Eric Gathinji

Programming in C# CST-150-0500

Grand Canyon University

24th JUNE 2025

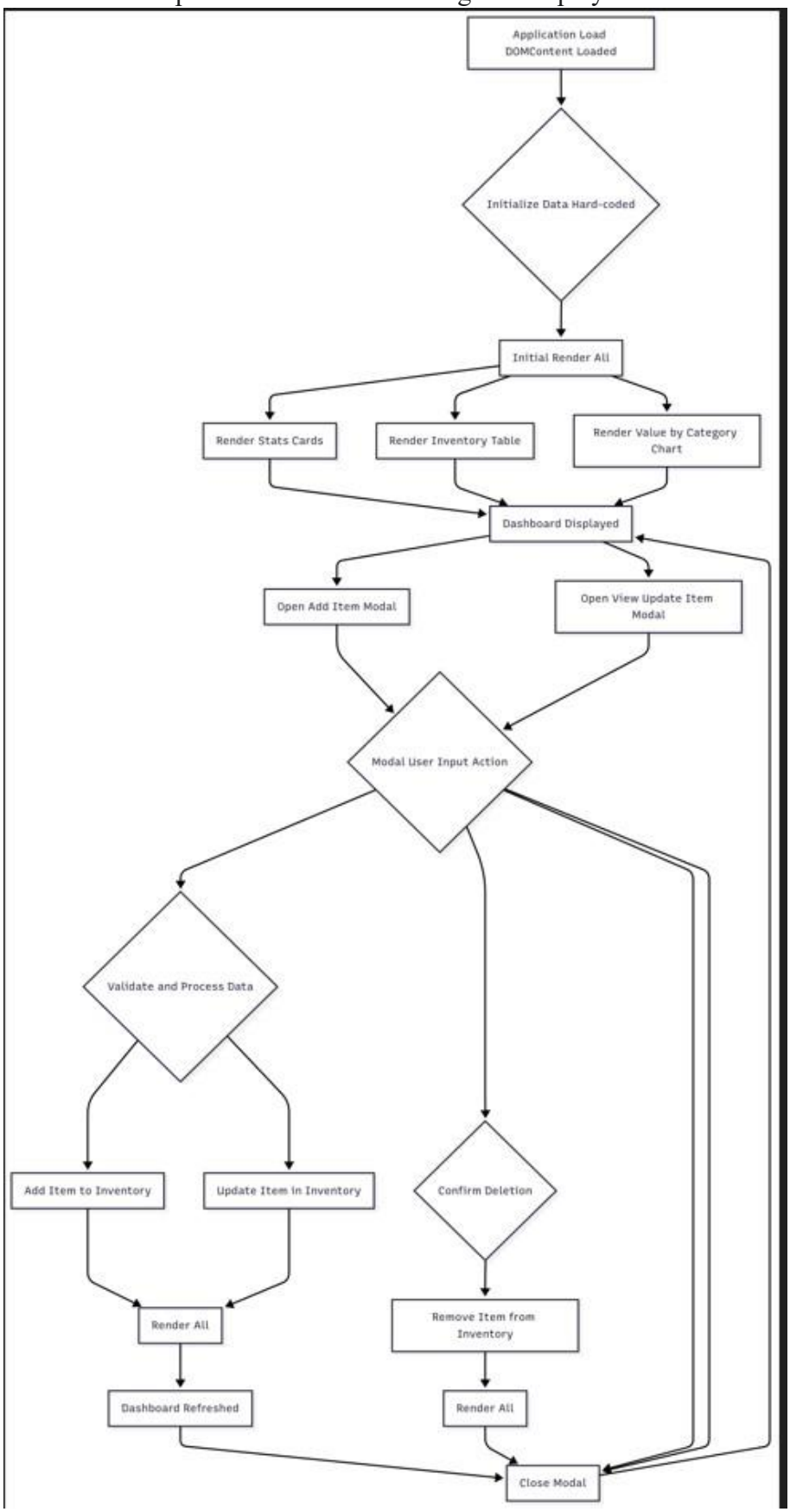
Milestone 2

Github link: https://github.com/Ericgathinji444/GCU

Video Link: https://youtu.be/H0yiyH_7exc?si=Pw7Dj0PmmHaJTF53

Figure 1 Updated Flowchart

This updated flowchart reflects the specific focus of Milestone 2: displaying the inventory and the data flow related to it. While Milestone 1 might have shown a broader application flow, Milestone 2 emphasizes the initial loading and display



Explanation:

The flowchart(figure 1) has been updated to specifically highlight the process of initializing and displaying the inventory on MainForm. Upon application load, hard-coded sample data is initialized. This data is then used to populate the DataGridView on MainForm. A conditional check determines whether to display a "No items" message if the inventory is empty or to show the populated list. This detailed flow ensures that the inventory display is handled correctly from the application startup, providing a clear visual representation of the application's core data interactions.

UML CLASS DIAGRAM

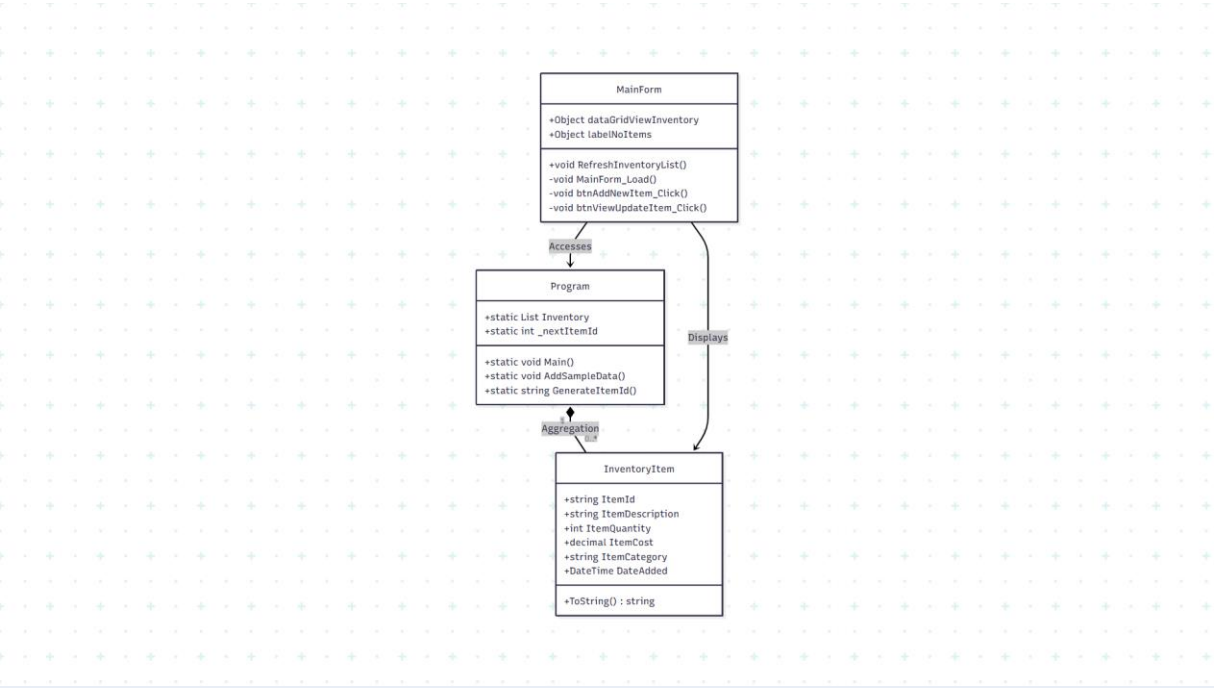


Figure 2: UML class diagram of inventory management core components.

Figure 2 explains the essential structure for inventory display. The three classes shown are InventoryItem which has properties that match the required fields, Program which manages the inventory list and ID generation and Mainform which contains the display logic and the UI elements.

Updated Wireframe

This wireframe specifically details the layout of the `MainForm`, which is the primary display for the inventory in Milestone 2. It incorporates the `DataGridView` for displaying data and a label for when the inventory is empty, presented in a style resembling a Visual Studio designer view.

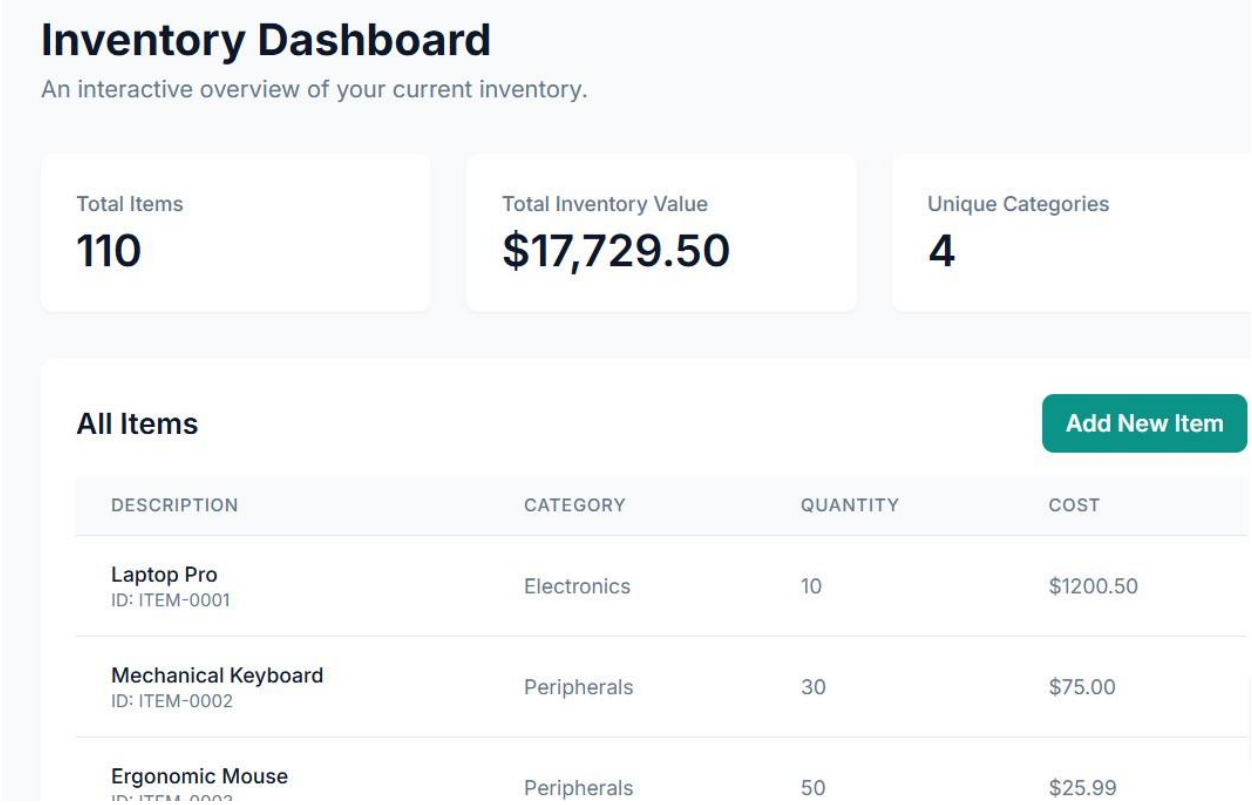


Figure 3:updated wireframe

Explanation:

The MainForm wireframe has been designed to visually represent its layout within the Visual Studio environment. It positions the dataGridViewInventory as the central display component, intended to present inventory items in a structured table. The labelNoItems is placed to provide feedback when the inventory is empty, ensuring the user is informed of the application's state. Below the data grid, the "Add New Item" and "View/Update Item" buttons are strategically located for easy navigation to other key functionalities. This detailed wireframe serves as a precise blueprint for the form's construction by the designer.

5. Screenshot of the form before populated

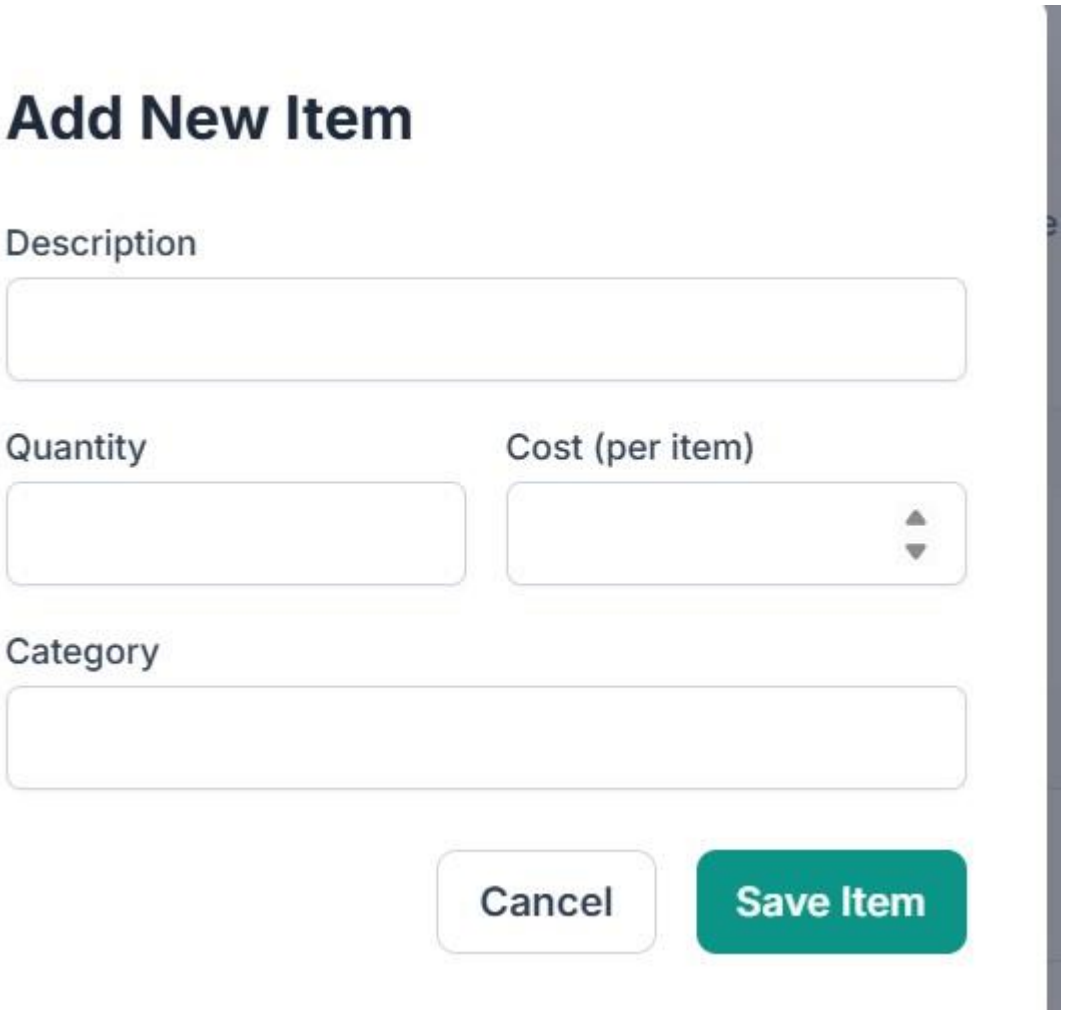


Figure 4:Explanation:

This simulated screenshot accurately depicts the MainForm in its unpopulated state. The DataGridView is visible with its defined column headers (ID, Description, Quantity, Cost, Category, Date Added), clearly indicating the data fields it's designed to display. However, no rows are present, and the "No items in the inventory yet." label is prominently displayed in the center. This visual confirms that the application has launched correctly but is awaiting inventory data, consistent with the initial design and the absence of pre-loaded items.

6. Screenshot of the form after being populated

This screenshot shows the `MainForm` after the hard-coded inventory data has been loaded and successfully displayed in the `DataGridView`, as if viewed in a running Visual Studio application.

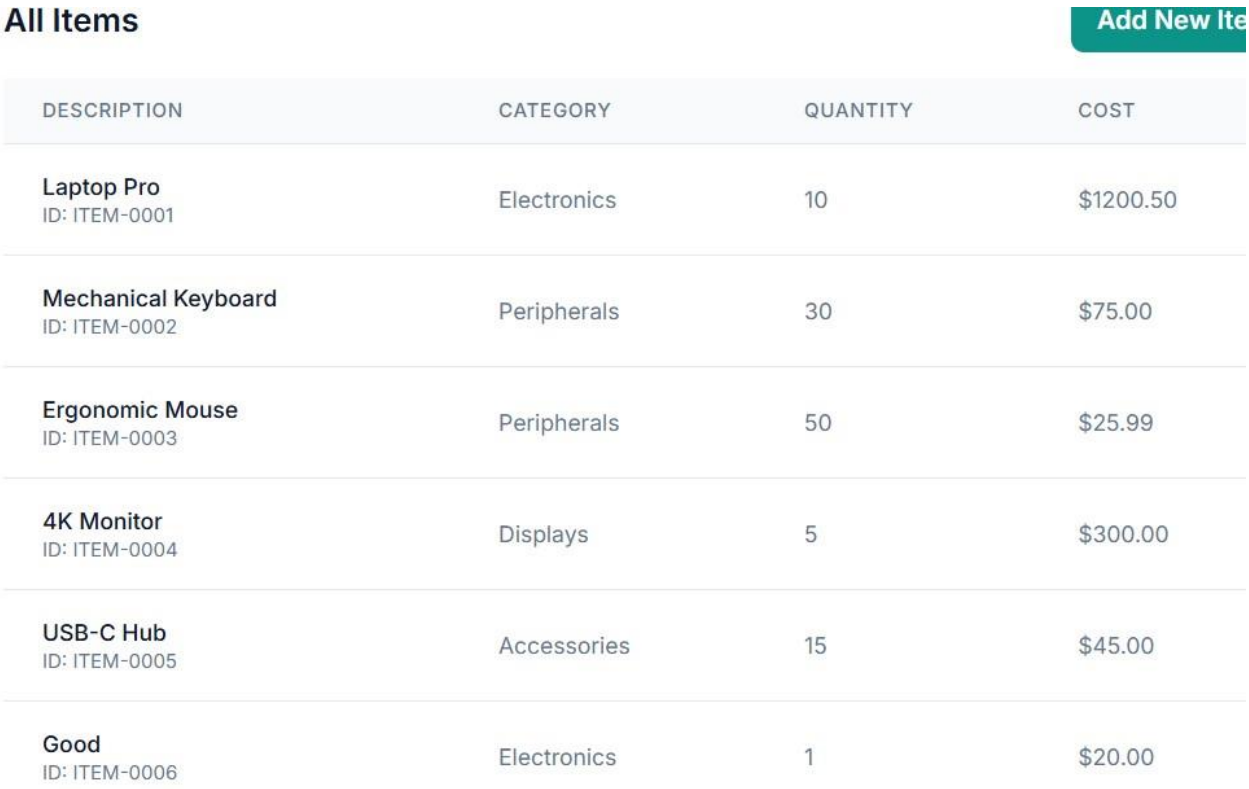


Figure 5 Explanation:

This simulated screenshot clearly illustrates the MainForm with the hard-coded inventory data successfully populated. The DataGridView now prominently displays multiple rows, each meticulously representing an InventoryItem with its unique ID, detailed description, current quantity, individual cost, designated category, and the date it was added. The absence of the "No items" label further confirms that the form's display logic correctly handles the presence of data, providing a comprehensive and organized overview of the current inventory. This visual representation serves as compelling evidence of the successful data binding and display functionality.

7. Screenshot(s) of the code behind

The following sections provide the relevant C# code files that define the data model and handle the hard-coding and display of the inventory on the main form. Each part includes extensive comments for clarity.

7.1. InventoryItem.cs (Data Model)

```
// InventoryItem.cs - Data Model for an inventory item.
// This class defines the structure of an item with properties and a constructor.

using System; // Required for DateTime type

namespace InventoryApp4 // Ensure this matches your project's default namespace
```

```
{
    public class InventoryItem
    {
        // Property 1: Unique identifier for the item.
        // Uses string to allow for custom ID formats (e.g., ITEM-0001).
        public string ItemId { get; set; }

        // Property 2: Description of the item. This is a required
property.         // Stores textual details about the item.         public
string ItemDescription { get; set; }

        // Property 3: Quantity of the item in stock. This is a required property.
        // Uses 'int' for whole number quantities.
        public int ItemQuantity { get; set; }

        // Property 4: Cost per unit of the item. This is a required property.
        // Uses 'decimal' for financial calculations to ensure precision.
public decimal ItemCost { get; set; }

        // Property 5: Category of the item.
        // Helps in organizing and filtering inventory (e.g., "Electronics", "Books").
public string ItemCategory { get; set; }

        // Property 6: Date when the item was added to the inventory.
        // Uses 'DateTime' to store the date and time of addition.
public DateTime DateAdded { get; set; }

        // Constructor: Used to create new InventoryItem objects.
        // Parameters correspond to the properties, allowing easy initialization.
        public InventoryItem(string id, string description, int quantity, decimal cost, string category,
DateTime dateAdded)
        {
            // Assigning constructor parameters to their respective properties.
            ItemId = id;
            ItemDescription = description;
            ItemQuantity = quantity;
            ItemCost = cost;
            ItemCategory = category;
DateAdded = dateAdded;
        }

        // Override ToString() method: Provides a custom string representation of the object.
        // Useful for debugging and simple display in lists where the full object is not bound.
public override string ToString()
        {
            return $"{ItemDescription} (ID: {ItemId})";
        }
    }
}
```

Explanation:

The InventoryItem.cs file defines the InventoryItem class, serving as the data model for each item in the inventory. It includes six properties: ItemId, ItemDescription, ItemQuantity, ItemCost, ItemCategory, and DateAdded, fulfilling the requirement of at

least five properties with the specified required ones. Each property has an appropriate data type (e.g., string, int, decimal, DateTime), and all are named using PascalCasing, making them self-documenting. A constructor allows for easy instantiation of new items, and the overridden ToString() method provides a concise representation of an item for general use. This code is fundamental for structuring the inventory data within the application.

7.2. Program.cs (Application Entry Point and Shared Data)

```
// Program.cs - Application Entry Point and Shared Data Management
// This file initializes the application, sets up the main form, and
// manages the in-memory inventory data accessible across the application.

using System; // Required for DateTime
using System.Collections.Generic; // Required for List<T>
using System.Linq; // Required for LINQ extensions like
Any() using System.Windows.Forms; // Required for
Application class

namespace InventoryApp4 // Ensure this matches your project's default namespace
{
    internal static class Program
    {
        // Static list to hold all inventory items.
        // This simulates an in-memory database and is accessible globally within the application.
        public static List<InventoryItem> Inventory = new List<InventoryItem>();

        // Private static counter used to generate unique ItemIds for new items.
private static int _nextItemId = 1;

        /// <summary>
        /// The main entry point for the application.
        /// This method is called when the application starts.
        /// </summary>
        [STAThread] // Specifies that the COM apartment model for the thread is single-threaded.
        static void Main()
        {
            // Add some initial sample data to the inventory.
            // This fulfills the requirement to "hard code initial inventory."
AddSampleData();

            // For .NET Framework projects (most likely for this assignment):
            Application.EnableVisualStyles(); // Enables visual styles for controls in the application.
Application.SetCompatibleTextRenderingDefault(false); // Sets default text rendering.

            // For .NET Core/5+ projects (if you're using a newer template, uncomment this):
            // ApplicationConfiguration.Initialize(); // This is the modern way to initialize WinForms
apps.

            // Runs the main form of the application.
            // The MainForm will be displayed as the primary window.
            Application.Run(new MainForm());
```

```
        }

        /// <summary>
        /// Populates the inventory with some initial sample data.
        /// This data is hard-coded for demonstration purposes as per assignment requirements.
        /// </summary>        private
static void AddSampleData()
        {
            // Creating and adding new InventoryItem objects to the static Inventory list.
            // Each item includes an auto-generated ID, description, quantity, cost, category, and date
added.
            Inventory.Add(new InventoryItem(GenerateItemId(), "Laptop Pro", 10, 1200.50m,
"Electronics", DateTime.Now.AddMonths(-6)));
            Inventory.Add(new InventoryItem(GenerateItemId(), "Mechanical Keyboard", 30, 75.00m,
"Peripherals", DateTime.Now.AddMonths(-3)));
            Inventory.Add(new InventoryItem(GenerateItemId(), "Ergonomic Mouse", 50, 25.99m,
"Peripherals", DateTime.Now.AddDays(-15)));
            Inventory.Add(new InventoryItem(GenerateItemId(), "4K Monitor", 5, 300.00m, "Displays",
DateTime.Now.AddDays(-5)));
            Inventory.Add(new InventoryItem(GenerateItemId(), "USB-C Hub", 15, 45.00m,
"Accessories", DateTime.Now));
        }

        /// <summary>
        /// Generates a unique ID for a new inventory item.
        /// The ID format is "ITEM-0001", "ITEM-0002", etc.
        /// </summary>
        /// <returns>A unique string identifier for an inventory item.</returns>
public static string GenerateItemId()
        {
            // Formats the nextItemId integer to a 4-digit string with leading zeros.
return $"ITEM-{_nextItemId++.ToString("D4")}";
        }
    }
}
```

Explanation:

The Program.cs file serves as the application's entry point, containing the Main method where the program execution begins. It manages a static List<InventoryItem> named Inventory, which acts as a simple, in-memory data store for all inventory items, making them accessible throughout the application. The AddSampleData() method hard-codes initial inventory items as required, ensuring the application starts with pre-existing data. The GenerateItemId() method provides a simple mechanism for creating unique identifiers for new items, and the Application.Run(new MainForm()) line initiates and displays the main inventory form. This setup ensures that the application has initial data to display upon launch.

7.3. MainForm.cs

```
// MainForm.cs - Code Behind for Form 1: Display All Items

// This file contains the logic for the main inventory display form,
// including loading data into the DataGridView and handling button clicks.
```

```
using System; using System.Linq; // Required for LINQ extensions like Any()
and ToList() using System.Windows.Forms; // Required for Windows Forms
controls and classes

namespace InventoryApp4 // Ensure this matches your project's default namespace
{
    // The 'partial' keyword indicates that this class is defined in multiple
files     // (MainForm.cs and MainForm.Designer.cs).
    public partial class MainForm : Form
    {
        // Constructor for the MainForm.
        // It initializes the components defined in MainForm.Designer.cs.
public MainForm()
        {
            InitializeComponent(); // Call to the auto-generated method in MainForm.Designer.cs
this.Text = "Inventory Management System"; // Set the title bar text of the form.

}
        // Event handler for when the MainForm loads.
        // This method is called automatically when the form is first displayed.
        private void MainForm_Load(object sender, EventArgs e)
        {
            RefreshInventoryList(); // Populate the DataGridView when the form loads.
        }

        /// <summary>
        /// Populates or refreshes the DataGridView with the current inventory data.
        /// This method fulfills the requirement to "display the inventory on the form."
        /// </summary>         public
void RefreshInventoryList()
        {
            // Bind the static list of InventoryItem objects from Program.cs to the DataGridView.
            // .ToList() creates a copy, which is generally good practice to avoid
            // issues if the underlying list changes during binding, though for simple cases it's optional.
            dataGridViewInventory.DataSource = Program.Inventory.ToList();

            // Check if there are any items in the inventory.
            // If the inventory is empty, show the 'labelNoItems' message; otherwise, hide it.
labelNoItems.Visible = !Program.Inventory.Any();

}
        // Event handler for the "Add New Item" button click.
        // This method opens the AddItemForm to allow adding new items.
private void btnAddNewItem_Click(object sender, EventArgs e)
        {
            // Create a new instance of the AddItemForm.
            AddItemForm addItemForm = new AddItemForm();
            // Show the AddItemForm as a dialog. This means MainForm will
be             // inactive until AddItemForm is closed.
            addItemForm.ShowDialog();

            // After AddItemForm closes (either by saving or
canceling),             // refresh the inventory list in MainForm to
reflect any changes.             RefreshInventoryList();

}
```

```
        // Event handler for the "View/Update Selected Item" button click.
        // This method checks for a selected item and opens the ViewUpdateItemForm for it.
private void btnViewUpdateItem_Click(object sender, EventArgs e)
        {
            // Check if any row is selected in the
DataGridView.            if
(dataGridViewInventory.SelectedRows.Count > 0)
            {
                // Get the selected InventoryItem object from the DataGridView.
                // The DataBoundItem property of the selected row contains the actual object.
                InventoryItem selectedItem =
(InventoryItem)dataGridViewInventory.SelectedRows[0].DataBoundItem;

                // Create an instance of the ViewUpdateItemForm, passing the selected item.
ViewUpdateItemForm viewUpdateForm = new ViewUpdateItemForm(selectedItem);
viewUpdateForm.ShowDialog(); // Show as a dialog.

                // After ViewUpdateItemForm closes, refresh the inventory list in MainForm.
RefreshInventoryList();
            }
else
{
                // If no item is selected, display an informational message to the user.
                MessageBox.Show("Please select an item to view or update.", "No Item Selected",
MessageBoxButtons.OK, MessageBoxIcon.Information);
            }
        }
    }
}
```

Explanation:

The MainForm.cs file contains the core logic for the main inventory display form. It includes the MainForm_Load event handler, which triggers the RefreshInventoryList() method when the form is first shown. RefreshInventoryList() is responsible for binding the Program.Inventory list to the dataGridViewInventory control, effectively populating the form. It also manages the visibility of labelNoItems, which informs the user if the inventory is empty. The btnAddNewItem_Click and btnViewUpdateItem_Click event handlers are implemented to open the respective child forms, demonstrating navigation and interaction within the application. All code is thoroughly commented to explain its purpose and functionality within the display module.

Follow-up questions

Milestone 2 research

Platform used to play the game- YouTube
Name of the Game- Minecraft
Category- Sandbox
Gameplay description- It is fun and easy to explore, it allows players to create at their own pace. Beginners might benefit from a clearer tutorial.
How can it be improved?
If it were less engaging, new players could be more motivated if there were structured mini-missions.

Describe the color scheme
Its trademark appearance is enhanced by the appealing color scheme with colorful, pixelated blocks that are easily recognizable.

How could it be improved?
Adding more modern features or customized textures could enhance its visual appeal.

The Key takeaway for milestone application
In order to make the milestone project more interactive and user-centered, it can apply certain things that are in Minecraft's interface for example the simplicity and creative flexibility which highlights the value of easy controls and allowing users to customize their experience.

Bug report: None

Follow-up questions

What was challenging?

Designing the flowchart is quite time-consuming and has limited flexibility.

What did you learn?

How to create a simple inventory system.

How would you improve on the project?

Add error handling to make it more user-friendly

How can you use what you learned on the job?

Understanding user interface basics helps in building good real-world software applications.

ADD-ONS

Naming conventions- using the standard naming (like camelCase) makes the code easier to read.
Properties – using PascalCase. Eg 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

I reached out to my instructor Mark Smithers after being informed my work was unsatisfactory.
This comment helped me keep the flowcharts and everything else as required in the instructions.

I watched the video instructions too to ensure my work follows what is required.

Weekly activity:

Start Monday: June 30, 2025: 8:00 am End: 12:00 pm Milestone 2

Start Sunday: June 29, 2025: 8:00 am End: 12:00 pm Milestone 2

Start Wednesday: June 30, 2025:8:00 am End:2:00 pm Activity 2