**SWE40006 – Deployment Activity 1**

**Student Details**

* **Full name:** Dang Khoa Le
* **Student ID:** 103844421
* **Due date:** 24/08/2025
* **Submission date:** 24/08/2025

**Unit Details**

* **Unit code & name:** SWE40006 – Software Deployment and Evolution
* **Semester:** Semester 2, Year 2025

**Declaration of Task Levels Attempted**

* **Task 1.1 (Pass)** – Visual Studio & WiX installation walkthrough, Simple C# Hello World Console app, with WiX Toolset and Windows Application Packaging (WAP).
* **Task 1.2 (Credit)** – Create a simple Windows Forms App (WFA) Calculator app, built with WiX and WAP.
* **Task 1.3 (Distinction)** – Create multiple-DLL WFA project for finance ledging (FinanceManagerApp), built with WiX and WAP.
* **Task 1.4 (High Distinction)** – Packaging and fully explain how to deploy FinanceManagerApp from task 1.3 and publicly published to the Microsoft Store.

**1.1 Visual Studio & WiX Installation – Simple App MSI (Pass Task)**

**1.1.1 Prepare Visual Studio (UWP workload & .NET SDK)**

The walkthrough’s Part 1 requires Visual Studio with specific workloads/components, notably Universal Windows Platform (UWP) development and .NET Framework 4.6.2 SDK, as shown in Wix Installation walkthrough so that Part 4 (MSIX packaging) can be completed. Steps:

1. Open the Visual Studio Installer → Modify.

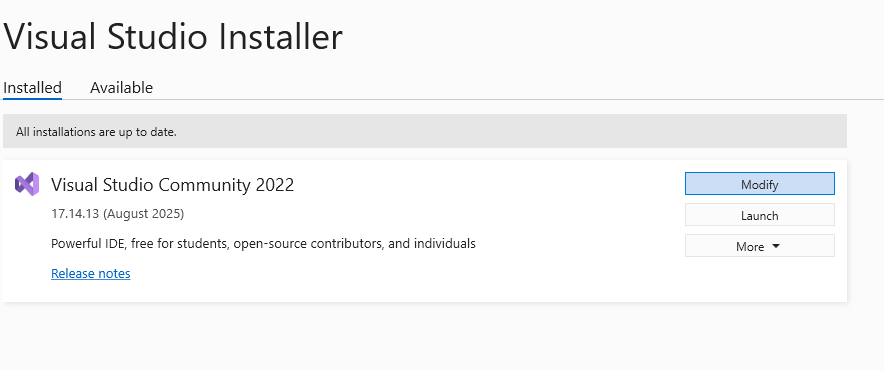


Figure. Visual Studio Installer – Modify option

1. While the WiX Walkthrough guide us to choose Universal Windows Platform development, this has been deprecated for Visual Studio 2022 (my version), therefore, we select development packages (.NET and C++) instead.

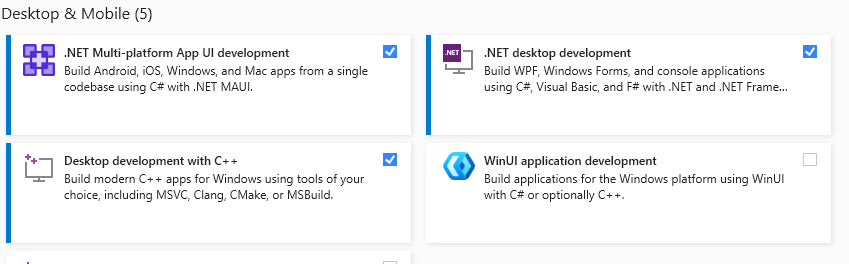


Figure. Visual Studio Installer – Package option

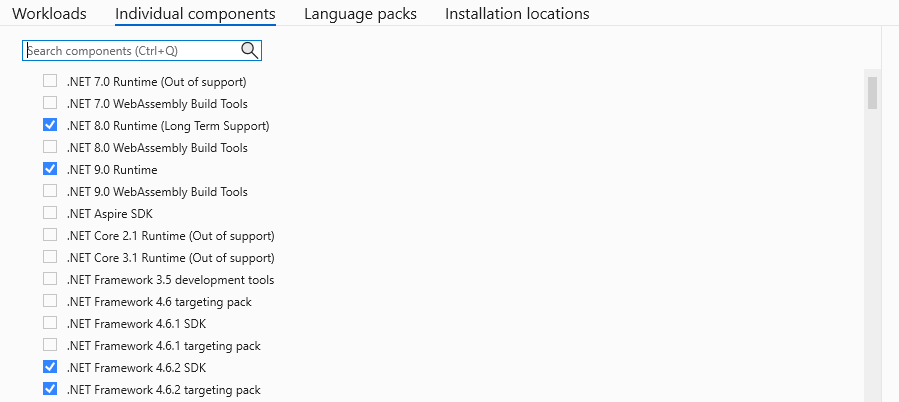
1. Switch to Individual components tab and ensure .NET Framework 4.6.2 SDK is selected. However, from experiment with WiX app packaging, my system (local device) failed when compiling the app (not on Visual Studio console build error but a terminal error when trying to run the EXE file, that I couldn’t take screenshot), since it no longer support version 4.6.2. Therefore, I decided to upgrade/retarget the project to .NET8.0 (as also ticked in the screenshot below).

Figure. Visual Studio Installer – Individual components

1. Click Modify (bottom-right) to apply changes and let the installer download/install components.

**1.1.2 Install the WiX Toolset (Core) & Add WiX Extension to Visual Studio**

The walkthrough instructs to install WiX from the official site, then add the WiX extension to Visual Studio (via the VS Installer flow). Steps:

1. We downloaded Wix311 EXE from <https://github.com/wixtoolset/wix3/releases> GitHub page. Run the installer and click Install WiX.

Figure. GitHub WiX file listing for installation

File `wix311.exe` are supposed to be downloaded. Then open the file and install it, we should see the window shown as below.

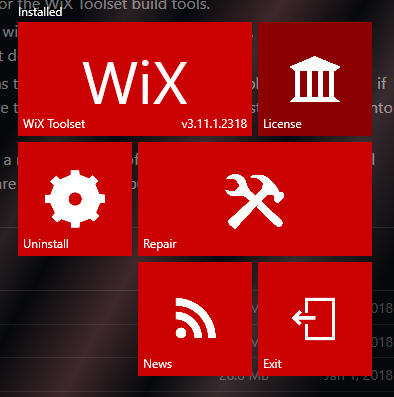


Figure. WiX page after installed with version

1. Return to the Visual Studio Installer and ensure the WiX extension is added/enabled for VS instance.

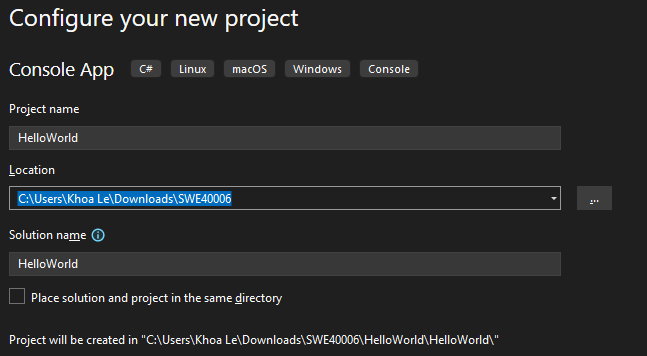
A screenshot of a computer program

Description automatically generated

Figure. Create project template options

**1.1.3 Create a Simple C# “Hello World” Console App**

**1.1.3.1 Project Creation**

1. File → New → Project
2. Choose Console App (.NET) (C#).
3. Framework: select .NET 8.0 (4.6.2 if needed to align with the WiX walkthrough).
4. A screenshot of a computer program

   Description automatically generatedProject name: HelloWorld → Create.

Figure. Creation of HelloWorld Console (C#) project

**1.1.3.2 Program**

**Program.cs:**

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Hello World!");

Console.ReadLine();

}

}

**1.1.3.3 Build & Test**

* Build → Build Solution.
* Run (Ctrl+F5) to verify output.
* Test on HelloWorld.exe.

A screenshot of a computer program

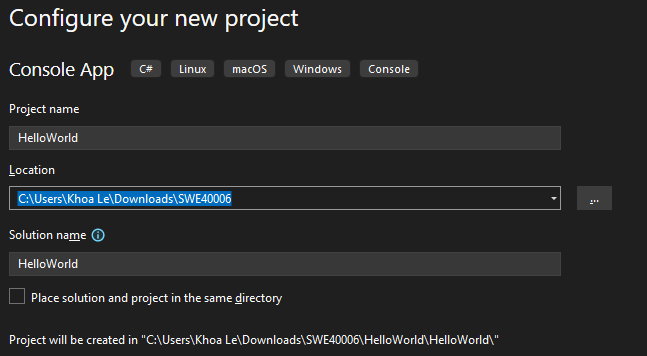
Description automatically generatedA black screen with white text

Description automatically generated

Figures. Program build successfully and HelloWorld app verified on EXE file

**1.1.4 Implement WiX Toolset for the Hello World App (MSI)**

**1.1.4.1 Add a WiX Setup Project (v3)**

1. Right-click the Solution → Add → New Project.
2. A screenshot of a computer program

   Description automatically generatedIn the side bar, choose WiX Toolset v3, select Setup Project for WiX v3.

Figures. Create WiX project

1. Name it HelloWorldInstaller → OK.
2. In HelloWorldInstaller, right-click References → Add Reference → Projects tab → tick HelloWorld → OK.

**1.1.4.2 Edit Product.wxs – include the EXE**

The walkthrough shows placing a <Component> with a <File> element that points to the built EXE. I replaced the content of Product.wxs with the following minimal, project name, UpgradeCode, and file Id and source.

<?xml version="1.0" encoding="UTF-8"?>

<Wix xmlns="http://schemas.microsoft.com/wix/2006/wi">

<Product Id="\*" Name="HelloWorld.exe" Language="1033" Version="1.0.0.0" Manufacturer="HP" UpgradeCode="bbfa057a-20e5-4b0a-bdfa-92f2f4e8c43e">

...

<ComponentGroup Id="ProductComponents" Directory="INSTALLFOLDER">

<Component Id="MainExecutable" Guid="\*">

<File Id="HelloWorldExe"

Source="$(var.HelloWorld.TargetPath)"

KeyPath="yes" />

</Component>

</ComponentGroup>

**Why these edits?**

* Define correct file path remains valid across Debug/Release builds.
* Generate GUIDs via Tools → Create GUID (copy in Registry format, remove braces).
* Set Manufacturer, Name, and INSTALLFOLDER to what we would want users to see in Apps & Features and the install path.
* MediaTemplate EmbedCab="yes" bundles files into the MSI as a single file.

**1.1.4.3 Build & Test the MSI**

1. Right-click HelloWorldInstaller → Build (or Ctrl+Shift+B).

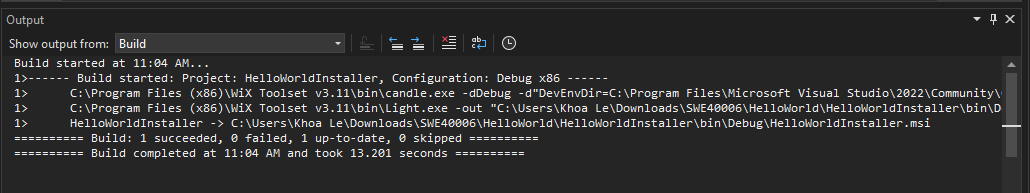


Figure. Program build with WiX project, created MSI file

1. Open the WiX project output directory (e.g., ...\HelloWorldInstaller\bin\Debug\net8.0) now, We should see the .msi.

A screenshot of a computer

Description automatically generated

Figure. MSI file with path to this file from Installer project

1. **Install**: Double-click the MSI and finish the wizard.

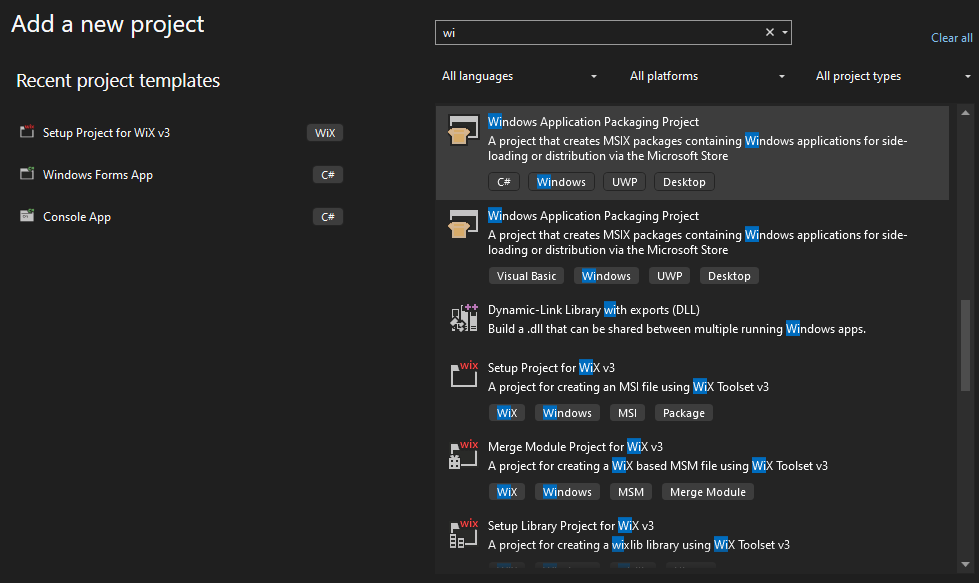
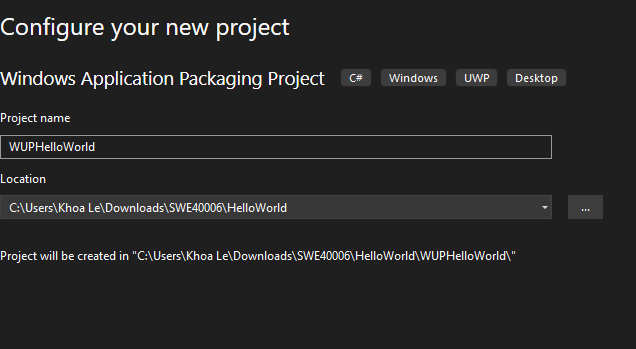
This completes the Hello World packaging with WiX, matching the walkthrough’s **Part 3** outcomes.

**1.1.5 Universal Windows App Packaging**

These tasks follow part 4 of the WiX Walkthrough document.

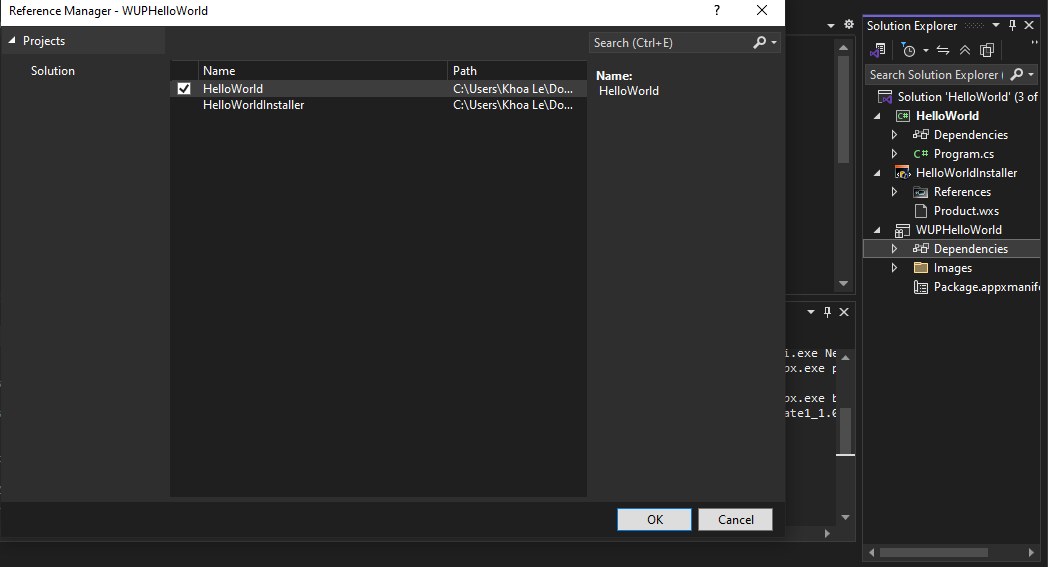
**1.1.5.1 Create Project**

1. We begin with create a Windows Application Packaging (WAP) by clicking to the project’s Solution Explorer → Add → New Project. I named it WUPHelloWorld (it was actually a misspelling that I intentionally name it WAPHelloWorld).

 Figures. Creation of the WAP project

2. Then we add a reference to the project to build by right clicking on WUPHelloWorld project in the solution explorer, then clicking add reference. Then tick this project to install (HelloWorld).

A screen shot of a computer

Description automatically generated

Figures. Adding HelloWorld package to WAP project

3. Right click on the project and go Publish → Create App Packages. This will open a Wizard to walk through creating the package.

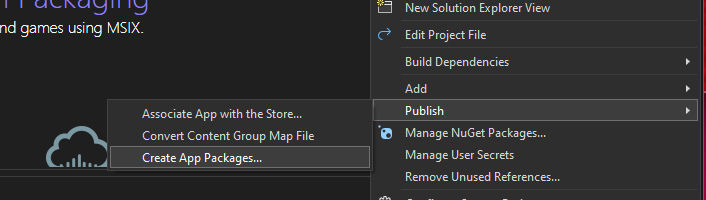


Figure. Create app package for WAP project

Proceed with “Sideloading” option.

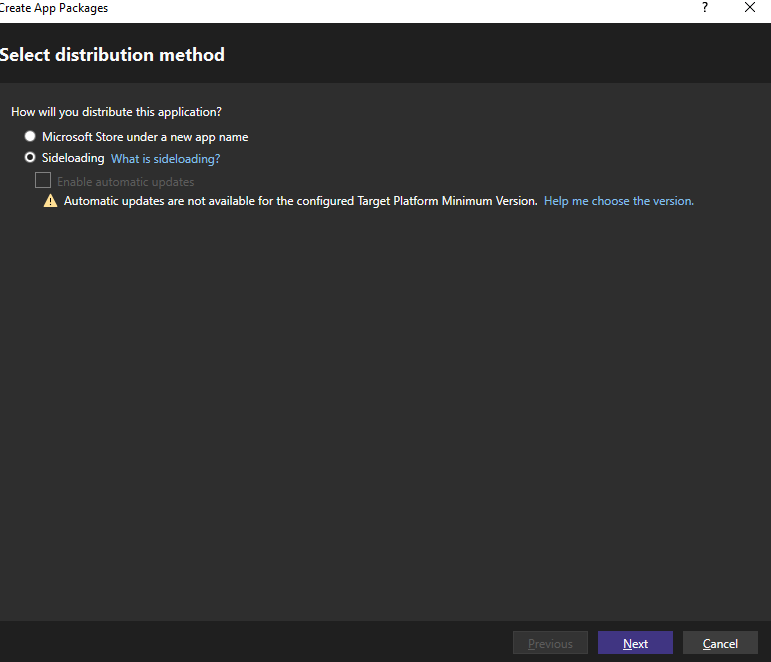


Figure. Distribution method

Then in “signing method” wizard, choose “select a certificate” → Create.

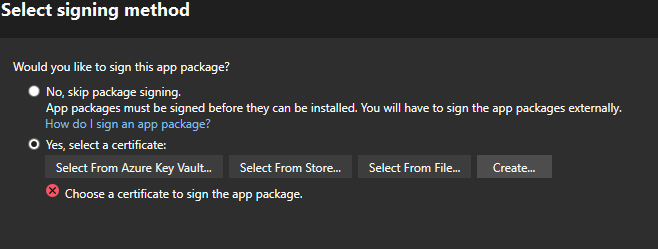


Figure. Signing method

4. Then configure our certificate with a Self-Signed certificate, adding our own Publisher Common Name, and password for the certificate.

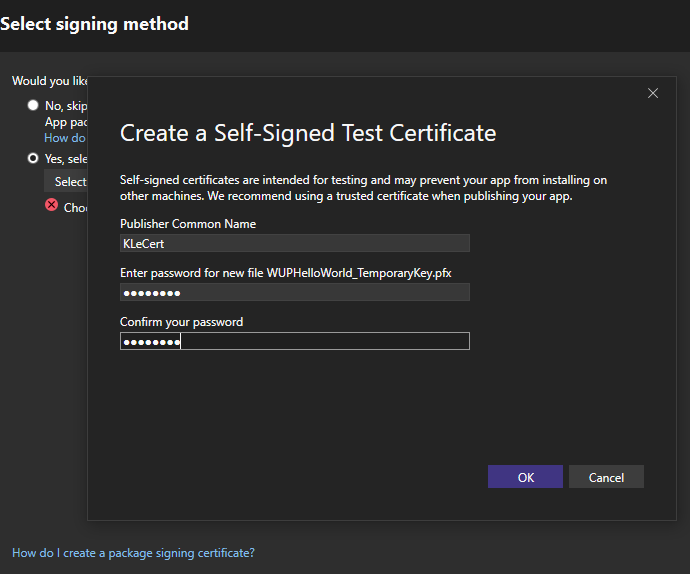


Figure. Create Self-signed certificate with common name and password

(Optional): Make sure our configure package matched, as my local device use x86 format, this supposed to be used for our project also.

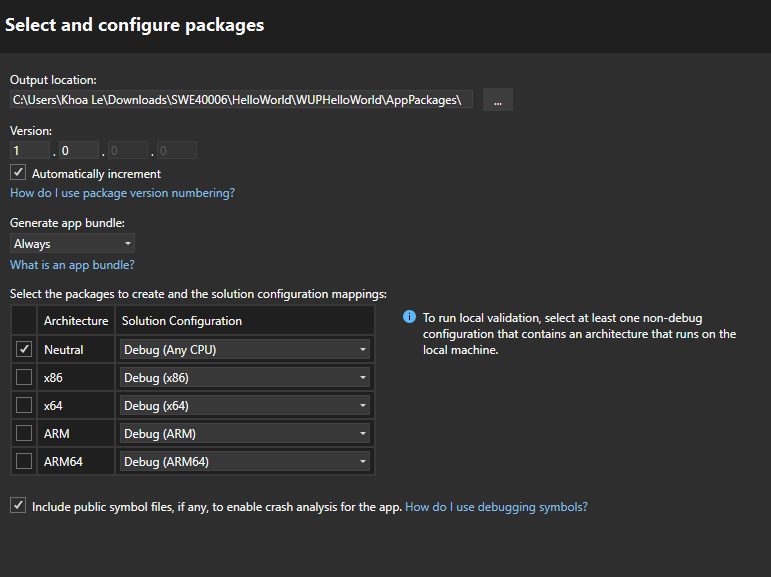


Figure. Package configuration matching device build

I will have to untick Neutral and switch to x86 here. Also, in my project solution (HelloWorld) → Properties, I need to make sure the WiX and WAP projects follow x86 format.

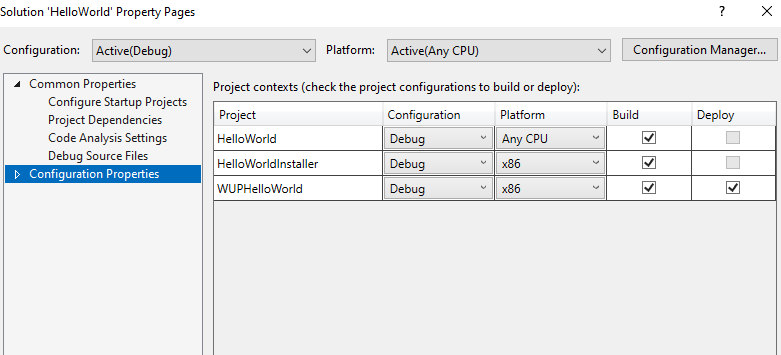


Figure. Solution configuration matching device build

5. Then we click to the WAP project → Build. The terminal below show successful building process.

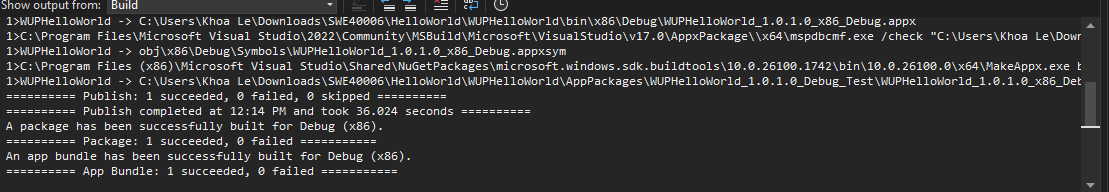


Figure. Successful WAP project build terminal

**1.1.5.2 Configure the certificate and Installation**

1. After build complete, a window pop-up redirecting us to the project package final directory, follow and click to the link.

A black box with white text

Description automatically generated

Figure. Link to app package

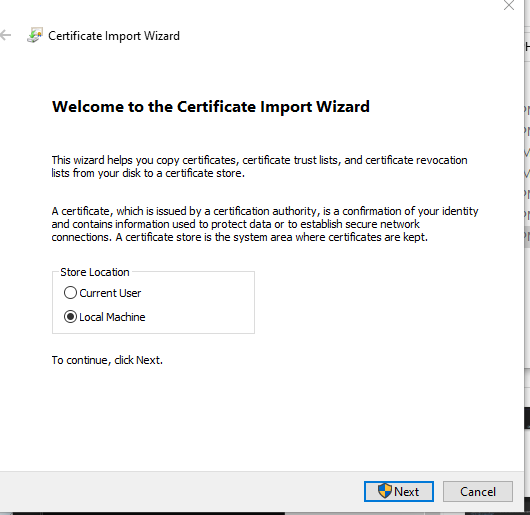
We have the last file (.cer) in the screenshot below, to be our certificate file, click onto it.

A screenshot of a computer

Description automatically generated

Figure. App package files with path

2. It trigger the certificate installation setup guiding wizard, first select “Install certificate”, this is the certificate we made earlier. Proceed with Local Machine option.

A screenshot of a certificate

Description automatically generatedA screenshot of a computer screen

Description automatically generatedA screenshot of a computer

Description automatically generated

Figures. Configure project certificate

3. Next, choose the “Place all certificates in the following store” then “Browse” to  
“Trusted Root Certification Authorities” and click “OK”, “Next” and “Finish”.

4. Our app now is properly configured, click on .appxbundle file (brown colored) in the below image, which pop-up an installation window, proceed with the installation step.

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

Figures. App installation

Once the installation complete, the HelloWorld app will be shown on the terminal as shown.

A screenshot of a computer

Description automatically generated

Figure. Successful build on terminal after the installation

**1.2 Calculator App – Deploy Simple Desktop (Credit Task)**

We extended from the Hello World project by creating a more functional Calculator application using Windows Forms (WinForms) and deploying it via WiX Toolset and WAP packaging.

**1.2.1 Program Foundation (Windows Forms App, .NET 8.0)**

1. File → New → Project → Windows Forms App (C#)
2. **Framework:** .NET 8.0.
3. **Name:** Calculator.

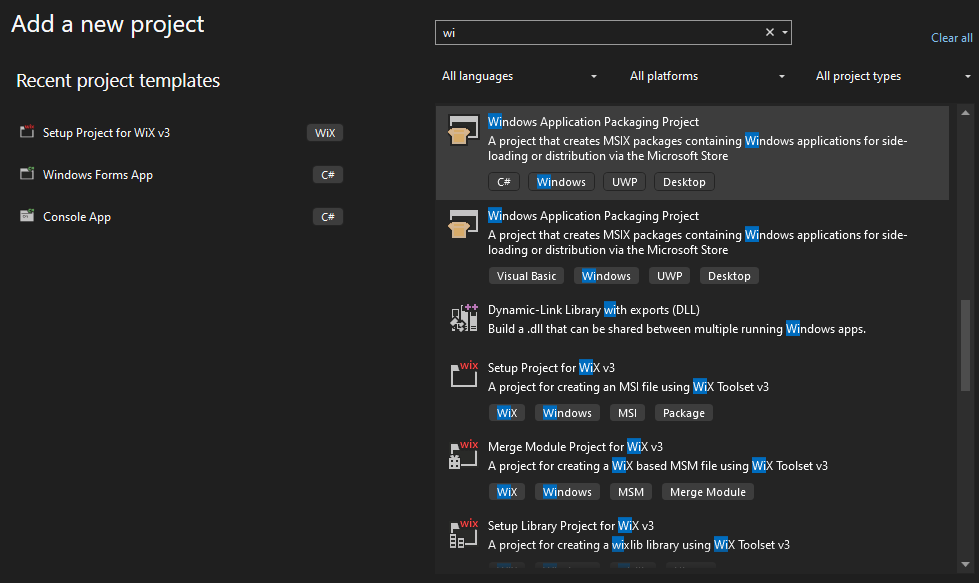


Figure. WFA option from project Templates

**A. UI Layout - Form1.Designer.cs:**

**Step 1: Define the UI Elements**

We start by laying out the controls in the Form1.Designer.cs file. These represent the input fields, buttons, and labels users will interact with.

namespace CalculatorApp

{

partial class Form1

{

private System.Windows.Forms.TextBox txtA;

private System.Windows.Forms.TextBox txtB;

private System.Windows.Forms.Label lblResult;

private System.Windows.Forms.Button btnAdd;

private System.Windows.Forms.Button btnSub;

private System.Windows.Forms.Button btnMul;

private System.Windows.Forms.Button btnDiv;

private System.Windows.Forms.Label lblTitle;

**Justification:**

* txtA and txtB are the two number inputs.
* lblResult displays the output after calculation.
* btnAdd, btnSub, btnMul, btnDiv trigger arithmetic operations.
* lblTitle makes the form user-friendly by clearly labelling it Simple Calculator.

**Step 2: Initialize and Position Controls**

The InitializeComponent() method positions controls on the form and sets basic properties like size, text, and click-event bindings.

private void InitializeComponent()

{

this.txtA = new System.Windows.Forms.TextBox();

this.txtB = new System.Windows.Forms.TextBox();

this.lblResult = new System.Windows.Forms.Label();

this.btnAdd = new System.Windows.Forms.Button();

// txtA

this.txtA.Location = new System.Drawing.Point(30, 60);

this.txtA.Name = "txtA";

this.txtA.Size = new System.Drawing.Size(100, 20);

this.txtA.TabIndex = 0;

**Justification:**

* Each control is given a **location** (x, y coordinates), **size**, and **name**.
* The TabIndex sets the order for keyboard navigation.
* This design keeps inputs aligned horizontally (txtA at 30px, txtB at 150px), followed by results and buttons below.

**Step 3: Result Display Label**

this.lblResult.AutoSize = true;

this.lblResult.Font = new System.Drawing.Font("Segoe UI", 12F, System.Drawing.FontStyle.Bold);

this.lblResult.Location = new System.Drawing.Point(30, 100);

this.lblResult.Name = "lblResult";

this.lblResult.Size = new System.Drawing.Size(79, 21);

this.lblResult.TabIndex = 2;

this.lblResult.Text = "Result = ";

**Justification:**

* The **bold font** makes results easy to read.
* Label initialized to "Result =" so users always know where to look after pressing a button.

**Step 4: Add the Operation Buttons**

this.btnAdd.Location = new System.Drawing.Point(30, 150);

this.btnAdd.Name = "btnAdd";

this.btnAdd.Size = new System.Drawing.Size(75, 30);

this.btnAdd.TabIndex = 3;

this.btnAdd.Text = "Add (+)";

this.btnAdd.UseVisualStyleBackColor = true;

this.btnAdd.Click += new System.EventHandler(this.btnAdd\_Click);

**Justification:**

* Each button is labelled with both **symbol** and **text** (e.g., "Add (+)") to reduce confusion.
* Event handlers (btnAdd\_Click, btnSub\_Click, etc.) are bound here, so when the user clicks a button, the program jumps to the correct function.

The same pattern is applied to Subtract, Multiply, and Divide buttons, positioned evenly at 90px intervals for neat layout.

**Step 5: Title Label for Better UX**

this.lblTitle.Font = new System.Drawing.Font("Segoe UI", 14F, System.Drawing.FontStyle.Bold);

this.lblTitle.Location = new System.Drawing.Point(120, 15);

this.lblTitle.Name = "lblTitle";

this.lblTitle.Size = new System.Drawing.Size(174, 25);

this.lblTitle.TabIndex = 7;

this.lblTitle.Text = "Simple Calculator";

**Step 6: Form Settings**

**Justification:**

* Each button is labelled with both **symbol** and **text** (e.g., "Add (+)") to reduce confusion.
* Event handlers (btnAdd\_Click, btnSub\_Click, etc.) are bound here, so when the user clicks a button, the program jumps to the correct function.

The same pattern is applied to Subtract, Multiply, and Divide buttons, positioned evenly at 90px intervals for neat layout.

**B. Event Handler - Form1.cs:**

Now we move from the **UI layout** to the **logic** that executes when buttons are clicked.

**Step 7: Input Parsing Helper**

private double ParseInput(TextBox txt)

{

return double.TryParse(txt.Text, out var val) ? val : 0;

}

**Justification:**

* Centralized function to **safely parse** input from txtA or txtB.
* Returns 0 if parsing fails → avoids runtime crashes if user types invalid text.
* Makes code DRY (Don’t Repeat Yourself), since parsing is needed for every operation.

**Step 8: Mathematical Operation**

private void btnAdd\_Click(object sender, EventArgs e)

{

double a = ParseInput(txtA);

double b = ParseInput(txtB);

lblResult.Text = "Result = " + (a + b).ToString();

}

private void btnDiv\_Click(object sender, EventArgs e)

{

double a = ParseInput(txtA);

double b = ParseInput(txtB);

lblResult.Text = (b == 0)

? "Result = ∞"

: "Result = " + (a / b).ToString("0.###");

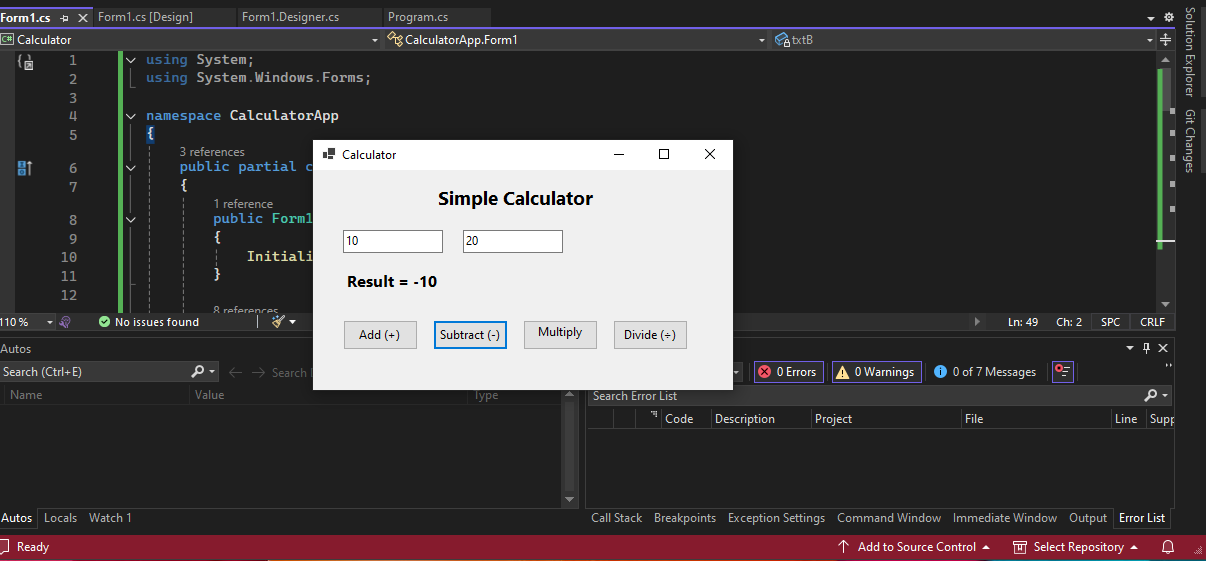
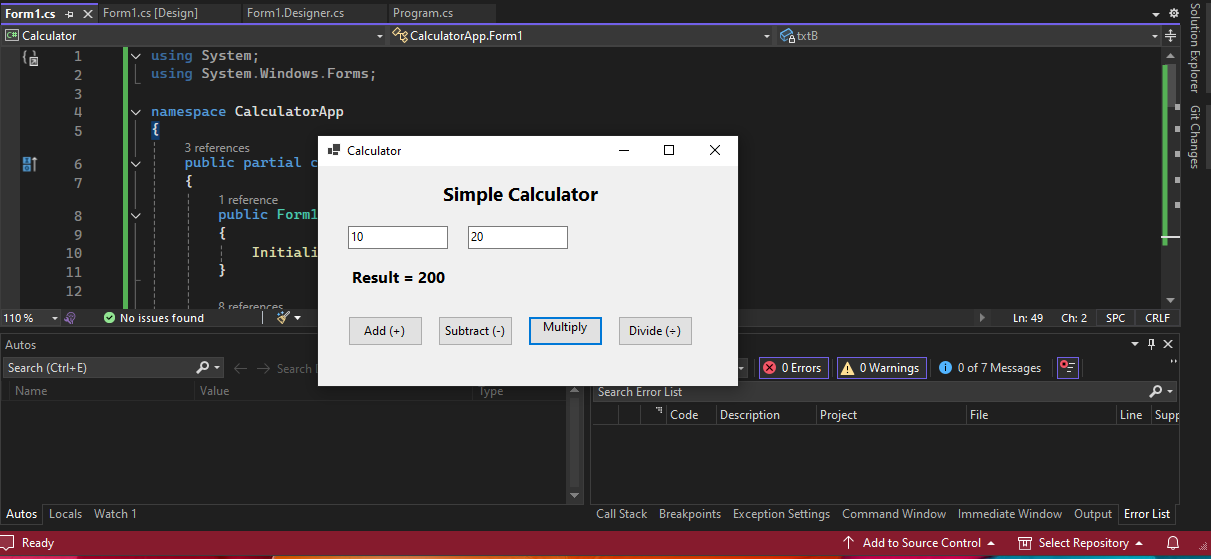
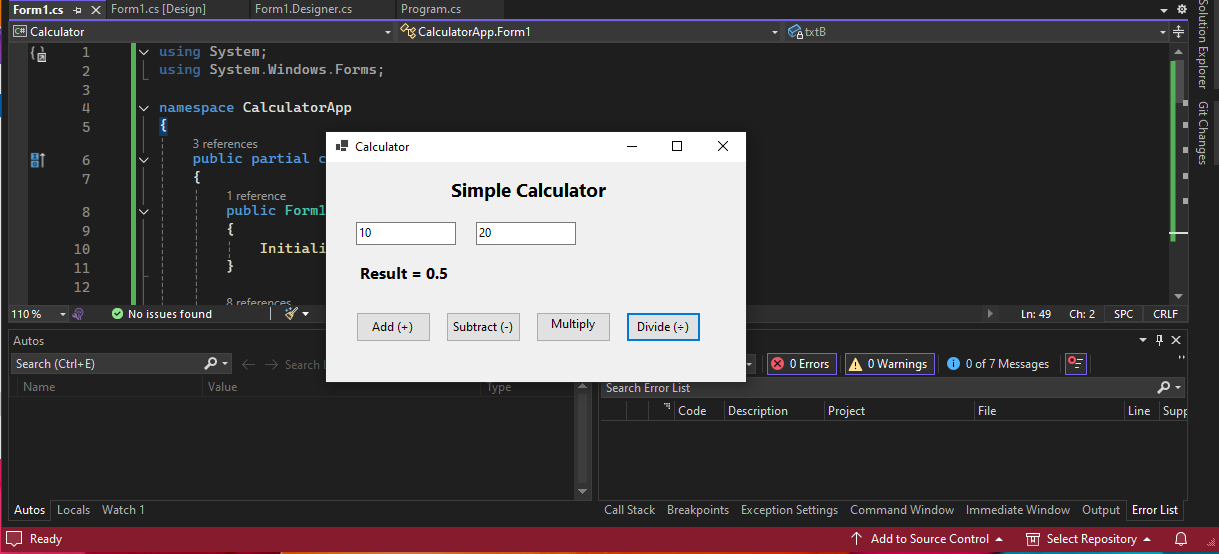
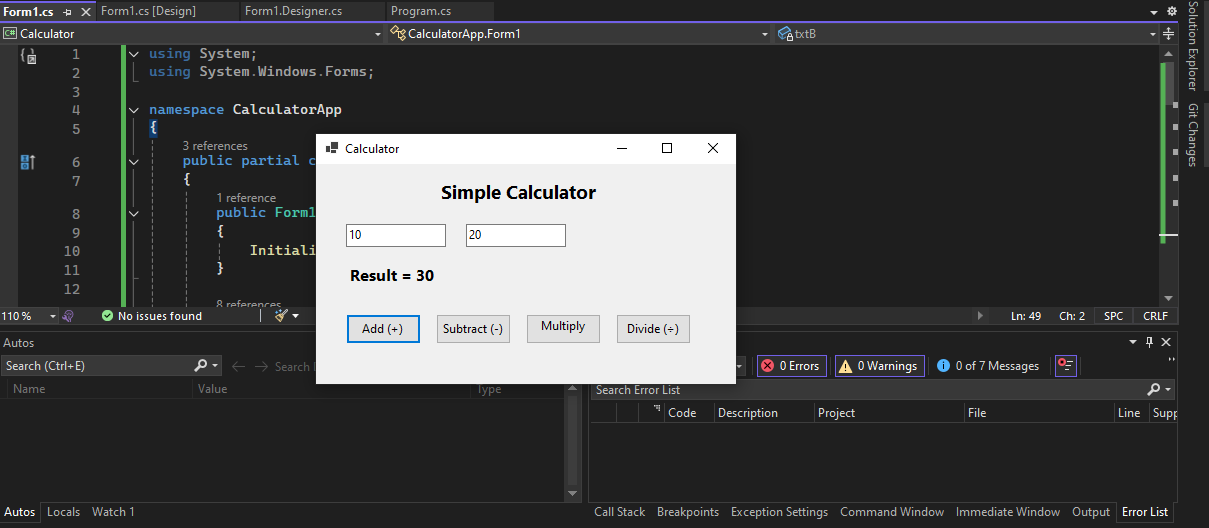
}

**Justification:**

* For each of Add, Subtract and Multiply actions, it reads both inputs, perform mathematical computation (+, -, \*) between the 2 input a & b, and updates lblResult.
* For Divide, it also perform:
  + Adds error handling for division by zero.
  + Displays infinity (∞) instead of crashing or showing incorrect results.
  + Formats division result to 3 decimal places for readability.
* String concatenation "Result = " keeps UI consistent.

**C. Build & Test Calculator App**

* Project created as **WinForms App (.NET 8.0)**.
* Form1.Designer.cs and Form1.cs implemented as shown earlier.
* Build succeeded with a functional UI supporting addition, subtraction, multiplication, and division with error handling for divide-by-zero.



Figures. App Demo with accurate computation output

Calculator app featuring accurate mathematical equation computations (add, subtract, multiply and divide actions) and well-friendly design panel.

**1.2.2 WiX installation**

* Added **WiX v3 Setup Project** to the solution (CalculatorInstaller).
* Added **Project Reference** → Calculator.

A screenshot of a computer program

Description automatically generated

Figure. Reference WiX project with WFA project

* Edited Product.wxs to point to Calculator EXE and add installation folder.

<ComponentGroup Id="ProductComponents" Directory="INSTALLFOLDER">

<Component Id="MainExecutable" Guid="\*">

<File Id="CalculatorExe"

Source="$(var.Calculator.TargetPath)"

KeyPath="yes" />

</Component>

</ComponentGroup>

* Build → MSI produced successfully.

A computer screen shot of a program

Description automatically generated

Figure. WiX project successful build, generate MSI file

* Installed via .msi → confirmed working Calculator in Program Files.

A screenshot of a computer

Description automatically generated

Figure. Install the app from MSI

**1.2.3 Windows Application Packaging**

* Added a **Windows Application Packaging Project** to the solution.

A screenshot of a black screen

Description automatically generatedA screenshot of a computer

Description automatically generated

Figure. Add WAP project

* A screenshot of a computer

  Description automatically generatedReferenced CalculatorApp → configured certificate (self-signed).

Figure. Reference WAP project

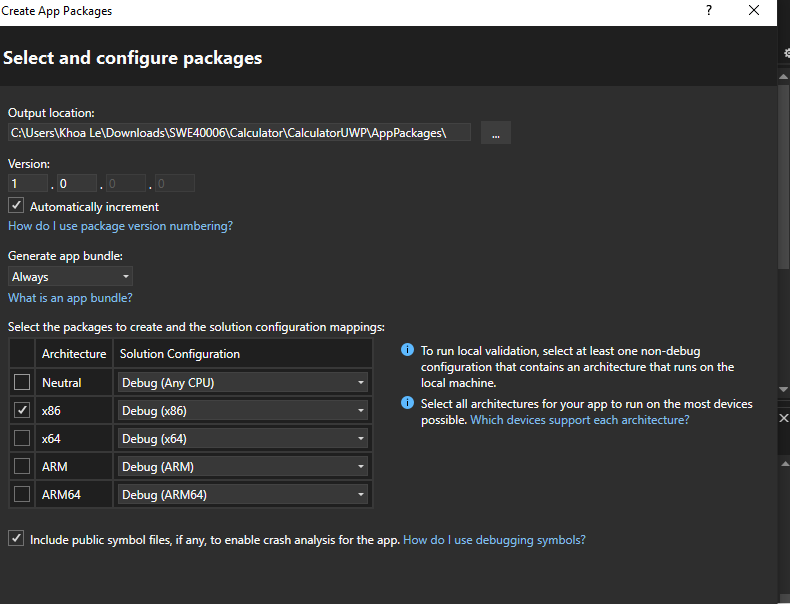
* Similarly, I need to set x86 matching my system configuration.

Figure. Configure package with x86

* A screenshot of a computer program

  Description automatically generatedBuilt .appxbundle for sideload testing.

Figure. Successful build from the app bundle, ready for certification and installation

* Installed certificate to Trusted Root (Local Machine → Trusted Root Certificaiton Authorities)→ installed bundle successfully.

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generatedFigure. Install certificate and the app bundle

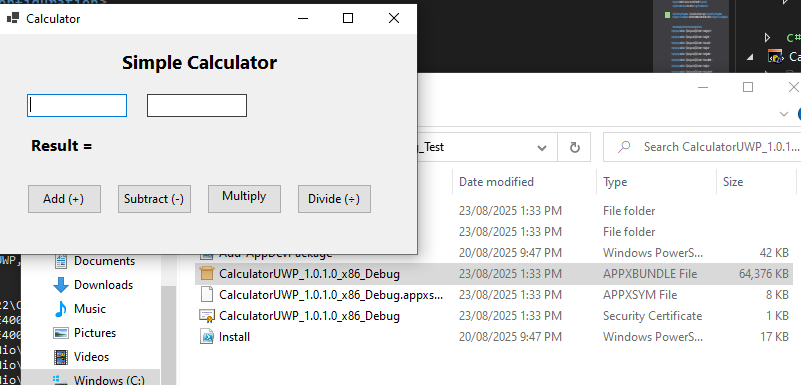
* Calculator now appears in Windows Start Menu as a packaged Windows Application Packaging app.

Figure. Calculator app

**Task 1.3 – Multi-DLL Application (Distinction Task)**

**1.3.1 App Concept and Theme**

For this task, we implemented a **“Personal Finance Manager”** desktop application.

**Overview (what the app does)**

* **Core features:**
  + Add transactions with description, amount, category; maintain a running **balance**.
  + Generate a **category total** report (and a monthly total API in the library).
  + **Export** all transactions to a CSV file.
* **Architecture:**
  + **WinForms UI** (Form1 + Program) orchestrates interactions.
  + **TransactionLibrary** manages the in-memory ledger.
  + **ReportLibrary** computes category and monthly aggregations.
  + **ExportLibrary** handles CSV export.
* **Goal:** Help users track incomes and expenses, calculate balances, and generate simple statistics.
* **Reason for this theme:** Finance apps naturally separate into modules (core logic, data storage, analytics), making it an excellent candidate to demonstrate DLL modularization.
* **Architecture:**
  + FinanceManagerApp.exe → Main WinForms application (UI only).
  + TransactionLibrary.dll → Handles adding/removing transactions and balance logic.
  + ReportLibrary.dll → Provides reporting & analytics (monthly totals, category stats).
  + ExportLibrary.dll → Supports exporting reports to text/CSV files.

This separation simulates a real professional deployment where business logic, reporting, and utilities are encapsulated as separate reusable libraries.

Initialisation (similar to Credit Task):

1. File → New → Project → Windows Forms App (C#)
2. **Framework:** .NET 8.0.
3. **Name:** FinanceManagerApp.

**1.3.2 Setups**

Once created the WFA project, go to project solution in Solution Explorer:

* File → Add → New Project.
* Select **Class Library (.NET)** -> .NET8.0.
* A screenshot of a computer

  Description automatically generatedName them:
  + TransactionLibrary
  + ReportLibrary
  + A screenshot of a computer

    Description automatically generatedExportLibrary

Figure. Create new project as Class Library (.NET)

Rename the class file appropriately for better readability and program understandability (for instance rename Class1.cs as default to Exporter.cs matching ExportLibrary’s public class name). Similarly, rename other class file accordingly.

A screenshot of a computer

Description automatically generatedA screenshot of a computer program

Description automatically generated

Figure. Rename class files

Our multi-DLL project will have cross-reference between each DLL/library. Hence, for each that we are going to import another libraries from, we need to add references.

First, add References in the WinForms project (FinanceManagerApp)

* Right-click FinanceManagerApp → Dependencies → Add Project Reference.
* Tick the three library projects (TransactionLibrary, ReportLibrary, ExportLibrary).
* Click OK.

A screenshot of a computer program

Description automatically generatedA screenshot of a computer

Description automatically generated

Figure. Add referencing to the main app

Similarly, ReportLibrary and ExportLibrary need reference from TransactionLibrary, so we repeated the same thing.

**1.3.3 DLL Implementations**

**1.3.3.1 Program.cs: application entry point**

**Features**

* Sets up WinForms app and launches the main form. It enables visual styles, uses GDI text rendering compatibility, and opens Form1. This is the correct minimal bootstrap for a classic WinForms app.
* [STAThread] is required by WinForms and many shell components (for .NET6.0+).
* EnableVisualStyles() and SetCompatibleTextRenderingDefault(false) are standard for modern UI look & feel.

**1.3.3.2 Form1.Designer.cs: UI layout and wiring**

**Design Features**

* **Window Frame (Form):**
  + Title set to **“Finance Manager”** (the app’s identity).
  + Start position is **center screen** so the window appears centered on launch.
  + Fixed client size (350 x 270) makes layout consistent across systems.
* **Title Label (lblTitle):**
  + Large, bold font (“Segoe UI 14pt, Bold”).
  + Displays **“Personal Finance Manager”** at the top.
  + Serves as the visual entry point, making the form look professional.
* **Description Label (lblDesc) and Input (txtDesc):**
  + Positioned at (30, 70), label shows **“Description:”**.
  + Input field (txtDesc) located beside it at (120, 67) with width 150.
  + Purpose: lets the user type what the transaction is (e.g., *“Groceries”*, *“Salary”*).
* **Amount Label (lblAmount) and Input (txtAmount):**
  + Positioned slightly below description ((30,100)).
  + Input field beside at (120, 97), width 150.
  + Purpose: lets user enter **transaction value** (positive/negative).
* **Category Label (lblCategory) and Drop-down (cmbCategory):**
  + Positioned at (30, 130).
  + Drop-down box at (120, 127), width 150.
  + Items pre-filled: **Income, Food, Transport, Bills, Entertainment**.
  + Purpose: allows user to classify each transaction quickly and consistently.
* **Action Buttons:**
  + **Add Txn (btnAdd)**:
    - Positioned (30, 170), labeled **“Add Txn”**.
    - When clicked → triggers btnAdd\_Click.
    - Purpose: add a new transaction (desc, amount, category) into the ledger.
  + **Category Report (btnReport)**:
    - Positioned (120, 170), labeled **“Category Report”**.
    - When clicked → triggers btnReport\_Click.
    - Purpose: generate total spending/income for the selected category.
  + **Export CSV (btnExport)**:
    - Positioned (210, 170), labeled **“Export CSV”**.
    - When clicked → triggers btnExport\_Click.
    - Purpose: export the ledger to a CSV file for external viewing.
* **Balance Label (lblBalance):**
  + Positioned at (30, 220), bold 12pt font.
  + Initial text **“Balance: $0.00”**.
  + Purpose: always show running balance after transactions are added.

**Design Concept**

* **Functional Grouping:**
  + Top = app title.
  + Middle = user input section (desc, amount, category).
  + Bottom = actions and output indicator (buttons, balance).
* **UX:**
  + **Sequential order** (desc → amount → category → action) mirrors natural workflow.
  + **Combo box** ensures consistent category classification, preventing user typos.
* **Wiring:**
  + All buttons are **connected** to event handlers in Form1.cs.
  + Layout handled by Designer → logic code remains clean.

**1.3.3.3 Form1.cs: UI controller and event handlers**

**Declared Features**

* **Transaction Manager Instance:**
  + TransactionManager manager = new();
  + Maintains an in-memory list of transactions for the session.
  + Ensures one consistent ledger is shared across Add, Report, and Export actions.
* **Form Initialization:**
  + Constructor (public Form1()) calls InitializeComponent() to build UI.
  + Ensures UI and event bindings are ready before user interaction.
* **Add Transaction (btnAdd\_Click):**
  + Reads **amount** from txtAmount.
  + Reads **description** from txtDesc.
  + Reads **category** from cmbCategory.
  + Calls manager.AddTransaction(...) → ledger updated.
  + Updates **Balance Label** with formatted currency string (e.g., *Balance: $125.00*).
  + Clears inputs for the next entry (txtDesc.Clear(), txtAmount.Clear()).
  + **Feature:** Allows user to continuously add transactions while always seeing the current balance.
* **Generate Report (btnReport\_Click):**
  + Reads selected category from cmbCategory.
  + Calls ReportGenerator.TotalByCategory(...) with all transactions.
  + Shows result in a **MessageBox** (e.g., *“Total for Food: $220.00”*).
  + **Feature:** Gives immediate insight into category-specific spending/income.
* **Export Transactions (btnExport\_Click):**
  + Calls Exporter.ExportToCsv(manager.GetAll(), "transactions.csv");.
  + Writes all current ledger transactions into transactions.csv.
  + Shows confirmation MessageBox: *“Export complete: transactions.csv”*.
  + **Feature:** Provides external portability of data — user can open in Excel or share records.

**A screenshot of a computer

Description automatically generated**

Figure. App Demo – Form 1 sketch

**Design Concept**

* **Coordinator role:**
  + Form1 doesn’t calculate or store directly — it calls specialized libraries:
    - TransactionManager (ledger management),
    - ReportGenerator (analytics),
    - Exporter (I/O).
* **Thin UI logic:**
  + Keeps presentation separated from domain logic → modularity and testability.
* **Flow for user:**
  + Enter → Add → Balance updated → (optional) Report/Export.

private void btnAdd\_Click(object sender, EventArgs e)

{

double amount = double.TryParse(txtAmount.Text, out var a) ? a : 0;

string category = cmbCategory.Text;

manager.AddTransaction(txtDesc.Text, amount, category);

lblBalance.Text = $"Balance: {manager.GetBalance():C}";

txtDesc.Clear();

txtAmount.Clear();

}

**1.3.3.3 TransactionManager.cs (TransactionLibrary)**

**Declared Features**

* **Domain Entity: Transaction**
  + **Properties:**
    - Description → a short text describing the transaction (*“Groceries”*, *“Salary”*).
    - Amount → numeric value (positive for income, negative for expenses).
    - Category → classification label (e.g., Food, Transport).
    - Date → timestamp of when the transaction was created (defaults to DateTime.Now).
  + **Feature:** Encapsulates the data structure of a financial record.
* **Ledger Manager: TransactionManager**
  + Maintains a private **list of Transaction objects** (List<Transaction>).
  + **AddTransaction(string desc, double amount, string category):**
    - Creates and appends a new transaction to the ledger with the provided details.
    - Stamps Date with the system time when added.
    - **Feature:** Provides a controlled way of inserting consistent records into the ledger.
  + **GetAll():**
    - Returns an enumerable collection of all transactions stored.
    - **Feature:** Supports iteration, reporting, exporting, or displaying in UI.
  + **GetBalance():**
    - Computes the sum of all transaction amounts.
    - **Feature:** Instantly returns the current net balance (income – expenses).

**Concept**

* Acts as the **backbone of the app’s business logic**.
* Keeps **ledger integrity** (every transaction stored, retrievable, and aggregatable).
* The UI doesn’t manage lists directly — it delegates to this library.

private List<Transaction> transactions = new();

public void AddTransaction(string desc, double amount, string category)

{

transactions.Add(new Transaction

{

Description = desc,

Amount = amount,

Category = category,

Date = DateTime.Now

});

}

public IEnumerable<Transaction> GetAll() => transactions;

public double GetBalance() => transactions.Sum(t => t.Amount);

}

**1.3.3.4 ReportGenerator.cs (ReportLibrary)**

**Features**

* **Static class ReportGenerator** (no state, purely functional).
  + Designed for reusable calculations based on a given transaction list.
* **TotalByCategory(IEnumerable<Transaction> txns, string category):**
  + Filters transactions where t.Category == category.
  + Sums their amounts and returns the total.
  + **Feature:** Provides quick per-category insights (e.g., “Food total = $220.00”).
* **MonthlyTotal(IEnumerable<Transaction> txns, int month, int year):**
  + Filters transactions where Date.Month and Date.Year match the parameters.
  + Computes the total of amounts in that time frame.
  + **Feature:** Enables time-based analysis (e.g., “Total spending in March 2025”).

**Concept**

* Serves as the **analytics engine** of the app.
* Separates calculation logic from storage (TransactionManager) and presentation (Form1).
* Allows extension with more statistical methods without touching the UI.

public static double TotalByCategory(IEnumerable<Transaction> txns, string category)

=> txns.Where(t => t.Category == category).Sum(t => t.Amount);

public static double MonthlyTotal(IEnumerable<Transaction> txns, int month, int year)

=> txns.Where(t => t.Date.Month == month && t.Date.Year == year)

.Sum(t => t.Amount);

**1.3.3.5 Exporter.cs (ExportLibrary)**

**Features**

* **Static Exporter class** (utility role, no persistent state).
* **ExportToCsv(IEnumerable<Transaction> txns, string filePath):**
  + Opens a file stream at the provided filePath.
  + Writes a **CSV header**: Date,Description,Category,Amount.
  + Iterates through all transactions and writes each as a CSV row:
    - Date (from Transaction.Date).
    - Description (text entered by user).
    - Category (Food, Income, etc.).
    - Amount (numeric value of transaction).
  + **Feature:** Produces a portable text file readable by Excel, Google Sheets, and other apps.

**Concept**

* Provides **data portability**.
* Allows users to **export their financial history** for record-keeping, sharing, or external analysis.
* Cleanly isolated from UI and logic — so export can later support different formats (e.g., JSON, XML, PDF) without affecting other components.

public static void ExportToCsv(IEnumerable<Transaction> txns, string filePath)

{

using var writer = new StreamWriter(filePath);

writer.WriteLine("Date,Description,Category,Amount");

foreach (var t in txns)

{

writer.WriteLine($"{t.Date},{t.Description},{t.Category},{t.Amount}");

}

**How They Fit Together**

1. **TransactionManager (ledger)** → adds, stores, and calculates balance.
2. **ReportGenerator (analytics)** → filters and computes category/time-based totals from the ledger.
3. **Exporter (I/O)** → converts ledger contents into an external CSV file.
4. **Form1 (UI)** → orchestrates all three:
   * calls AddTransaction to update ledger,
   * calls ReportGenerator to show category totals,
   * calls Exporter to save records to disk.

Build all 4 of our libraries (FinanceManagerApp and the 3 DLL libraries)

A computer screen shot of a computer screen

Description automatically generated

Figure. Successful build on the multi-DLL app

**1.3.4 WiX Installation**

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generatedCreate new WiX project (Named FinanceManagerInstallation) → Link all of the 4 project with this.

Figure. Reference WiX project to WFA and Class Libraries projects

Configure a custom Product.wxs for this multi-DLL app, similar setup to the previous Pass and Credit tasks, however, we have to create different Components tags in ComponentGroup for each of our DLL libraries (linked with DLL file on built) and the Main app (linked with EXE file on built):

<ComponentGroup Id="ProductComponents" Directory="INSTALLFOLDER">

<!-- Main App -->

<Component Id="FinanceExe" Guid="\*">

<File Id="FinanceAppExe"

Source="$(var.FinanceManagerApp.TargetPath)"

KeyPath="yes"/>

</Component>

<!-- Transaction Library -->

<Component Id="TransactionLib" Guid="\*">

<File Id="TransactionLibraryDll"

Source="$(var.TransactionLibrary.TargetPath)"

KeyPath="yes"/>

</Component>

...

</ComponentGroup>

* Build → MSI produced successfully.

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

Figures. WiX build and MSI file creation

* Installed via .msi → confirmed working FinanceManager in Program Files.

**1.3.5 Window Application Packaging Project**

* A screenshot of a computer

  Description automatically generated**A screenshot of a computer

  Description automatically generated**Add new Windows Application Packaging project to the solution, named FinanceManagerUWP. Add references to this project by our main app – FinanceManagerApp (other DLL doesn’t need to be referred since they has been referenced by this main app, and they don’t have an executable file, e.g., EXE)

Figure. WAP project creation and reference only the main app (WFA)

* A screenshot of a computer

  Description automatically generatedThen Publish → Create App Packages.

Figure. WAP project creation and reference only the main app (WFA)

* A screenshot of a computer

  Description automatically generatedA screenshot of a computer

  Description automatically generatedConfigured certificate (self-signed), named FinanceManagerKLeCert. Also for our system, we have to consistently choose x86. Once publish complete, navigate to the output location as instructed on pop-up window.

Figures. Configure self-singed certificate and proceed through with the app package path

* Install our certificate (Local Machine → Trusted Root Certificaiton Authorities) → installed bundle successfully.

A screenshot of a computer

Description automatically generatedA screenshot of a certificate

Description automatically generatedFigures. Installation of certificate and WAP project

A screenshot of a computer screen

Description automatically generatedA screenshot of a computer screen

Description automatically generated**1.3.5 App Demo**

Figures. Add task feature, check balance

Designed for financial management and recording, we can add any money in/out flow with custom amount (float), description of the task and its category. Our total balance will be summed up and displayed, as we click to “Add Txn” button.

Figures.A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated Check categorial total

A screenshot of a computer screen

Description automatically generatedA screenshot of a spreadsheet

Description automatically generatedUser can check for which specific category (filtered in drop-down menu), then click “Category” button to display their sum-up total for that sector only.



Figures. Export and review CSV record

The last feature is CSV exporting, user can click to “Export CSV” to check the financial ledger.

**1.4 Publish App to Microsoft Store (High Distinction)**

**Step 1: Create / access Partner Center developer account**

1. A screenshot of a computer

   Description automatically generatedSign in to **Microsoft Partner Center** at: <https://partner.microsoft.com/en-us/dashboard/home>. Create a Microsoft or sign in if you have an account already.

Figure. Microsoft Partner Center dashboard

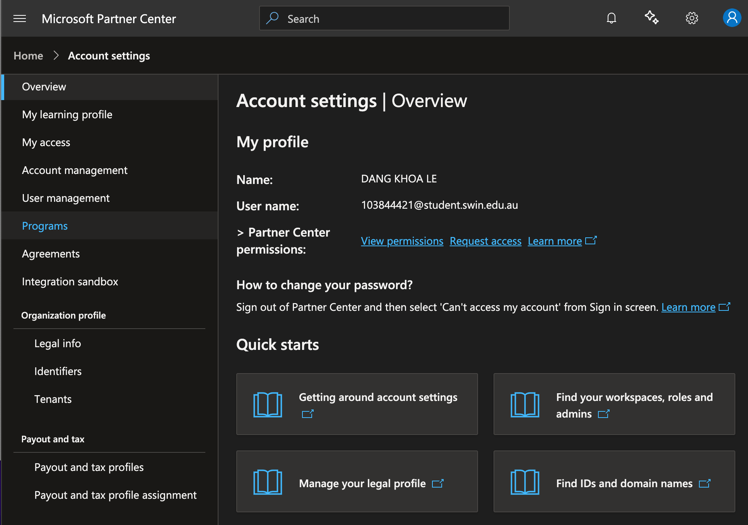
1. Go to Settings (Gear icon) → Account settings → Programs

Figure. Microsoft Partner Center Programs page

1. Find Windows → Proceed with Windows and Xbox sign-in page.

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

Figures. Proceed with Windows and Xbox option

1. Once they prompt-in to verify account, make sure we have to sign-in with personal account (not school/work account if activating). This is because school/work account may have their own preferences configured and not meant for individual/business partnership with Microsoft.

A screenshot of a computer

Description automatically generatedA screenshot of a login box

Description automatically generated

Figures. Currently in school account

1. I’m using my personal account now. Then proceed with Windows and Xbox option.

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

Figures. Logout and re login again with personal account

1. Then we need to configure our account information and proceed through with company name and details. Nonetheless, since a registration fee is requisite, we would not pay it there but rather we will continue the scenario of deployment with detailed explanation of which steps we would have to go and finally deploy the app simulatively.

A screenshot of a computer

Description automatically generatedA screenshot of a computer screen

Description automatically generated

Figures. Forms to be filled

**Step 2: Associate packaging project with the reserved app**

In order to do this, we need to with our Windows Application packaging.

* Right-click packaging project → **Publish → Associate App with the Store.**
* A screenshot of a computer

  Description automatically generatedChoose **Microsoft Store**; sign in; select **reserved app name** when prompted.

A screenshot of a computer program

Description automatically generated

Figures. Associate our app with Microsoft Store from Visual Studio

However, since we can’t have the developer account, we will continue this with real-life practice simulation.

A screenshot of a computer

Description automatically generated

Figures. Reserve app name and sign in requisite

Association pulls down the Store identity (Publisher, DisplayName, Package Identity) into packaging project automatically.

**Step 3: Check package identity, version, architecture & assets**

1. Open Package.appxmanifest (Design).
2. **Packaging**: confirm **Publisher** & **Package display name** match reserved app.
3. **Version**: set 1.0.0.0 (increment per release just like in real-world context).
4. **Visual Assets**: ensure icons/tiles are filled (PNG).
5. **Architectures**: Since I standardized on **x86**. I will **x86 only** now; later add **x64** for broader reach.

Since my app is .NET 8, it’s preferably to **self-contained** publishing (win-x86) to avoid runtime prerequisites. You can set this in the app’s Publish profile (or via .csproj); MSIX will include EXE + DLLs from the app’s output folder.

E.g., in FinanceManagerApp.csproj:

<PropertyGroup>

<RuntimeIdentifier>win-x86</RuntimeIdentifier>

<SelfContained>true</SelfContained>

</PropertyGroup>

**Step 4: Build Store packages (.msixupload / .appxupload)**

1. Right-click the packaging project → **Publish → Create App Packages.**
2. Select **Microsoft Store** (not sideloading).
3. Choose **x86** architecture (and **Generate app bundle = Always**).
4. Complete the wizard to produce the **Store upload package**.

**Step 5: Validate with Windows App Certification Kit (WACK)**

1. Launch **Windows App Certification Kit**.
2. Select**.msixbundle**, run the full test suite.
3. Resolve any issues, rebuild, re-test until “Passed”.

**Step 6: Create a Store submission**

1. Go back to **Partner Center** → open app → **Create a new submission**.
2. Complete each required section:

**Packages:**

* Upload **.msixupload** (or .appxupload) from Step 4.

**Store Listings**

* Title (e.g., “Finance Manager – Personal Ledger”)
* Short & full description (what it does: add income/expense, category totals, CSV export)
* Screenshots (minimum required; target all device families you selected)
* Keywords

**Age rating**

* Complete the questionnaire (financial tool with no user-generated public content → typically low rating).

**Pricing & availability**

* Choose **Free** (for coursework).
* Select **Markets** (choose **All markets** unless instructed otherwise).
* Visibility **Public**.

**Privacy policy**

* Provide a URL (e.g., a simple page stating *no data is collected or transmitted; all data remains local on device*).

Optional sections: **Cryptography** (if none, answer accordingly), **Optional add-ons** (not used).

**Step 7: Submit for certification**

* Review the **Submission overview** checklist; resolve any warnings.
* Click **Submit to the Store**.
* Status moves to **Certification in progress**; after passing, **Published**.

**Step 8: Verify the public listing and install from Store**

* Open app’s public Store page; copy the public URL for report.
* Install on a clean device to sanity-check:
  + Launch → Add Txn → Category total → Export CSV.
  + Confirm FinanceManagerApp.exe and all three DLLs function correctly.

**Quick pre-submission checklist**

* Packaging project **associated** with reserved Store app
* **Version** bumped in manifest (1.0.0.0 for first release). E.g., in FinanceManagerApp.csproj:

<PropertyGroup>

<AssemblyVersion>1.0.0.0</AssemblyVersion>

<FileVersion>1.0.0.0</FileVersion>

</PropertyGroup>

* **Icons/tiles** provided in Visual Assets
* **x86** architecture (consistent with current build)
* **.msixupload** generated (Store package)
* **WACK** passed
* Store submission: **Packages**, **Listings**, **Age rating**, **Pricing**, **Privacy policy** complete

**Appendix**

Source codes will be provided from this Google Drive (no EXE/MSI files uploaded, only project codes.

This drive has 3 folder/directories which contain full codes on:

* HelloWorld (as aligned with task 1.1)
* Calculator (as aligned with task 1.2)
* Finance (as aligned with task 1.3 and 1.4)

<https://drive.google.com/drive/folders/11KYaBIb43BPi3LX1a2RlT3iXQQ8Wkyrt?usp=sharing>