



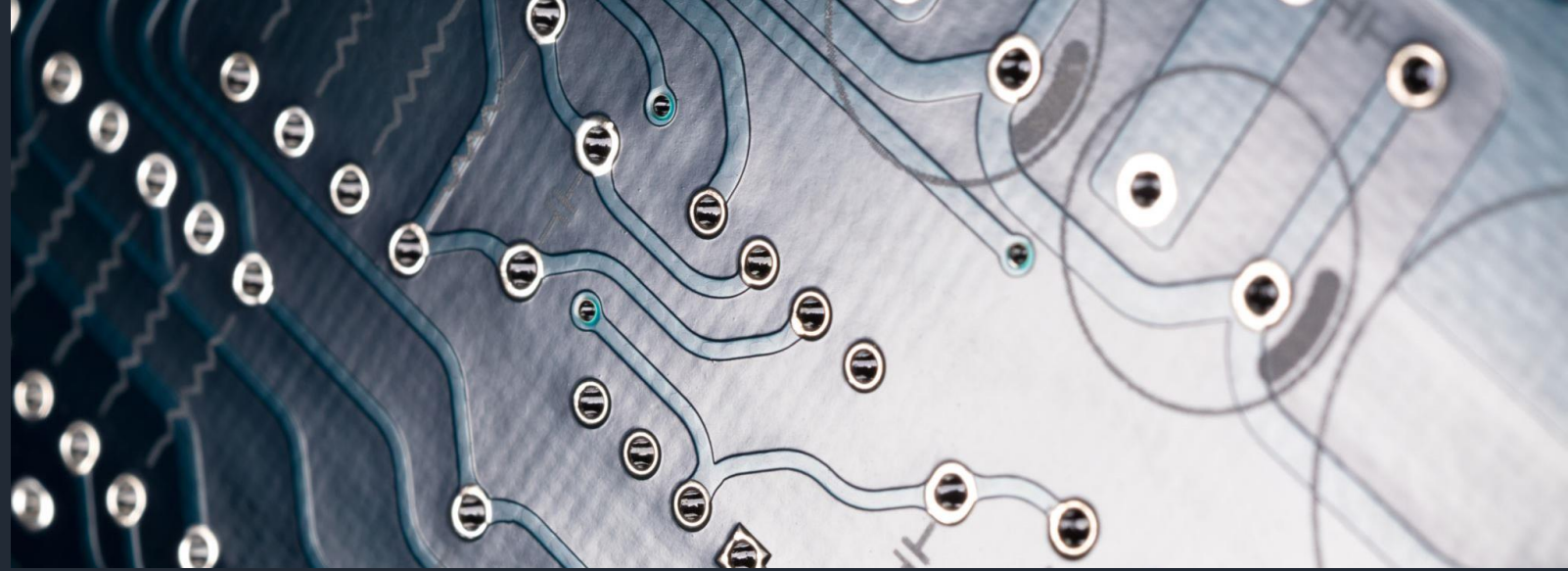
# Session 3: Unit Testing in MVVM - A Deep Dive into the Model (Part 1)

Leveraging the Arrange, Act, and Assert (AAA) Methodology

# Introduction to Unit Testing & TDD using LINQ and LINQPad.







# The Need for Unit Testing

✖ UserSentimentAnalysis.Tests (17)	83 ms	
✖ UserSentimentAnalysis.Tests (17)	83 ms	
✔ EmojiTests (5)	6 ms	
✔ EmojiClothingTest	5 ms	
✔ EmojiExtraSpecialCharatersTest	< 1 ms	
✔ EmojiFaceSearchTest	1 ms	
✔ EmojiHeartsTest	< 1 ms	
✔ TearsOfJoyTest	< 1 ms	
ⓘ HomeControllerTests (6)		
✖ TwitterDataModelTests (6)	77 ms	
✖ ActiveCognitiveServiceTest	74 ms	Assert.AreEqual failed. Expected...
✖ AverageTweetTest	3 ms	Test method UserSentimentAna...
✖ GetTweetSentimentTest	< 1 ms	Assert.AreEqual failed. Expected...
✖ HistoricalTweetTest	< 1 ms	Assert.AreEqual failed. Expected...
✔ MultipleAverageTweetTest	< 1 ms	
✖ ParseTweetTest	< 1 ms	Assert.AreEqual failed. Expected...
Group Summary		
TwitterDataModelTests		
Tests in group: 6		
⌚ Total Duration: 77 ms		
Outcomes		
✔ 1 Passed		

- 1. **Definition:** Unit Testing is the process of testing individual units or components of software to determine if they work as intended.
- 2. **Catch Bugs Early:** Unit tests help identify bugs and issues at the early stages of development, ensuring that individual components are correctly implemented.
- 3. **Visual:** Consider displaying a simple graphic showing the software development process, highlighting where unit testing fits in and emphasizing its role in the early detection of defects. (Integrated in modern IDE's)



## Benefits of Unit Testing

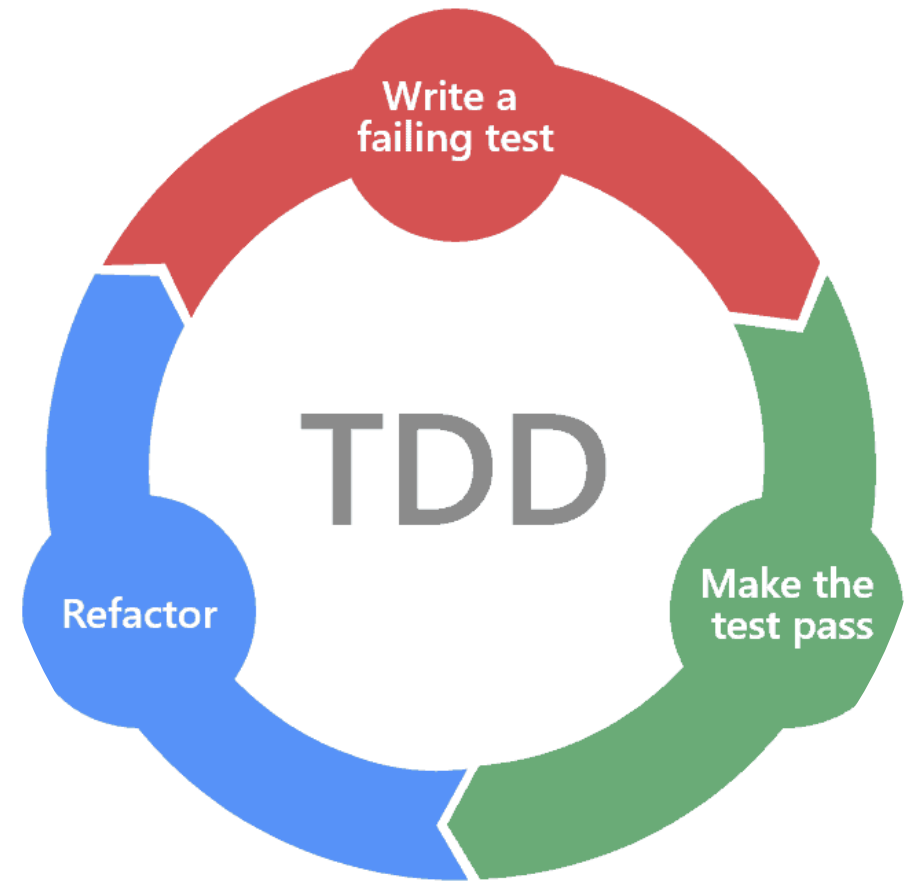
- 1. Improved Code Quality:** Regular unit testing ensures code modifications do not introduce new bugs, leading to higher quality software.
- 2. Facilitates Changes & Refactoring:** With a robust suite of unit tests, developers can confidently make changes to the codebase, knowing that regressions will be caught.
- 3. Documentation:** Unit tests can serve as documentation. New developers can understand the functionality of a module or component by reading its tests.

# Introduction to Test Driven Development (TDD)

**1. Definition:** TDD is a software development approach in which tests are written before the actual code. It follows a short and repetitive cycle: Write the test, run it (which should fail), write code to pass it and refactor it if necessary.

## 2. Rethinking Code with TDD

- **Traditional Coding:** Write code → Test it
- **TDD Approach:** ✦ Test First ✦ → Write code to pass the test



# TDD as a Mindset

---

## 1. Proactive vs. Reactive:

Traditional testing is reactive – we find bugs after introducing them. TDD is proactive – we're preventing bugs from occurring in the first place.

## 2. Writing for Testability: TDD forces developers to write modular and maintainable code because such code is more testable.

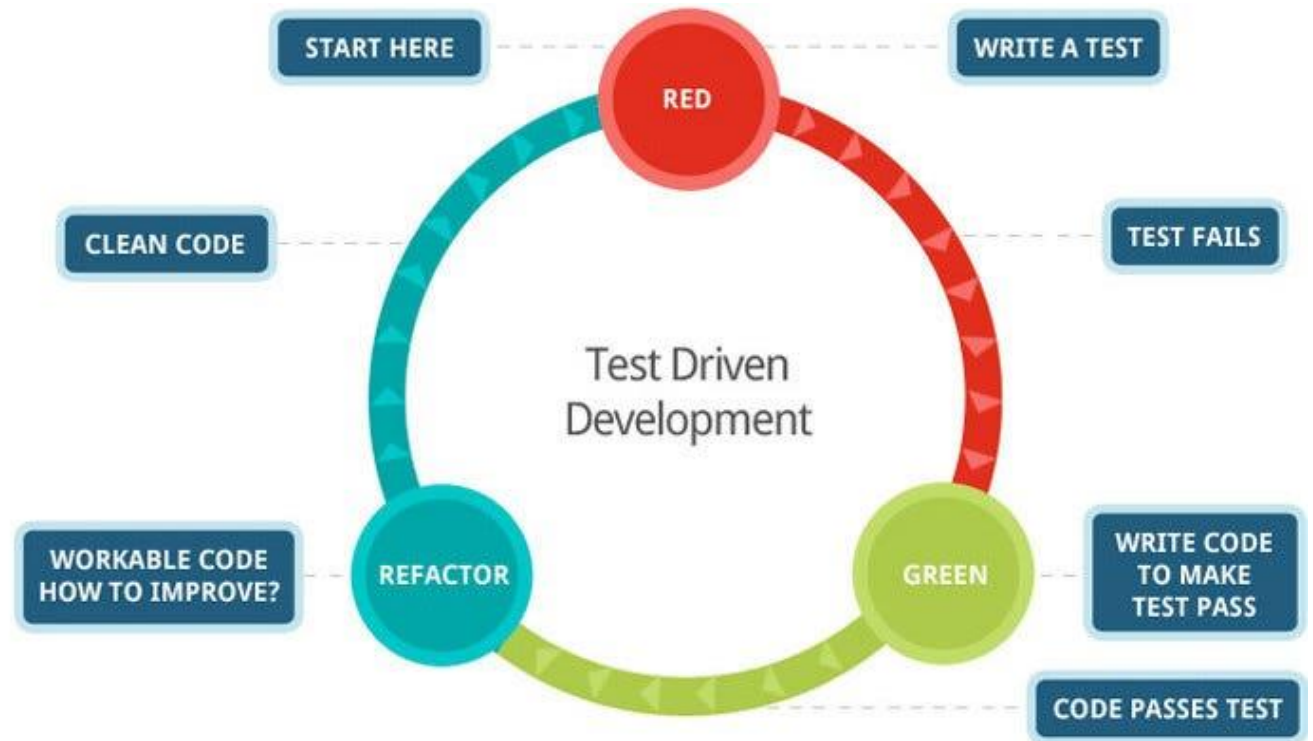
Traditional testing	Test-driven development
A test-last approach wherein the developer creates the code but leaves testing until the end of the development process.	A test-first approach wherein the developer or test automation engineer first creates the test, and then the developer codes to meet the test's requirements.
Focuses on code correctness but may not detect all coding defects.	Tests then refactors until code passes the test, continues until code meets functionality, and reduces the number of bugs in the system.
Linear process. (Design-code-test)	Cyclical process. (Test-code-refactor)

# TDD as a Process

## 1. Red-Green-Refactor: Three main steps of TDD.

1. **Red:** Write a failing test.
2. **Green:** Write minimal code to make the test pass.
3. **Refactor:** Improve the code without changing its behavior.

## 2. Immediate Feedback: TDD provides immediate feedback, which means errors are caught early, and code quality is consistently maintained.



# Using “Use Case Scenario” for TDD



- **Use Case 1: Basic Course Purchase Item**
- **Goal in Context:** Purchase an item from the vending machine.
- **Actors:** User
- **Trigger:** User inserts coins or selects an item.
- **Preconditions:**
  - The vending machine is operational.
- **Postconditions on Success:**
  - Item is provided.
- **Postconditions on Failure:**
  - Item is not provided.
- **Description:** (Part of the Arrange, Act, Assert mention later in the slides)
  - User inserts a sufficient amount of money. (Arrange)
  - User selects the item on the panel. (Act)
  - Selected item and returned money are dispensed. (Assert)
  - User takes the item and any returned money.
- **Alternative Courses:**
- **Insufficient Amount:**
  - User inserts more cash or aborts (Errors).
- **Selected Item Not Available:**
  - User selects a different item or aborts (Errors).
- **Exceptions:**
- Item should be dispensed no more than 3 seconds after selection.
- If no user selection within 3 seconds, the use case is aborted and inserted cash is returned.





## Unit Testing: A Time-Saving Investment?

- **Initially:** No, writing tests takes time upfront.
- **Long-Term Benefits:** Prevents elusive bugs, saving stress and hours later.

**Think About It:** Remember math problems? Spotting an early arithmetic error vs. discovering it after several steps? Unit tests help catch those "early errors" in coding!

# Mathematical Example

## God Loves Us

- **Create a method for dividing 1 number from another.**
- **Entry Point - Main Method:**
  - Declares a decimal variable num1 and initializes it with the value 6.
  - Declares a decimal variable num2 and initializes it with the value 5.
  - Calls the Div function with num1 and num2 as arguments and assigns the result to the result variable.
  - Outputs the value of the result using the Dump method.
- **Div Method:**
  - Takes in two parameters of type decimal: a and b.
  - Returns the result of dividing a by b.

**NOTE: “God Loves Us” is a high-level belief in programming that nothing can go wrong. Data is always well-formatted, the internet works, users understand how our application works, and ALWAYS click items on the screen in the order of operations we envisioned.**

```
void Main()  
{  
    decimal num1 = 6;  
    decimal num2 = 5;  
    var result = Div(num1, num2);  
    result.Dump();  
}
```

```
public decimal Div(decimal a, decimal b)  
{  
    return a / b;  
}
```

Results λ SQL IL+Native Tree AI

1.2

# Mathematical Example Reality

- Create a method for dividing 1 number from another.
- **Entry Point - Main Method:**
  - Declares a decimal variable num1 and initializes it with the value 6.
  - Declares a decimal variable num2 and initializes it with the value 0.
  - Calls the Div function with num1 and num2 as arguments and assigns the result to the result variable.
  - Outputs the value of the result using the Dump method.
- **Div Method:**
  - Takes in two parameters of type decimal: a and b.
  - Returns the result of dividing a by b.

**NOTE: “Reality” is the result of programming in the real world. You must always code defensively.**

```
DivideByZeroException: Attempted to divide by zero.
```

```
decimal num1 = 6;  
decimal num2 = 0;  
var result = Div(num1, num2);  
result.Dump();  
}
```

```
public decimal Div(decimal a, decimal b)  
{  
    return a / b;  
}
```

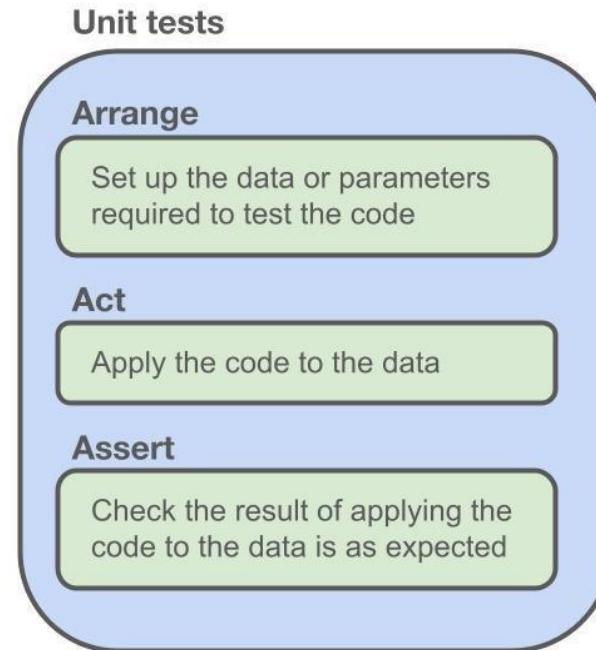
Results | SQL | IL+Native | Tree | AI

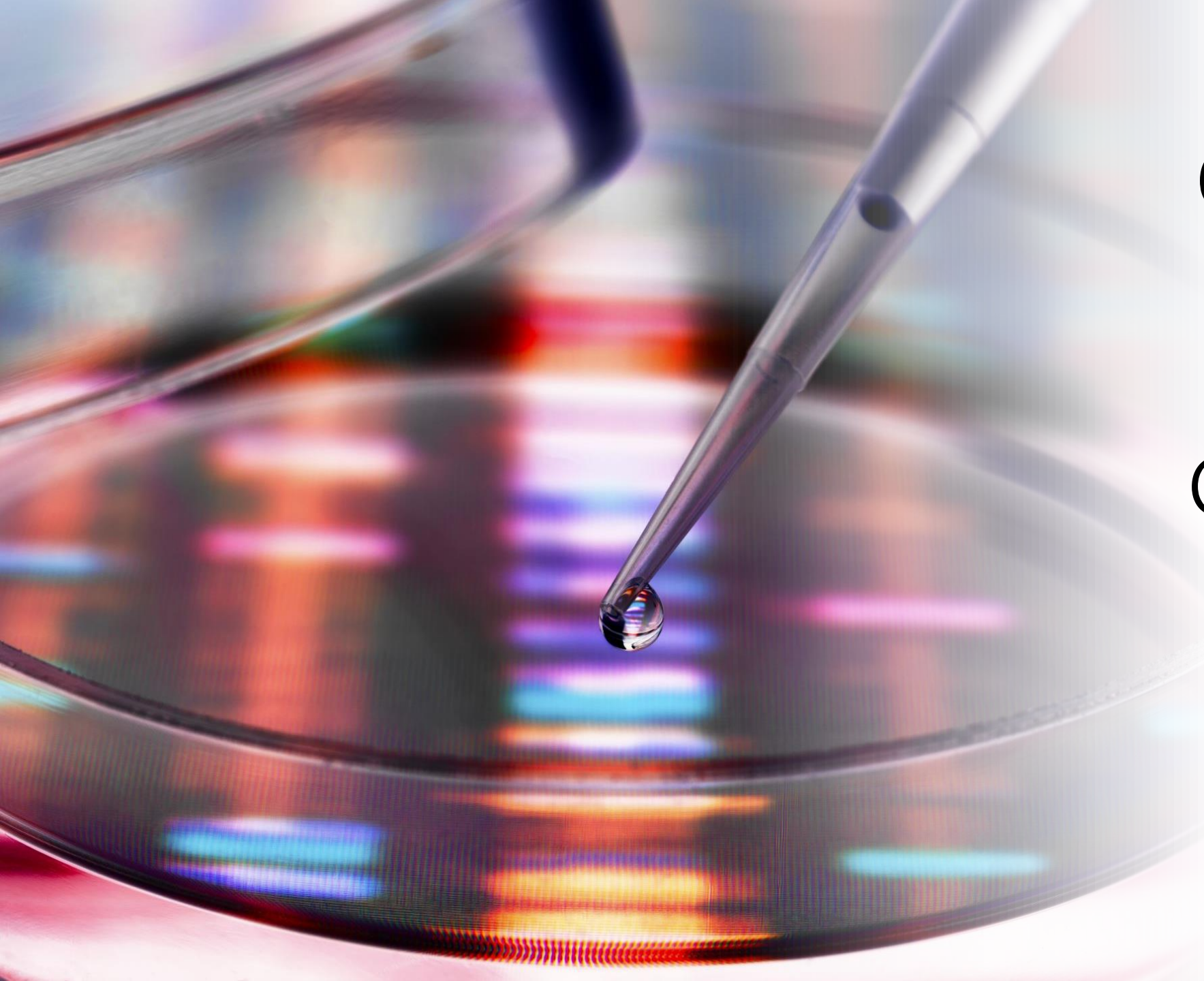
▼ DivideByZeroException ...

Attempted to divide by zero.

# The AAA Pattern - An Overview

- 1. Definition:** The AAA pattern is a common way of writing unit tests. It stands for Arrange, Act, and Assert.
- 2. Purpose:** Its structured format ensures tests are clear and maintainable.





Creating Unit  
Test

God Loves US





## AAA Pattern - Arrange

**Arrange:** This is where you set up the test, prepare any inputs, and set initial conditions. Think of it as "setting the stage" for the test.

- **Objective:**
  - Create a controlled and predictable environment for the unit test.
- **Isolation:**
  - The Arrange phase should ensure the test is isolated from external factors. Tests should be repeatable under the same conditions.
- **Clarity:**
  - The setup should be as simple and clear as possible. A fellow developer (or even our future self!) should be able to understand the setup without extensive documentation.
- **Minimalism:**
  - Only set up what is absolutely necessary for the test. Avoid over-complicating the arrangement with unnecessary objects or data.

# Refactor Code for Arrange

- This is where we set up the initial data to be pass into our method.
  1. Create a new method where our testing will be done called Test\_Math\_Division\_IsValid()
  2. Create a comment to show the "Arrange" area of our unit test.

```
void Main()  
{  
    Test_Math_Division_IsValid();  
}  
  
// validate that the division function  
// returns a correct value  
public void Test_Math_Division_IsValid()  
{  
    // Arrange  
    // Initialize any variable for our test  
    decimal num1 = 10;  
    decimal num2 = 2;  
}  
  
// Math method for return a number that when 2 numbers  
// are divide by each other.  
public decimal Div(decimal num1, decimal num2)  
{  
    return num1 / num2;  
}
```



## AAA Pattern - Act

**Act:** You invoke the method or function you want to test here. It's the main action of the play after the stage is set.

- **Objective:**
  - Execute the functionality under test to produce a result or side effect.
- **Simplicity:**
  - The Act phase should ideally involve only the call to the method/function under test. Avoid including additional logic or calls.
- **Single Responsibility:**
  - Ideally, the method/function under test should have a single responsibility. This ensures that the test is focused and clear..
- **No Assertions:**
  - Avoid placing assertions or checks in the "Act" phase. Those belong in the "Assert" phase to maintain a clear separation of concerns.

# Refactor Code for Act

- We need to make a call to our “Div” method
  1. Create a comment to show the “Act”
  2. Create a variable to store our result and call our “Div” method. We will call it “**actual**” as this is the actual result return.

```
void Main()
{
    Test_Math_Division_IsValid();
}

// validate that the division function
// returns a correct value
public void Test_Math_Division_IsValid()
{
    // Arrange
    // Initialize any variable for our test
    decimal num1 = 10;
    decimal num2 = 2;

    // Act
    // Only the call to the method/function you are testing.
    // Rename result to actual for comparing
    var actual = Div(num1, num2);
}

// Math method for return a number that when 2 numbers
// are divide by each other.
public decimal Div(decimal num1, decimal num2)
{
    return num1 / num2;
}
```



## AAA Pattern - Act

**Assert:** This is where you check the result of the "Act" to ensure it worked as expected. It's the conclusion or the outcome you're verifying

- **Objective:**
  - Verify that the outcome of the "Act" phase matches the expected results or behavior.
- **Focused Tests:**
  - Each test should assert a single "logical unit of work" to ensure tests are clear and maintainable. Avoid multiple unrelated assertions in a single test.
- **Avoid External Dependencies:**
  - Ideally, the method/function under test should have a single responsibility. This ensures that the test is focused and clear.
- **Immediate Feedback:**
  - Tests should be designed to fail fast, providing immediate feedback if something is amiss.



## Refactor Code for Assert (Fail)

- We need to compare the **expected result** to the **actual result**
  1. Setup the expected result. NOTE: On the first pass, we ensure that the result fails so that when we refactor it with the actual result, we get a **"PASS"**
  2. We create a message that we display to the user about the results of the test.

```
var actual = Div(num1, num2);

// Assert
// Verify that the outcome of the "Act" phase matches
// the expected result or behaviour.
// NOTE: We set the expected result to 7 to force
// a failure.
decimal expected = 7;

// Validate if we got the expected results.
string isValid = actual == expected ? "Pass" : "Fail";

// Display the result to the user
Console.WriteLine($"-- {isValid} -- Test_Math_Division_IsValid: Expected {expected} vs Actual {actual}");
```

# Refactor Code for Assert (Fail)

- Run all Unit Test(s)

```
// Only the call to the method/function your are testing
// Rename result to actual for comparing.
var actual = Div(num1, num2);

// Assert
// Verify that the outcome of the "Act" phase matches
// the expected results or behavior
// NOTE: We set the expected result to 7 to force a
// failure.
decimal expected = 7;

// Validate if we got the expected results
string isValid = actual == expected ? "Pass" : "Fail";

// Display the results to the user
Console.WriteLine($"-- {isValid} --| Test_Math_Division_Valid: Expected {expected} vs Actual {actual}");
```

-- Fail -- Test\_Math\_Division\_Valid: Expected 7 vs Actual 5

# Refactor Code for Assert (Pass)

- Update the **expected result** so that the Unit Test will **"Pass"**
- Re-run the Unit Test(s)

```
// Only the call to the method/function your are testing
// Rename result to actual for comparing.
var actual = Div(num1, num2);

// Assert
// Verify that the outcome of the "Act" phase matches
// the expected results or behavior
// NOTE: We set the expected result to 7 to force a
// failure.
decimal expected = 5;

// Validate if we got the expected results
string isValid = actual == expected ? "Pass" : "Fail";

// Display the results to the user
Console.WriteLine($"-- {isValid} -- Test_Math_Division_Valid: Expected {expected} vs Actual {actual}");
```

```
-- Pass -- Test_Math_Division_Valid: Expected 5 vs Actual 5
```

# Creating Unit Test

## Reality

---





# Error Handling

- **We will focus on three types of error handling:**
- **AggregateException:** Used to wrap multiple exceptions into a single exception, commonly encountered in parallel programming. Also, when returning a list of exceptions to the user (i.e., Missing first and last name).
- **ArgumentNullException:** Thrown when a method receives a **null** argument that it doesn't expect or allow.
- **Exception:** The base class for all exceptions in C#, acting as a general catch-all for unhandled exceptions.



# Improved Structure For Crafting Methods With Exception Handling.

## 1. Method Execution Block:

- **public void MethodName(...)** { ... } is the entry point for your code (Client Side).
- Within the method, there's a **try** block that aims to execute the main logic of the script safely, catching and handling any potential exceptions that arise.

## 2. Driver Area:

- This is where the main logic of the method resides.
- It is named "DRIVER area" to indicate the part of the code where specific functionalities are "driven" or tested.

## 3. Exception Handling Block:

- Following the main logic is a **catch** section that aims to capture and handle exceptions that might arise during execution.
- There are three specific **catch** blocks:
  - **AggregateException**: Catches exceptions that encapsulate multiple exceptions, iterating through each of them.
  - **ArgumentNullException**: Catches exceptions related to null arguments.
  - **Exception**: General catch-all for any other exceptions not specifically caught above.

## 4. Utility Method:

1. **private Exception GetInnerException(Exception ex)**: A helper method used in the exception handling block to retrieve the most inner exception, which could provide more context about the root cause of an error.

# Improved Structure For Crafting Methods With Exception Handling - Example

```
void Main()  
{  
    try  
    {  
        // This is the DRIVER area.  
    }  
  
    #region catch all exceptions  
    catch (AggregateException ex)  
    {  
        foreach (var error in ex.InnerExceptions)  
        {  
            error.Message.Dump();  
        }  
    }  
    catch (ArgumentNullException ex)  
    {  
        GetInnerException(ex).Message.Dump();  
    }  
    catch (Exception ex)  
    {  
        GetInnerException(ex).Message.Dump();  
    }  
    #endregion  
}  
  
private Exception GetInnerException(Exception ex)  
{  
    while (ex.InnerException != null)  
        ex = ex.InnerException;  
    return ex;  
}
```

1) Method Execution Block

2) Driver Area

3) Exception Handling Block

4) Utility Method

# AggregateException

An **AggregateException** is special because it wraps multiple exceptions into a single exception. It's commonly encountered in parallel programming and tasks because multiple threads or tasks can throw exceptions simultaneously. By wrapping them in an **AggregateException**, you can handle all these exceptions in a unified manner. Also, when returning a list of exceptions to the user (i.e. Missing first & last name).

**Explanation:** In this snippet, the code catches an **AggregateException** and then loops through its **InnerExceptions** property. This property is a collection of all the exceptions wrapped by the **AggregateException**.

The **error.Message.Dump()** outputs the message of each exception.

```
csharp Copy code  
  
catch (AggregateException ex)  
{  
    foreach (var error in ex.InnerExceptions)  
    {  
        error.Message.Dump();  
    }  
}
```

## AggregateException Example

```
void Main()
{
    try
    {
        // passing in an empty first and last name
        AggregateExceptionTest("", "");
    }

    #region catch all exceptions
    catch (AggregateException ex)
    {
        foreach (var error in ex.InnerExceptions)
        {
            error.Message.Dump();
        }
    }
    catch (ArgumentNullException ex)
    {
        GetInnerException(ex).Message.Dump();
    }
    catch (Exception ex)
    {
        GetInnerException(ex).Message.Dump();
    }
    #endregion
}

private Exception GetInnerException(Exception ex)
{
    while (ex.InnerException != null)
        ex = ex.InnerException;
    return ex;
}
```

## AggregateException Example

```
public void AggregateExceptionTest(string firstName, string lastName)
{
    #region Business Logic and Parameter Exceptions
    // create a List<Exception> to contain all discovered errors
    List<Exception> errorList = new List<Exception>();
    // Business Rules
    //     rule:  first name cannot be empty or null
    //     rule:  last name cannot be empty or null

    // parameter validation
    if (string.IsNullOrWhiteSpace(firstName))
    {
        errorList.Add(new Exception("First name is required and cannot be empty"));
    }

    if (string.IsNullOrWhiteSpace(lastName))
    {
        errorList.Add(new Exception("Last name is required and cannot be empty"));
    }
    #endregion

    /*
     * actual code for method.
     */

    if (errorList.Count() > 0)
    {
        // throw the list of business processing error(s)
        throw new AggregateException("Unable to proceed! Check concerns", errorList);
    }
}
```

Add errors to error list

Throw an exception if there are errors



## AggregateException Example

```
9  #region catch all exceptions
10 catch (AggregateException ex)
11 {
12     foreach (var error in ex.InnerExceptions)
13     {
14         error.Message.Dump();
15     }
16 }
17 catch (ArgumentNullException ex)
18 {
19     GetInnerException(ex).Message.Dump();
20 }
21 catch (Exception ex)
22 {
23     GetInnerException(ex).Message.Dump();
24 }
25 #endregion
26 }
```


First name is required and cannot be empty  
Last name is required and cannot be empty

# ArgumentNullException

The **ArgumentNullException** is thrown when a method is invoked, and one of the arguments passed to the method is **null**, but the method does not allow it.

**Explanation:** If an argument passed to a method is **null** where it shouldn't be, this block will catch that exception. It then retrieves any inner exceptions (if they exist) using the **GetInnerException** method and displays the message.

csharp

 Copy code


```
catch (ArgumentNullException ex)
{
    GetInnerException(ex).Message.Dump();
}
```

# Methods: GetInnerException

The provided **GetInnerException** method is a utility that retrieves the innermost exception (if any) from the given exception. This can be useful because sometimes exceptions wrap other exceptions, providing more specific context or details about the root cause of the error.

**Explanation:** The method keeps looping if there is an inner exception, updating the **ex** variable each time. Once there is no more inner exception, it returns the deepest exception it found.

csharp

 Copy code

```
private Exception GetInnerException(Exception ex)
{
    while (ex.InnerException != null)
        ex = ex.InnerException;
    return ex;
}
```

## ArgumentNullException Example

```
void Main()
{
    try
    {
        // passing an track ID larger than max TrackID
        ArgumentNullExceptionTest(10000);
    }

    #region catch all exceptions
    catch (AggregateException ex)
    {
        foreach (var error in ex.InnerExceptions)
        {
            error.Message.Dump();
        }
    }
    catch (ArgumentNullException ex)
    {
        GetInnerException(ex).Message.Dump();
    }
    catch (Exception ex)
    {
        GetInnerException(ex).Message.Dump();
    }
    #endregion
}

private Exception GetInnerException(Exception ex)
{
    while (ex.InnerException != null)
        ex = ex.InnerException;
    return ex;
}
```

## ArgumentNullException Example

```
public void ArgumentNullExceptionTest(int trackID)
{
    #region Business Logic and Parameter Exceptions
    // create a List<Exception> to contain all discovered errors
    List<Exception> errorList = new List<Exception>();
    // Business Rules
    //     rule:  Track must exist in the database

    // parameter validation
    var track = Tracks
        .Where(x => x.TrackId == trackID)
        .Select(x => x).FirstOrDefault();

    if (track == null)
    {
        throw new ArgumentNullException($"No track was found for Track ID: {trackID}");
    }
    #endregion

    /*
     * actual code for method.
     */

    if (errorList.Count() > 0)
    {
        // throw the list of business processing error(s)
        throw new AggregateException("Unable to proceed! Check concerns", errorList);
    }
}
```

Check if the track object is null

## ArgumentNullException Example

```
8
9 #region catch all exceptions
10 catch (AggregateException ex)
11 {
12     foreach (var error in ex.InnerExceptions)
13     {
14         error.Message.Dump();
15     }
16 }
17 catch (ArgumentNullException ex)
18 {
19     GetInnerException(ex).Message.Dump();
20 }
21 catch (Exception ex)
22 {
23     GetInnerException(ex).Message.Dump();
24 }
25 #endregion
```


Value cannot be null. (Parameter 'No track was found for Track ID: 10000')

# Exception

The **Exception** class is the base class for all exceptions in C#. If an exception is thrown that is not caught by the more specific catch blocks above (like **AggregateException** or **ArgumentNullException**), this catch block will handle it

**Explanation:** This block acts as a safety net. It will catch any exceptions that are not caught by the preceding catch blocks. Like the **ArgumentNullException** block, it retrieves any inner exceptions and displays the message.

csharp

 Copy code

```
catch (Exception ex)
{
    GetInnerException(ex).Message.Dump();
}
```



# Exception Example

```
void Main()
{
    try
    {
        // passing an invalid track ID (less than 1)
        ExceptionTest(0);
    }

    #region catch all exceptions
    catch (AggregateException ex)
    {
        foreach (var error in ex.InnerExceptions)
        {
            error.Message.Dump();
        }
    }
    catch (ArgumentNullException ex)
    {
        GetInnerException(ex).Message.Dump();
    }
    catch (Exception ex)
    {
        GetInnerException(ex).Message.Dump();
    }
    #endregion
}

private Exception GetInnerException(Exception ex)
{
    while (ex.InnerException != null)
        ex = ex.InnerException;
    return ex;
}
```

# Exception Example

```
public void ExceptionTest(int trackID)
{
    #region Business Logic and Parameter Exceptions
    // create a List<Exception> to contain all discovered errors
    List<Exception> errorList = new List<Exception>();
    // Business Rules
    //     rule:  Track ID must be valid

    // parameter validation
    // This is a show stopper and no reason to go beyond this
    //     point of code!!!
    if (trackID < 1 )
    {
        throw new Exception("TrackID is invalid");
    }
    #endregion

    /*
     actual code for method.
    */

    if (errorList.Count() > 0)
    {
        // throw the list of business processing error(s)
        throw new AggregateException("Unable to proceed! Check concerns", errorList);
    }
}
```

Check if the track ID is invalid

## Exception Example

```
9  #region catch all exceptions
10 catch (AggregateException ex)
11 {
12     foreach (var error in ex.InnerExceptions)
13     {
14         error.Message.Dump();
15     }
16 }
17 catch (ArgumentNullException ex)
18 {
19     GetInnerException(ex).Message.Dump();
20 }
21 catch (Exception ex)
22 {
23     GetInnerException(ex).Message.Dump();
24 }
25 #endregion
26 }
```

Results λ SQL IL+Native Tree AI

TrackID is invalid

## Statement Regarding Slide Accuracy and Potential Revisions

- Please note that the content of these PowerPoint slides is accurate to the best of my knowledge at the time of presentation. However, as new research and developments emerge, or to enhance the learning experience, these slides may be subject to updates or revisions in the future. I encourage you to stay engaged with the course materials and any announcements for the most current information