National Teachers College College Department

J Nepomuceno, Quiapo, Manila

TITLE OF THE ACTIVITY

Prelims Activity #2 in IT Elective 2 - Advanced Object-Oriented Programming and Robustness

Submitted by: James Lemit Onia

To be submitted to: Ms. Justin Louise Neypes

Date: February 21, 2026

CHOSEN SCENARIO:

Scenario D: The "Clean-Water" Treatment Plant

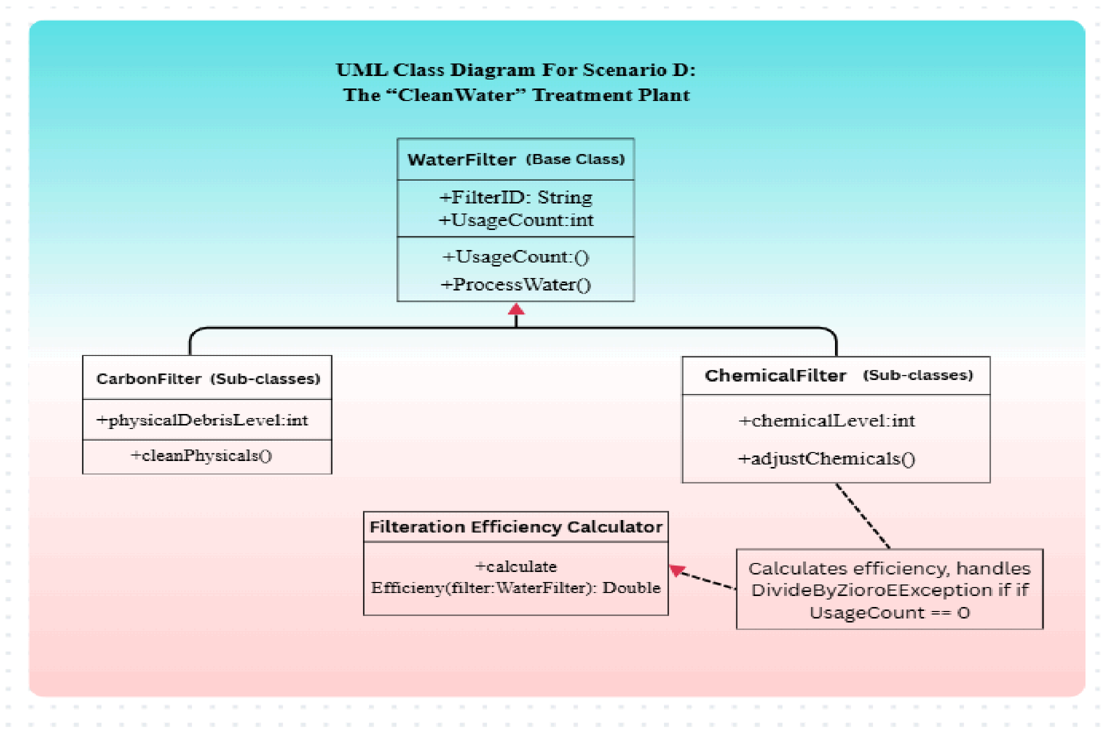
- **Base Class:** WaterFilter (Properties: FilterID, UsageCount).
- **Sub-Classes:** CarbonFilter and ChemicalFilter.
- **Logic:** An abstract method ProcessWater(). The ChemicalFilter must track chemical levels, while the Carbon tracks physical debris.
- **Exception:** Handle a DivideByZeroException when calculating the "Filtration Efficiency" if no water has passed through yet.

Step-by-Step Instructions

Phase 1: The Design Blueprint

1. **UML Class Diagram:** Create a visual diagram showing your Base Class, Sub-Classes, and any Interfaces. Identify which members are `private`, `protected`, or `public`.

UML Class Diagram for **Scenario D: The "Clean-Water" Treatment Plant**


UML Class Diagram For Scenario D:
The "CleanWater" Treatment Plant
WaterFilter (Base Class)
+FilterID: String
+UsageCount:int
+UsageCount:()
+ProcessWater()
CarbonFilter (Sub-classes)
+physicalDebrisLevel:int
+cleanPhysicals()
ChemicalFilter (Sub-classes)
+chemicalLevel:int
+adjustChemicals()
Filteration Efficiency Calculator
+calculate
Efficieny(filter:WaterFilter): Double
Calculates efficiency, handles DivideByZioroEException if if UsageCount == 0

So here are the explanation of the **UML Diagram** in Scenario D: The “Clean Water” Treatment Plant

- **WaterFilter (Base Class):** This is the "parent" class. It contains the common identity and usage tracking for all filters. The `ProcessWater()` method is abstract, meaning it provides a template that the sub-classes must fill in with their own logic.
- **Carbon & Chemical Filters (Sub-classes):** These are the "children." They inherit everything from `WaterFilter` but add their own specific responsibilities—Carbon handles physical debris, while Chemical handles chemical levels.
- **Filtration Efficiency Calculator:** This represents a utility or service class. It contains the logic to calculate performance and is specifically designed to handle the error that occurs if you try to divide by a `UsageCount` of zero.

2. **Logic Logic Flow:** Write a step-by-step description of how the pillars of OOP **are used** in your scenario (e.g., how the `CalculateRange` behaves differently for a bus vs. a van).

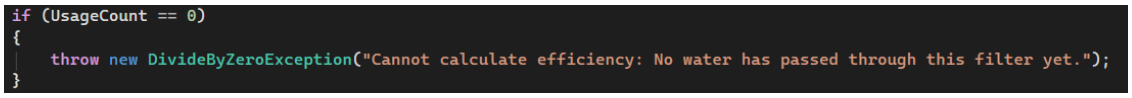
So here are the step-by-step description of how pillars of **OOP** are used in my chosen scenario which is the **Scenario D: The “Clean-Water” Treatment Plant**

1. **Encapsulation:** So let’s start with **encapsulation**, so I used **encapsulation** in the system to protect sensitive data by setting the `chemicalLevel` to **Private** (`-`). This prevents external classes from tampering with chemical balances directly. Access is only granted through controlled methods like `adjustChemicals()`, ensuring the internal state of the filter remains valid and safe.
2. **Inheritance:** So next is the **Inheritance,** So I used **Inheritance** to avoid redundant code, `CarbonFilter` and `ChemicalFilter` derive from the `WaterFilter` base class. This structure ensures that both specialized filters automatically possess the `FilterID` and `UsageCount` properties. It creates a "is-a" relationship where every specific filter is fundamentally a `WaterFilter`.
3. **Polymorphism:** So next is **Polymorphism**, So I used **Polymorphism** because Although both filters use the same method name, `ProcessWater()`, they behave differently based on their type. Because the first type is The **CarbonFilter** executes logic to trap **physical debris**. And next is the **ChemicalFilter** executes logic to neutralize **chemical levels**. This allows the plant to trigger the same command across a diverse list of filters while achieving different results.
4. **Abstraction:** And last but not the least is the **Abstraction**, So I used **Abstraction** in The system uses the `WaterFilter` **Abstract Class** to define a high-level "contract." It hides the complex internal mechanics of water purification and only exposes the essential method: `ProcessWater()`. This allows the plant to interact with any filter without needing to understand the specific science behind it.

3. **Exception Mapping:** List at least two specific scenarios in your code where a `try-catch` block is necessary to prevent a crash.

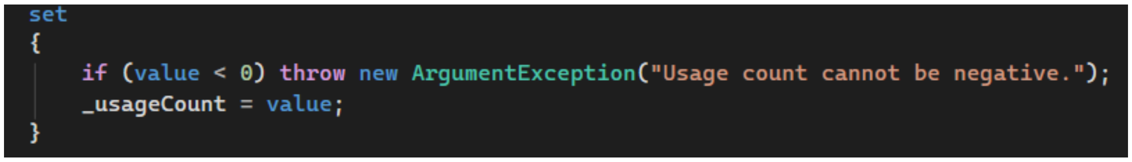
So here are the list of specific scenarios in my code where try-catch block is necessary to prevent a crash:

- So this is the **First Scenario**, The **"New Filter"** Calculation Error. The CalculateEfficieny() is inside of method of WaterFilter class:
- So in this Scenario, Without the try-catch in my Main method, the program would terminate instantly when the method is called. To catch it, the program stays open and tells the operator why it failed, and moves to the finally block safely.
- So here is the Code Logic used in this Scenario:



```
if (UsageCount == 0)
{
    throw new DivideByZeroException("Cannot calculate efficiency: No water has passed through this filter yet.");
}
```

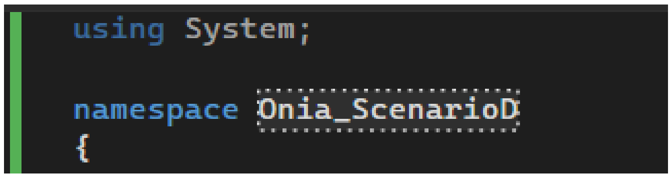
- So this is the **Next Scenario** which is The **"Negative Usage"** Data Error. The Code Logic that is used in this scenario is the set because Inside the set accessor for the UsageCount properly.
- So how it prevents a crash in this scenario, When this ArgumentException is thrown, the catch (Exception ex) blocks the Main method and intercept it. This prevents the "invalid state" from breaking the rest of the application's logic by allowing the system to log the error and shutdown gracefully.
- So here is the Code Logic used in this Scenario:



```
set
{
    if (value < 0) throw new ArgumentException("Usage count cannot be negative.");
    _usageCount = value;
}
```

Phase 2: The Implementation

1. **Project Setup:** Create a new C# Console Application. Use the naming convention: `Surname_ScenarioLabel.sln`.



```
using System;

namespace Onia_ScenarioD
{
```

2. **Encapsulation:** Ensure all class fields are `private`. Use Public Properties with logic inside the `set` accessor (e.g., `if (value < 0) throw new ArgumentException();`).

   Encapsulation: All fields (like _filterId and _usageCount) are **private**. The public properties use logic to prevent invalid data (like negative usage counts).

3. **Inheritance:** Use the `: base()` constructor syntax to pass data from sub-classes to the parent class.

   Inheritance: Both subclasses use the : base(id, usage) syntax to pass data up to the WaterFilter constructor, ensuring a centralized way to initialize IDs.

4. **Polymorphism:** Use the `virtual` keyword in the base class and the `override` keyword in sub-classes.

   Polymorphism: The ProcessWater() method is **abstract** in the parent and **overridden** in the children, allowing each filter to execute its specific cleaning logic.

5. **Robustness:** Wrap your "Main" execution logic in a `try-catch-finally` block. The `finally` block should print "System Shutdown" or "Session Ended."

   Robustness: The Main method is wrapped in a try-catch-finally block. This captures the DivideByZeroException when a filter's efficiency is calculated before it has been used, and the finally block ensures the shutdown message always appears.

Knowledge Check Questions

1. **Encapsulation:** Why is it dangerous to make the `CurrentTemp` or `BatteryPercentage` field `public`? How does a Property solve this?

It is dangerous because making fields like UsageCount or chemicalLevel public is dangerous because it allows external code to bypass the "rules" of the object. For example, any part of the program could set a filter's usage to a negative number or a chemical level to an unsafe value without the object knowing. In addition, this **Property** solves this by acting as a "gatekeeper." Using the set accessor, we can include validation logic (e.g., if (value < 0)). This ensures the internal state of the object remains valid and protected, forcing the rest of the application to interact with the data only on the object's terms.

2. **Inheritance:** What is the benefit of using a Base Class instead of creating two entirely separate classes from scratch?

The primary benefit of using a Base Class instead of creating two entirely separate classes from scratch are **Code Reusability** and **Consistency** because without a WaterFilter base class, we would have to rewrite the FilterID and UsageCount logic inside every single filter type (Carbon, Chemical, UV, etc.). Besides using inheritance, we write that code once in the parent class. If we ever need to add a new feature to *all* filters—such as a "Last Inspection Date"—we only have to add it to the base class, and every subclass will automatically receive it. This reduces bugs and makes the system much easier to maintain.

3. **Polymorphism:** In your code, how did the program "know" which version of the method to run (e.g., the Solar version vs. the Wind version) at runtime?

The program "know" which version of the method to run (e.g.., the Solar version vs. the Wind version at runtime is handled through **Dynamic Binding** (or Late Binding). In the code, the base class method is marked as abstract or virtual, and the subclasses use the override keyword.

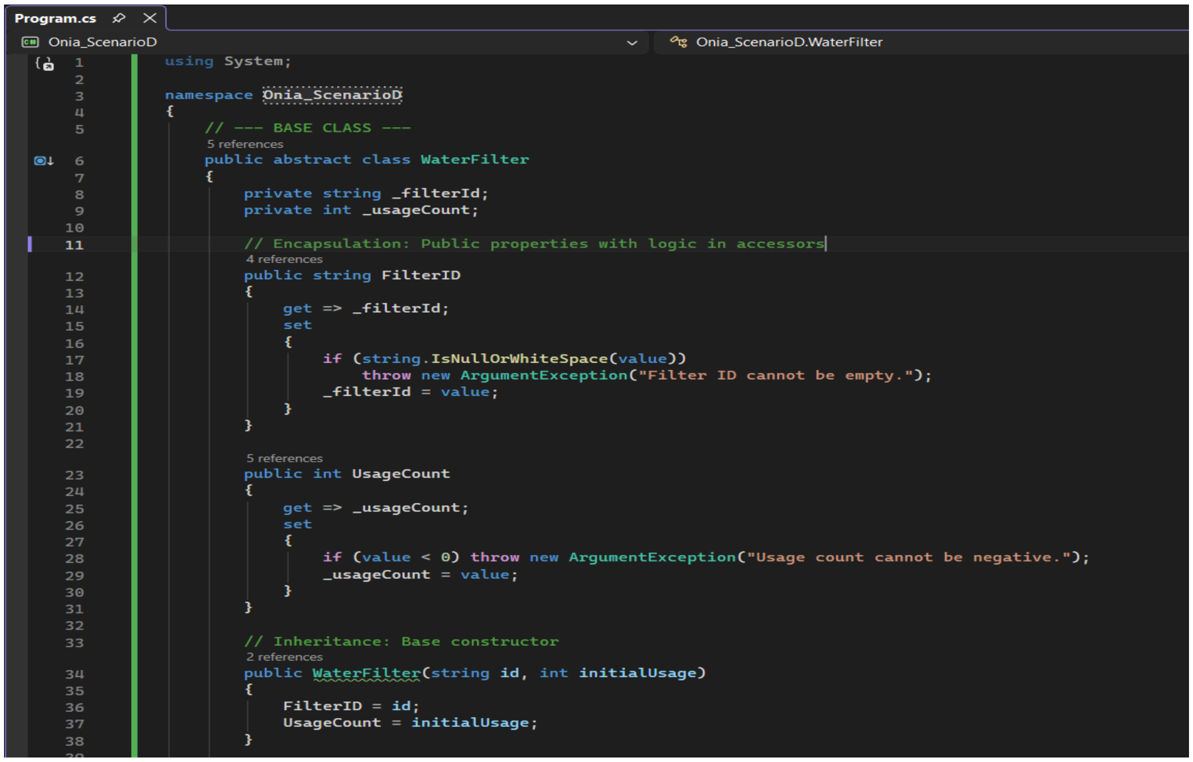
When you call ProcessWater(), the .NET Runtime checks the actual "type" of the object created in memory (the instance). Even if the variable is stored as a generic WaterFilter, the runtime sees that the object was instantiated as a ChemicalFilter and redirects the call to that specific version. This allows the program to be flexible and handle different behaviors using the same command.

4. **Robustness:** Explain the difference between `catch (Exception ex)` and a specific catch like `catch (ArgumentException ex)`. Which is a better practice?

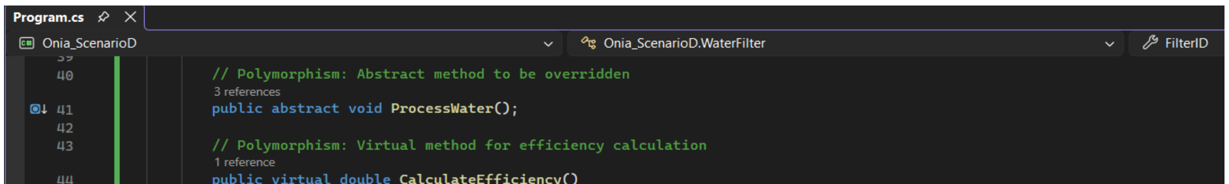
- The difference between catch (Exception ex) and a specific catch like catch (ArgumentException ex) is the catch (Exception ex) is a General Catch that can catches everything including unexpected system failures while catch (ArgumentException ex) is a Specific Catch that can handles only a specific types of errors like an invalid input.

- The better practice between those two is the catch (Exception ex) because of **three main reasons:** first is the **Precision** because it allows the program to provide accurate feedback. Second is **Debugging Efficiency** because If you catch everything as a generic error, you might accidentally mask a serious crash (like a memory failure) as a simple data entry mistake. And Lastly is the **Error Hierarchy** because you handle known issues gracefully while leaving the general Exception block as a final, last-resort safety net for the unexpected.

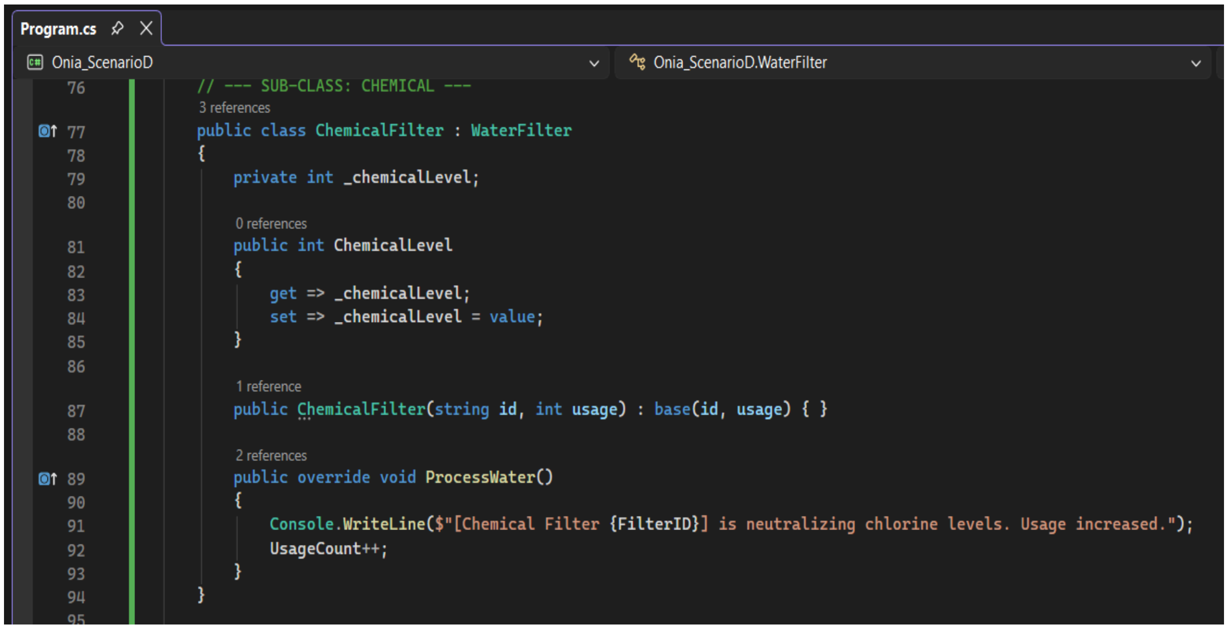
SCREENSHOTS OF CODES AND OUTPUT IN THE CHOSEN SCENARIO:


Program.cs
Onia_ScenarioD
Onia_ScenarioD.WaterFilter

```
using System;

namespace Onia_ScenarioD
{
    // --- BASE CLASS ---
    5 references
    public abstract class WaterFilter
    {
        private string _filterId;
        private int _usageCount;

        // Encapsulation: Public properties with logic in accessors
        4 references
        public string FilterID
        {
            get => _filterId;
            set
            {
                if (string.IsNullOrWhiteSpace(value))
                    throw new ArgumentException("Filter ID cannot be empty.");
                _filterId = value;
            }
        }

        5 references
        public int UsageCount
        {
            get => _usageCount;
            set
            {
                if (value < 0) throw new ArgumentException("Usage count cannot be negative.");
                _usageCount = value;
            }
        }

        // Inheritance: Base constructor
        2 references
        public WaterFilter(string id, int initialUsage)
        {
            FilterID = id;
            UsageCount = initialUsage;
        }
```


Program.cs
Onia_ScenarioD
Onia_ScenarioD.WaterFilter
FilterID

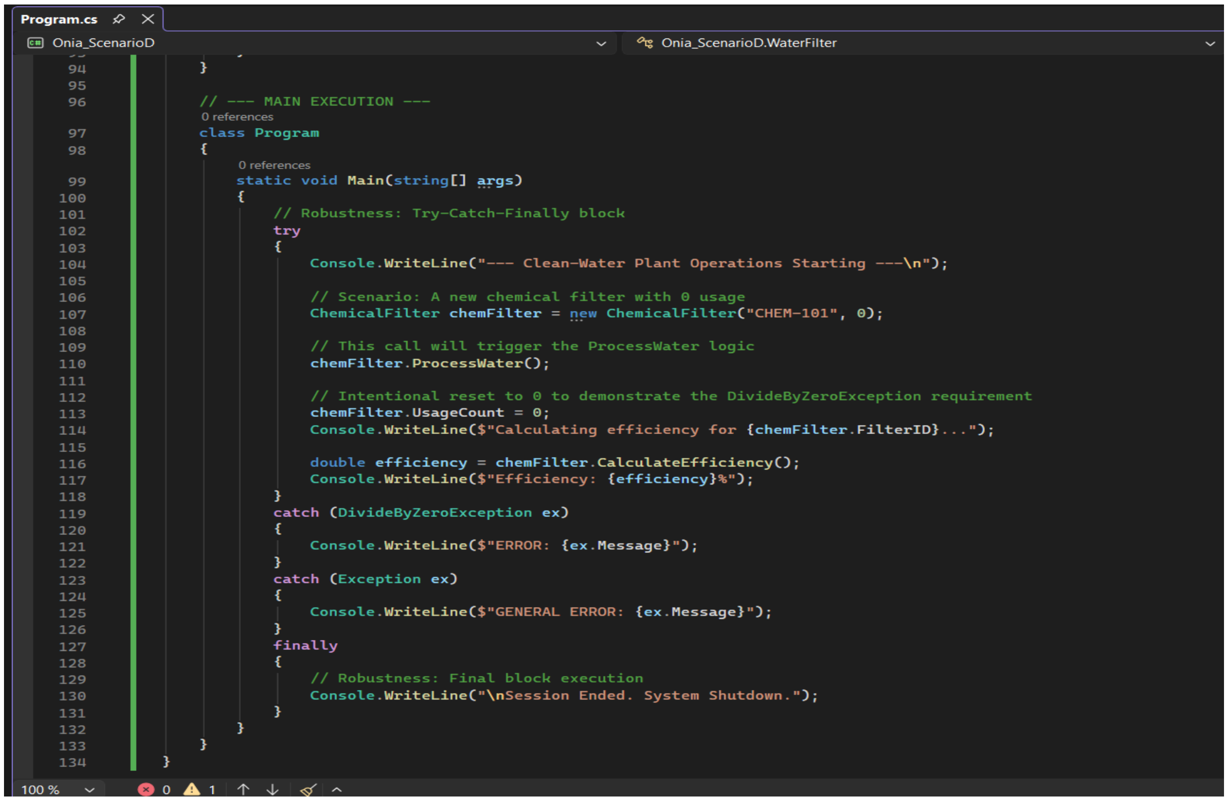
```
        // Polymorphism: Abstract method to be overridden
        3 references
        public abstract void ProcessWater();

        // Polymorphism: Virtual method for efficiency calculation
        1 reference
        public virtual double CalculateEfficiency()
```



Program.cs

Onia_ScenarioD | Onia_ScenarioD.WaterFilter

```
// --- SUB-CLASS: CHEMICAL ---
3 references
public class ChemicalFilter : WaterFilter
{
    private int _chemicalLevel;

    0 references
    public int ChemicalLevel
    {
        get => _chemicalLevel;
        set => _chemicalLevel = value;
    }

    1 reference
    public ChemicalFilter(string id, int usage) : base(id, usage) { }

    2 references
    public override void ProcessWater()
    {
        Console.WriteLine($"[Chemical Filter {FilterID}] is neutralizing chlorine levels. Usage increased.");
        UsageCount++;
    }
}
```



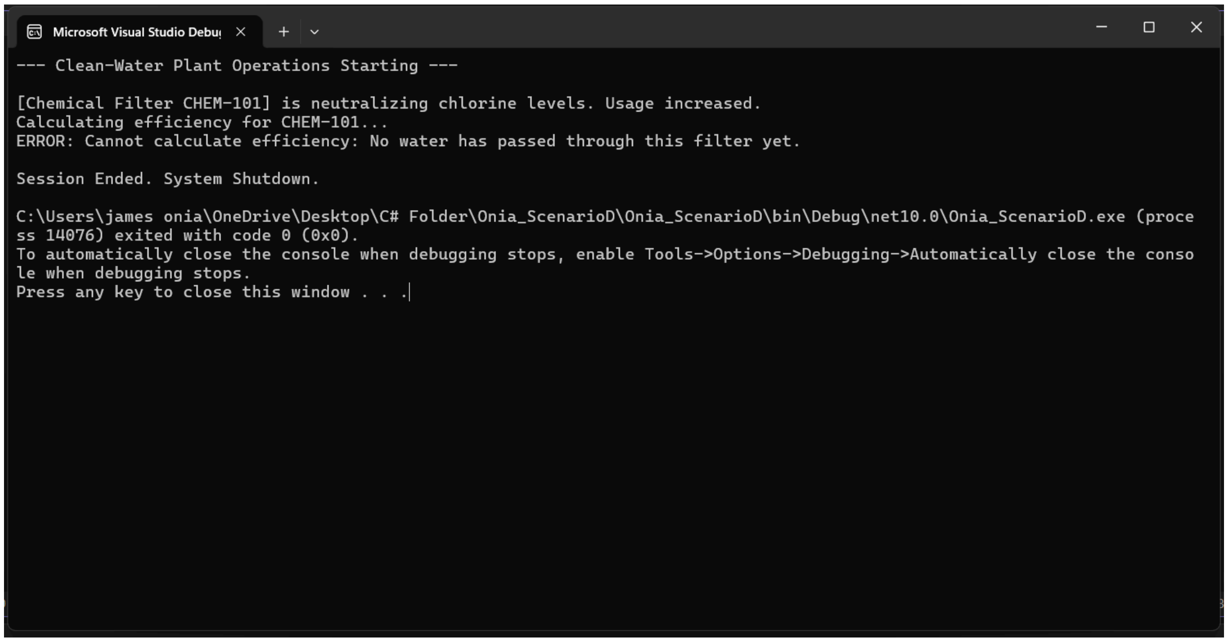
Program.cs

Onia_ScenarioD | Onia_ScenarioD.WaterFilter

```csharp
    }

    // --- MAIN EXECUTION ---
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            // Robustness: Try-Catch-Finally block
            try
            {
                Console.WriteLine("--- Clean-Water Plant Operations Starting ---\n");

                // Scenario: A new chemical filter with 0 usage
                ChemicalFilter chemFilter = new ChemicalFilter("CHEM-101", 0);

                // This call will trigger the ProcessWater logic
                chemFilter.ProcessWater();

                // Intentional reset to 0 to demonstrate the DivideByZeroException requirement
                chemFilter.UsageCount = 0;
                Console.WriteLine($"Calculating efficiency for {chemFilter.FilterID}...");

                double efficiency = chemFilter.CalculateEfficiency();
                Console.WriteLine($"Efficiency: {efficiency}%");
            }
            catch (DivideByZeroException ex)
            {
                Console.WriteLine($"ERROR: {ex.Message}");
            }
            catch (Exception ex)
            {
                Console.WriteLine($"GENERAL ERROR: {ex.Message}");
            }
            finally
            {
                // Robustness: Final block execution
                Console.WriteLine("\nSession Ended. System Shutdown.");
            }
        }
    }
}
```

100 % | 0 | 1

OUTPUT FOR THIS CODE:



Microsoft Visual Studio Debug

```
--- Clean-Water Plant Operations Starting ---

[Chemical Filter CHEM-101] is neutralizing chlorine levels. Usage increased.
Calculating efficiency for CHEM-101...
ERROR: Cannot calculate efficiency: No water has passed through this filter yet.

Session Ended. System Shutdown.

C:\Users\james onia\OneDrive\Desktop\C# Folder\Onia_ScenarioD\Onia_ScenarioD\bin\Debug\net10.0\Onia_ScenarioD.exe (proce
ss 14076) exited with code 0 (0x0).
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
Press any key to close this window . . .
```