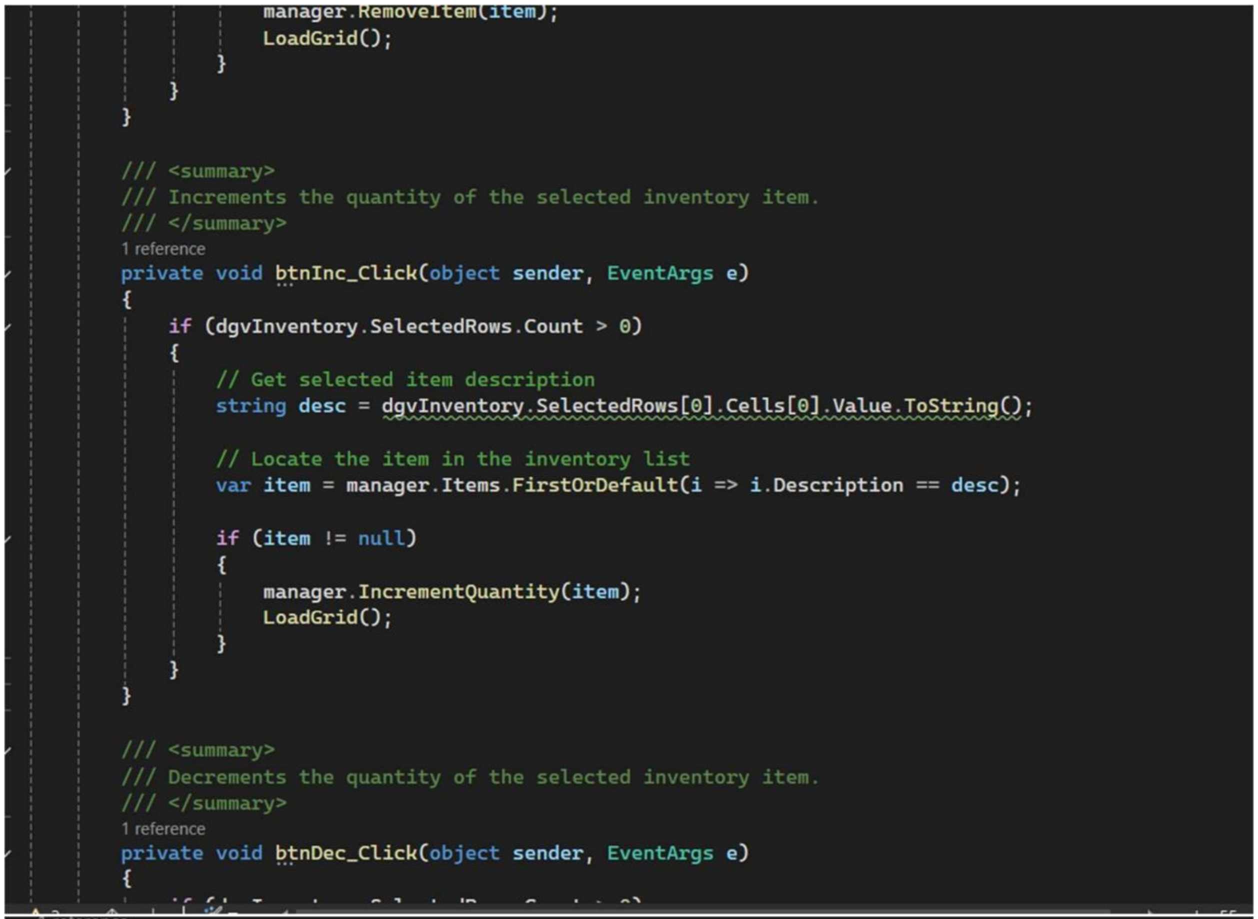
```
/// <summary>
/// Loads the inventory items into the DataGridView from the inventory list.
/// </summary>
5 references
private void LoadGrid()
{
    dgvInventory.Rows.Clear(); // Clear existing rows

    // Loop through all inventory items and add them to the grid
    foreach (var item in manager.Items)
    {
        dgvInventory.Rows.Add(
            item.Description,
            item.Quantity.ToString("N0"), // Format quantity with commas
            item.Cost.ToString("C2"),     // Format cost with $ and 2 decimal places
            item.Supplier,
            item.DateAdded.ToShortDateString()
        );
    }
}
```

(local variable) InventoryItem? item

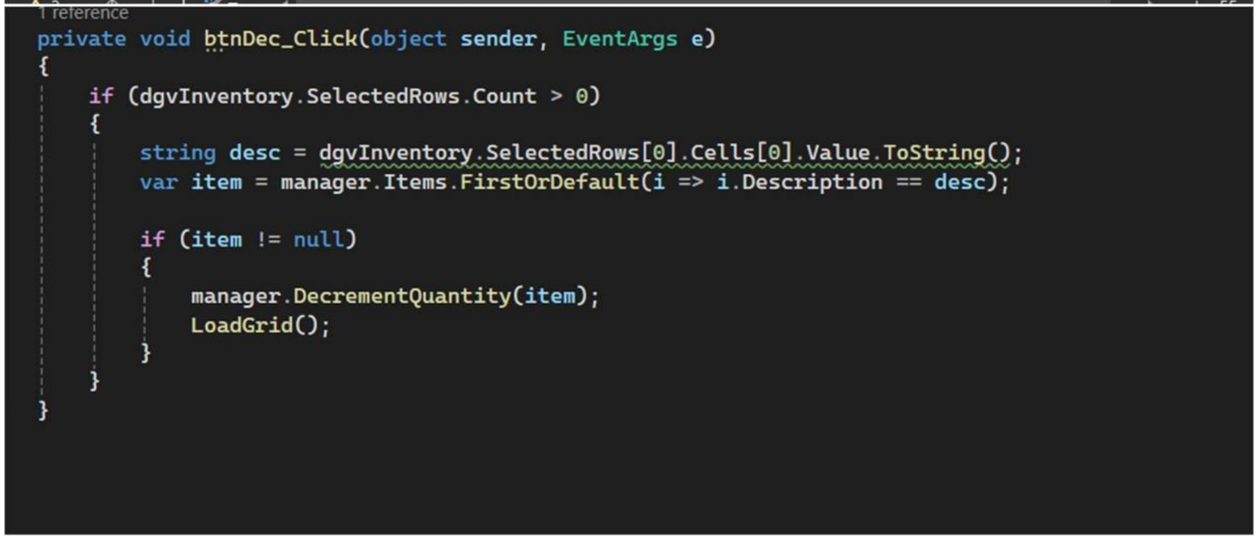
'item' is not null here.

```
/// <summary>
/// Adds a new inventory item based on user input and refreshes the grid.
/// </summary>
1 reference
private void btnAdd_Click(object sender, EventArgs e)
{
    try
    {
        // Create a new InventoryItem from the text boxes
        var newItem = new InventoryItem(
            txtDescription.Text,
            int.Parse(txtQuantity.Text),
            decimal.Parse(txtCost.Text),
            txtSupplier.Text,
            DateTime.Now
```


```
                DateTime.Now
            );

            // Add the new item to the inventory list and update file
            manager.AddItem(newItem);

            // Reload the updated list into the grid
            LoadGrid();
        }
        catch
        {
            // Show error message for invalid input
            MessageBox.Show("Invalid input. Please enter all fields correctly.");
        }
    }

    /// <summary>
    /// Deletes the selected inventory item from the list and updates the grid.
    /// </summary>
    1 reference
    private void btnDelete_Click(object sender, EventArgs e)
    {
        if (dgvInventory.SelectedRows.Count > 0)
        {
            // Get the description of the selected item
            string desc = dgvInventory.SelectedRows[0].Cells[0].Value.ToString();

            // Find the item in the inventory list
            var item = manager.Items.FirstOrDefault(i => i.Description == desc);

            // Remove it if found
            if (item != null)
            {
                manager.RemoveItem(item);
```



```
                    manager.RemoveItem(item);
                    LoadGrid();
                }
            }
        }

        /// <summary>
        /// Increments the quantity of the selected inventory item.
        /// </summary>
        1 reference
        private void btnInc_Click(object sender, EventArgs e)
        {
            if (dgvInventory.SelectedRows.Count > 0)
            {
                // Get selected item description
                string desc = dgvInventory.SelectedRows[0].Cells[0].Value.ToString();

                // Locate the item in the inventory list
                var item = manager.Items.FirstOrDefault(i => i.Description == desc);

                if (item != null)
                {
                    manager.IncrementQuantity(item);
                    LoadGrid();
                }
            }
        }

        /// <summary>
        /// Decrements the quantity of the selected inventory item.
        /// </summary>
        1 reference
        private void btnDec_Click(object sender, EventArgs e)
        {
```



```
1 reference
private void btnDec_Click(object sender, EventArgs e)
{
    if (dgvInventory.SelectedRows.Count > 0)
    {
        string desc = dgvInventory.SelectedRows[0].Cells[0].Value.ToString();
        var item = manager.Items.FirstOrDefault(i => i.Description == desc);

        if (item != null)
        {
            manager.DecrementQuantity(item);
            LoadGrid();
        }
    }
}
```

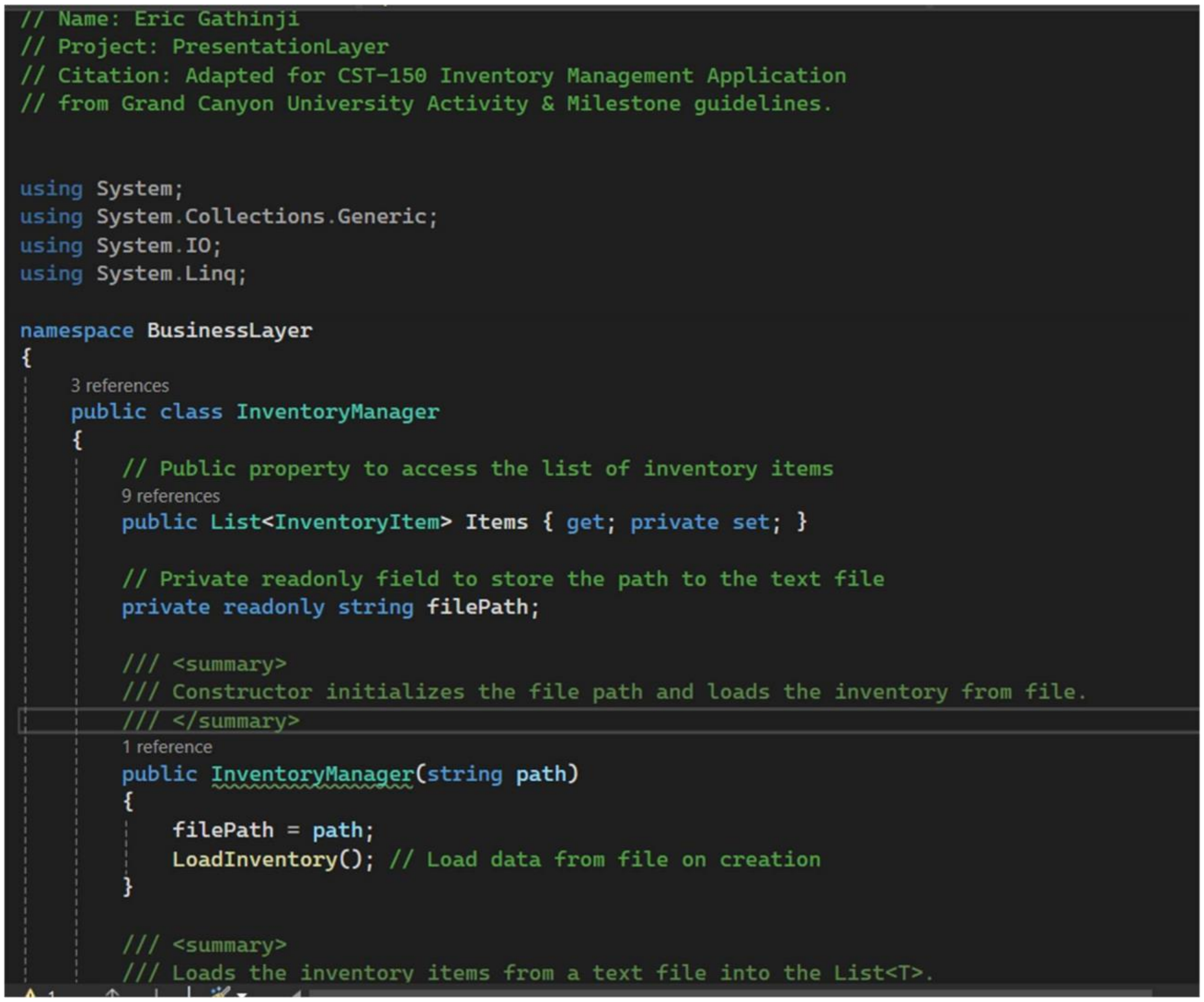
Figure 7: The code behind FrmMain.cs handles user interactions, including adding, deleting, and modifying inventory items.

EXPLANATION;

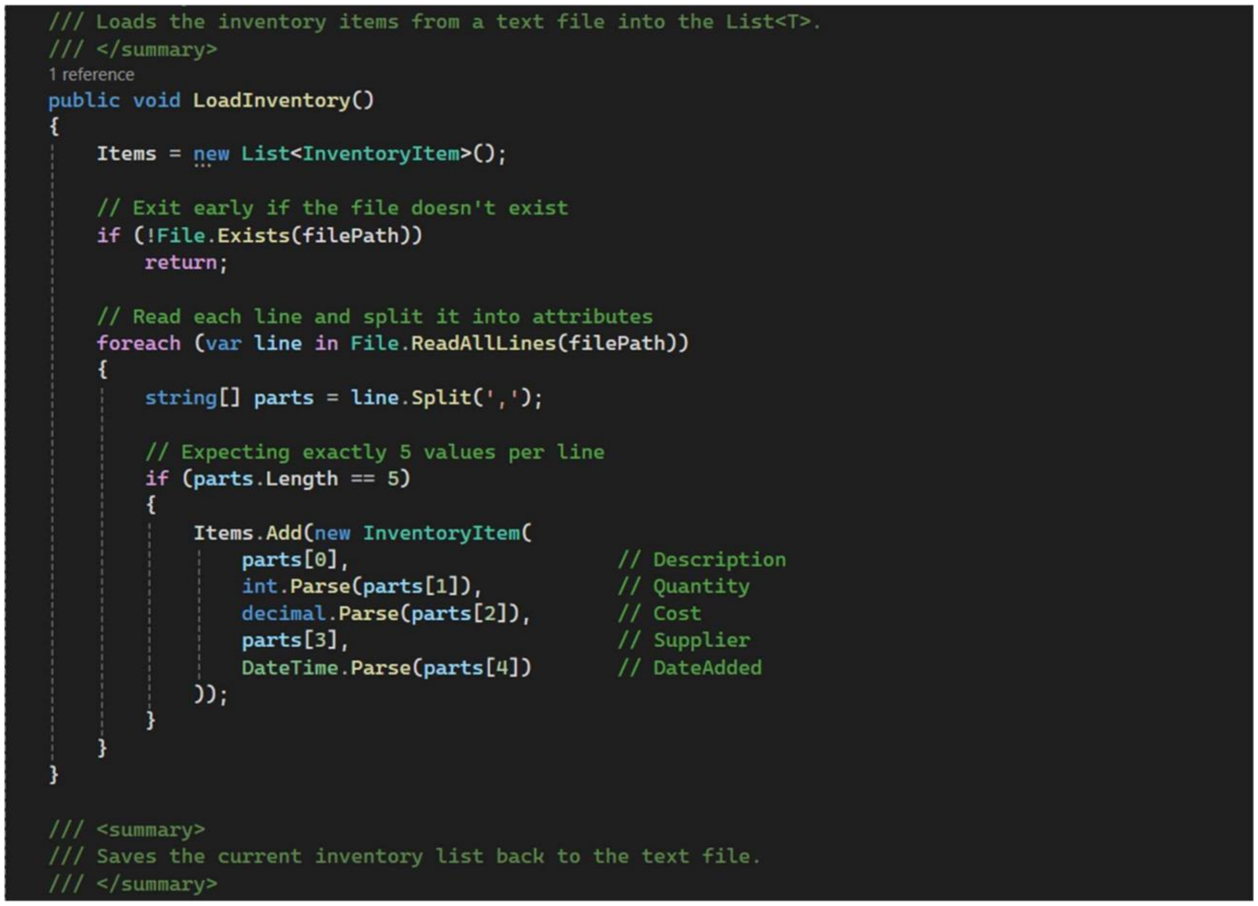
This file contains event handlers (btnAdd_Click, btnDelete_Click, btnInc_Click, btnDec_Click) that interact with the business logic layer via an instance of InventoryManager.

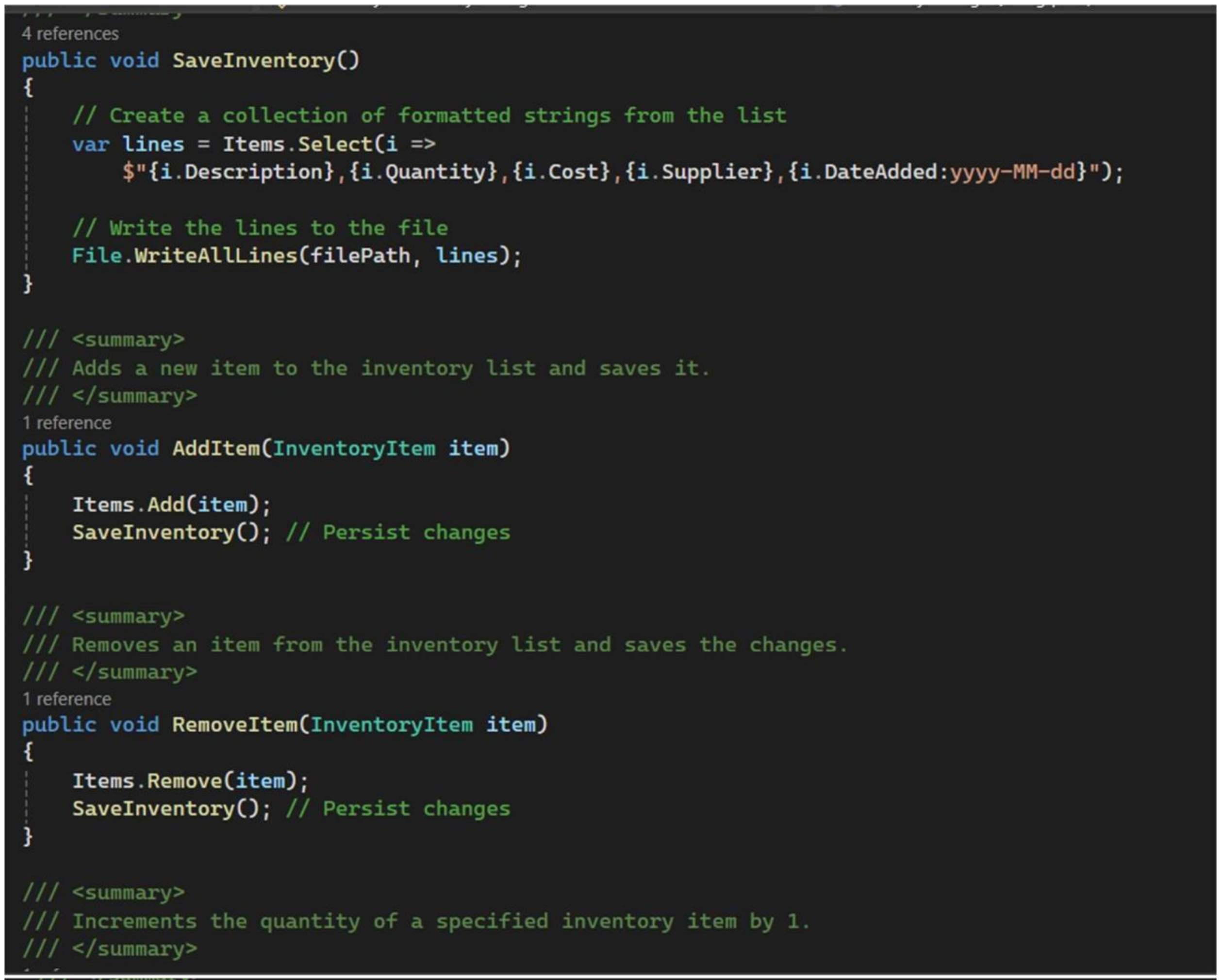
It maintains a synced state between UI and inventory file using the List<T> structure.

Screenshot of All Classes Focusing on List Collection

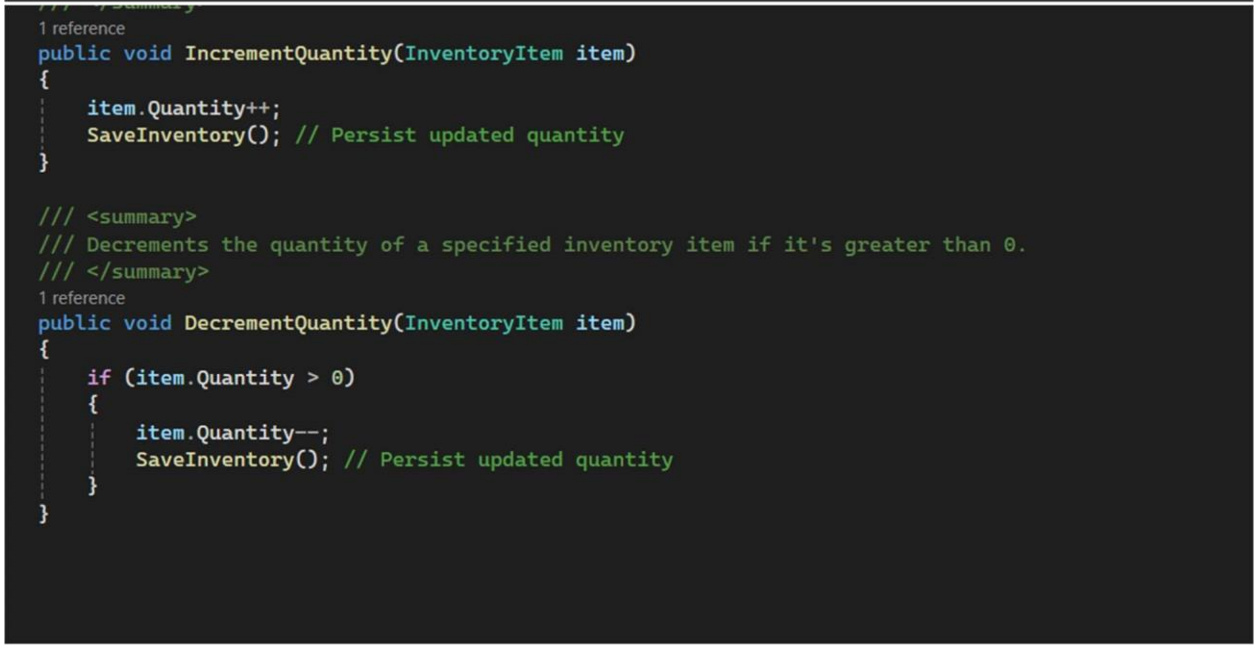


```
// Name: Eric Gathinji
// Project: PresentationLayer
// Citation: Adapted for CST-150 Inventory Management Application
// from Grand Canyon University Activity & Milestone guidelines.


using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;

namespace BusinessLayer
{
    3 references
    public class InventoryManager
    {
        // Public property to access the list of inventory items
        9 references
        public List<InventoryItem> Items { get; private set; }

        // Private readonly field to store the path to the text file
        private readonly string filePath;

        /// <summary>
        /// Constructor initializes the file path and loads the inventory from file.
        /// </summary>
        1 reference
        public InventoryManager(string path)
        {
            filePath = path;
            LoadInventory(); // Load data from file on creation
        }

        /// <summary>
        /// Loads the inventory items from a text file into the List<T>.
```


```
/// Loads the inventory items from a text file into the List<T>.
/// </summary>
1 reference
public void LoadInventory()
{
    Items = new List<InventoryItem>();

    // Exit early if the file doesn't exist
    if (!File.Exists(filePath))
        return;

    // Read each line and split it into attributes
    foreach (var line in File.ReadAllLines(filePath))
    {
        string[] parts = line.Split(',');

        // Expecting exactly 5 values per line
        if (parts.Length == 5)
        {
            Items.Add(new InventoryItem(
                parts[0],                       // Description
                int.Parse(parts[1]),            // Quantity
                decimal.Parse(parts[2]),        // Cost
                parts[3],                       // Supplier
                DateTime.Parse(parts[4])        // DateAdded
            ));
        }
    }
}

/// <summary>
/// Saves the current inventory list back to the text file.
/// </summary>
```



```
4 references
public void SaveInventory()
{
    // Create a collection of formatted strings from the list
    var lines = Items.Select(i =>
        $"{i.Description},{i.Quantity},{i.Cost},{i.Supplier},{i.DateAdded:yyyy-MM-dd}");

    // Write the lines to the file
    File.WriteAllLines(filePath, lines);
}

/// <summary>
/// Adds a new item to the inventory list and saves it.
/// </summary>
1 reference
public void AddItem(InventoryItem item)
{
    Items.Add(item);
    SaveInventory(); // Persist changes
}

/// <summary>
/// Removes an item from the inventory list and saves the changes.
/// </summary>
1 reference
public void RemoveItem(InventoryItem item)
{
    Items.Remove(item);
    SaveInventory(); // Persist changes
}

/// <summary>
/// Increments the quantity of a specified inventory item by 1.
/// </summary>
```



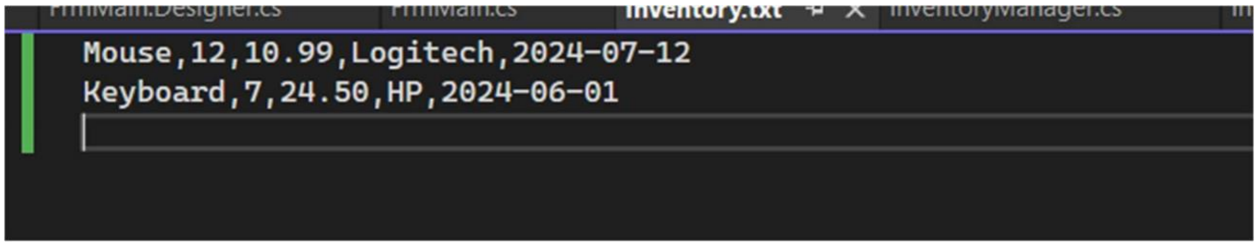
```
1 reference
public void IncrementQuantity(InventoryItem item)
{
    item.Quantity++;
    SaveInventory(); // Persist updated quantity
}

/// <summary>
/// Decrements the quantity of a specified inventory item if it's greater than 0.
/// </summary>
1 reference
public void DecrementQuantity(InventoryItem item)
{
    if (item.Quantity > 0)
    {
        item.Quantity--;
        SaveInventory(); // Persist updated quantity
    }
}
```

Figure 8:The InventoryManager.cs class reads, writes, and modifies the master inventory using a List<InventoryItem>.

EXPLANATION;

The class defines methods like LoadInventory(), SaveInventory(), AddItem(), RemoveItem(), IncrementQuan ty(), and DecrementQuan ty(). Internally, all items are stored in a List<InventoryItem>, which ensures dynamic resizing and easy manipula on

Screenshot of the Original Text File



```
Mouse,12,10.99,Logitech,2024-07-12
Keyboard,7,24.50,HP,2024-06-01
```

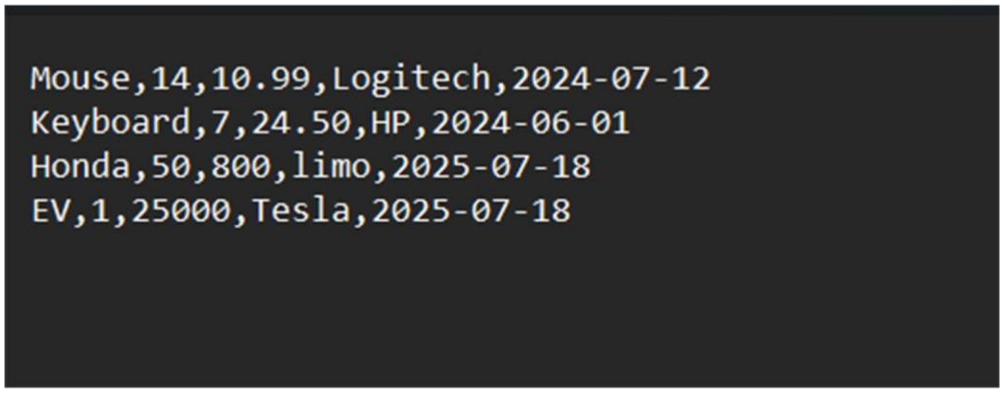
Figure 9: The original Inventory.txt contains comma-separated records of inventory items.

EXPLANATION;

Each line of the text file represents an inventory item with attributes: Description, Quan ty, Cost, Supplier, and DateAdded.

This file is read at application startup and serves as the base state for inventory management.

Screenshot of the Text File A er Inventory Is Updated



```
Mouse,14,10.99,Logitech,2024-07-12
Keyboard,7,24.50,HP,2024-06-01
Honda,50,800,limo,2025-07-18
EV,1,25000,Tesla,2025-07-18
```

Figure 10: The updated Inventory.txt reflects changes made through the application—newly added, incremented, or deleted items.

EXPLANATION;

Once users interact with the form, the updated inventory is written back to the text file using SaveInventory().

The file structure remains consistent, with one item per line, ensuring data can always be reloaded correctly at next launch.

ADD ONS

Milestone 6 research

Follow-up questions

<u>Milestone 6 Follow-up questions</u>
Platform used to play the game- Windows
Name of the Game: Portal 2
Category: Puzzle
Gameplay description: In Portal 2, players use a portal gun to build connected portals on flat objects to solve physics-based puzzles. The action gets increasingly more complicated as the player moves through the test chambers and encounters additional features, including propulsion gels, laser redirection, and gravity fields.

<u>How can it be improved?</u>
By providing adjustable difficulty settings and more thorough visual cues for new gamers who are not experienced with 3D puzzle navigation, Portal 2 could increase accessibility.

<u>Describe the color scheme</u>
The portals in Portal 2 are represented with a futuristic, industrial color scheme that consists of chilly grays, whites, and warm oranges and blues. To preserve mood and narrative tone, lighting and color change as the story progresses, from pristine labs to dilapidated testing facilities

<u>How could it be improved?</u>
In order to guarantee that puzzle components like laser trajectories and portal colors are discernible to all players, the game might improve contrast in specific low-light conditions and provide a color-blind mode.

<u>The Key takeaway for the milestone project</u>

Portal 2 shows how an engaging interactive experience may be created by fusing powerful user feedback, simple controls, and increasingly complex problems. My inventory management app can become more engaging and user-friendly by implementing the following principles: feedback-driven design and simple user interface.

Bug report: None

<u>Follow-up questions</u>
What was challenging? Maintaining file data and
UI synchronization following each user action.

What did you learn? How to use layered architecture and use List<T> with DataGridView.

How would you improve the project? Include search capabilities, error management, and input validation.

How can you use what you learned on the job? To create dependable data management and data-driven desktop applications with a clean user interface.

ADD ON
Naming conventions- using the standard naming (like camelCase) makes the code easier to read.
Properties – using PascalCase. E.g Quantity Code structure-
using comments for clarification.
Consistency- The formatting is consistent throughout my project.

Computer specs- The OS: Windows 11

RAM: 8GB

Processor: Intel Core i5

Tutor discovery 5
I read the instructions given by my instructor, Mark Smithers.
<u>Weekly Activity</u>
Start Tuesday, 15th July 2025 End
9:00 am 4:00 pm Milestone 6
Start Wednesday 16th July 2025 End
11:00 am 6:00 pm Milestone 6
Start Thursday 17th July, 2025 End
9:00 am 2:00 pm Milestone 6