Unit Testing

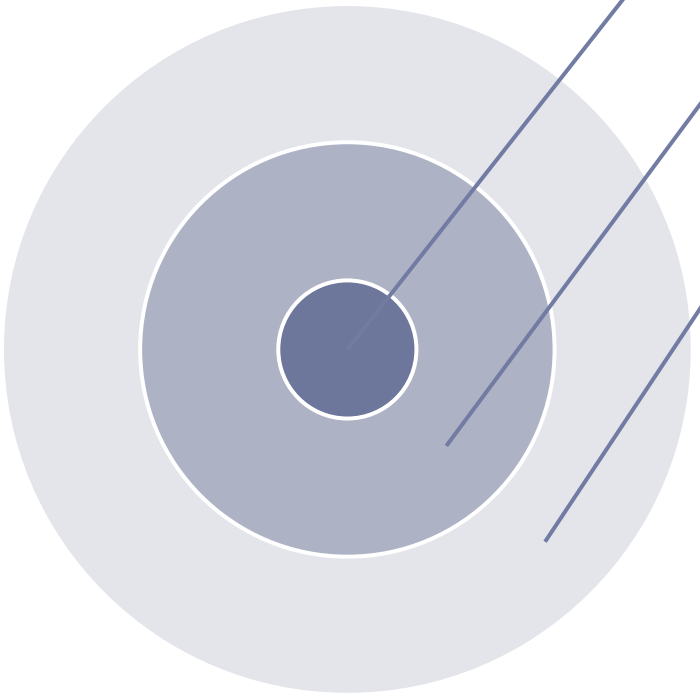
Concepts, Tools and Best Practices

"I don't have time to write tests because I am too busy debugging."


Take 20 seconds to debug yourself.
A reminder from Apple Wellness


ELECTRICAL
SUPPLY
BUILDERS
HARDWARE
DEBUG
SOUTHPOINT

Types of Software Testing



Unit Testing (do the parts perform correctly alone?)

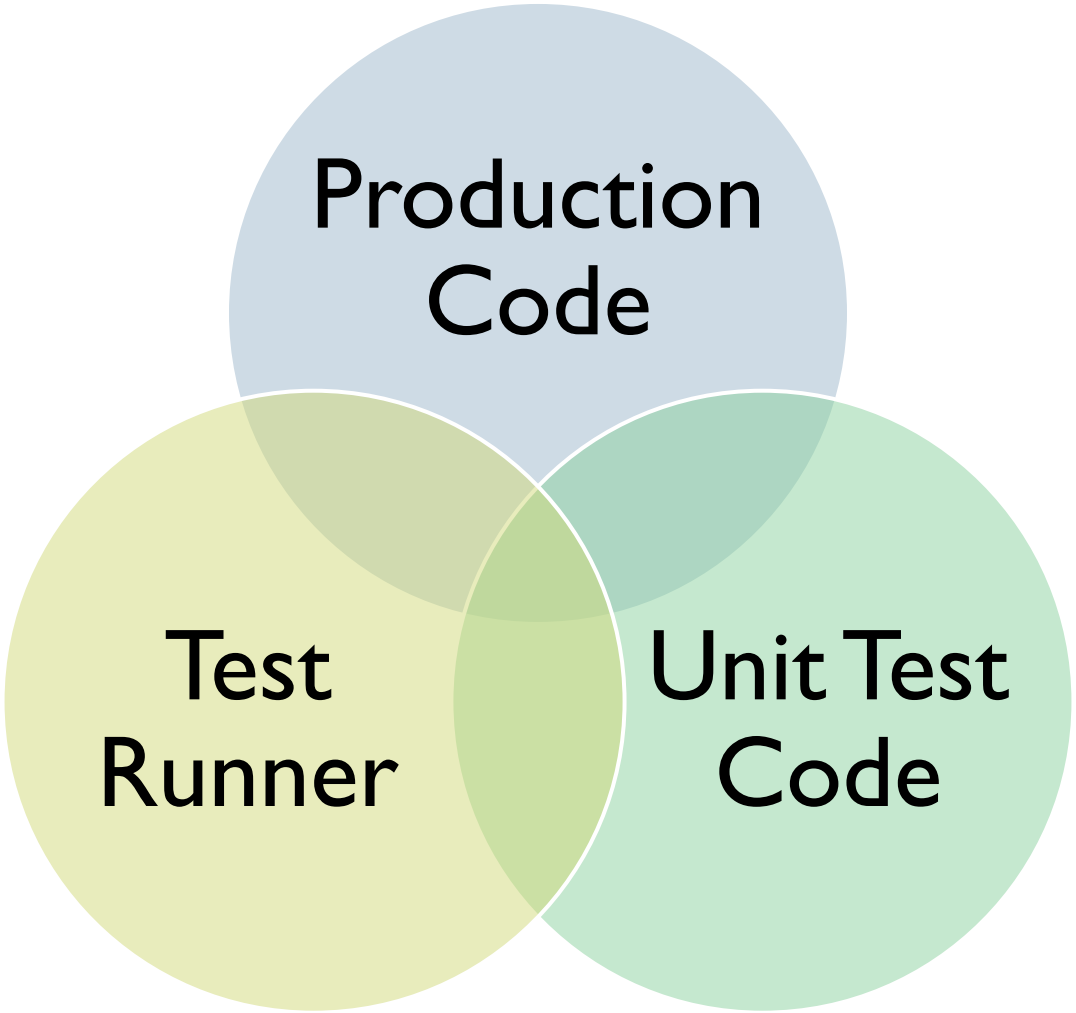
Integration Testing (do the parts perform correctly together?)

User Acceptance Testing (does the system meet the end user's expectations?)

The Concept of Unit Testing

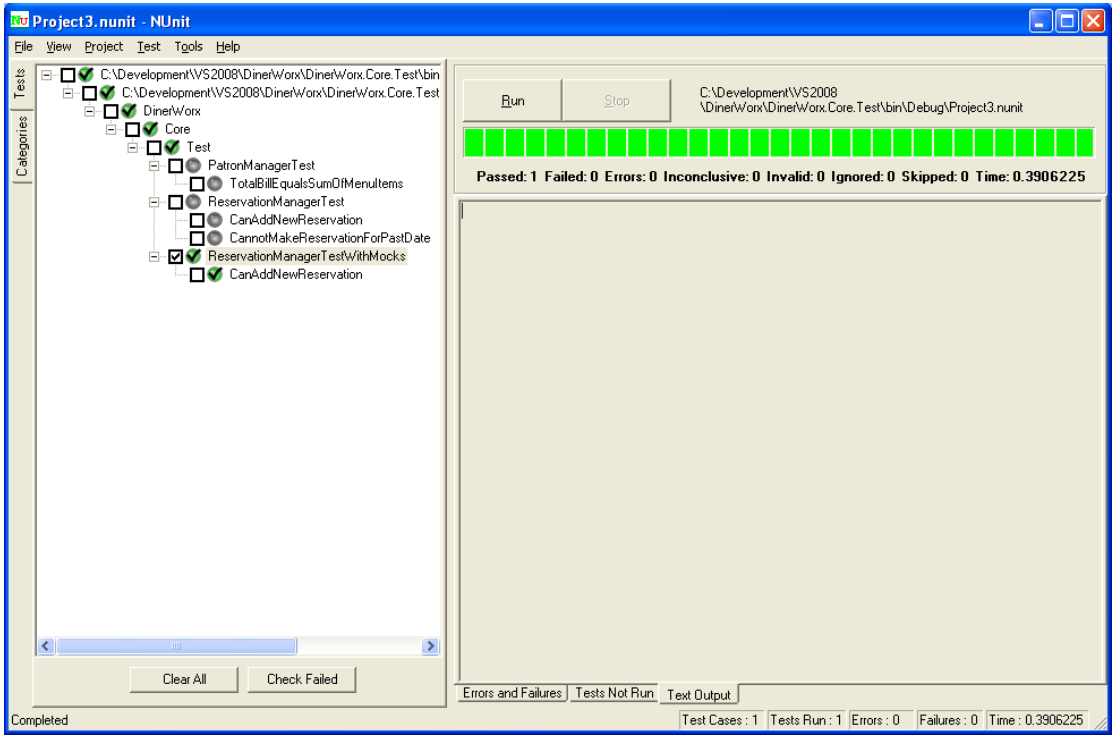
- A unit test is code written by a developer that tests as small a piece of functionality (the unit) as possible.
- One function may have multiple unit tests according to the usage and outputs of the function.
- Tests ensure
  - The code meets expectations and specifications: Does what it says it should do.
  - The code continues to meet expectations over time: Avoiding regression.

Unit Testing Tools

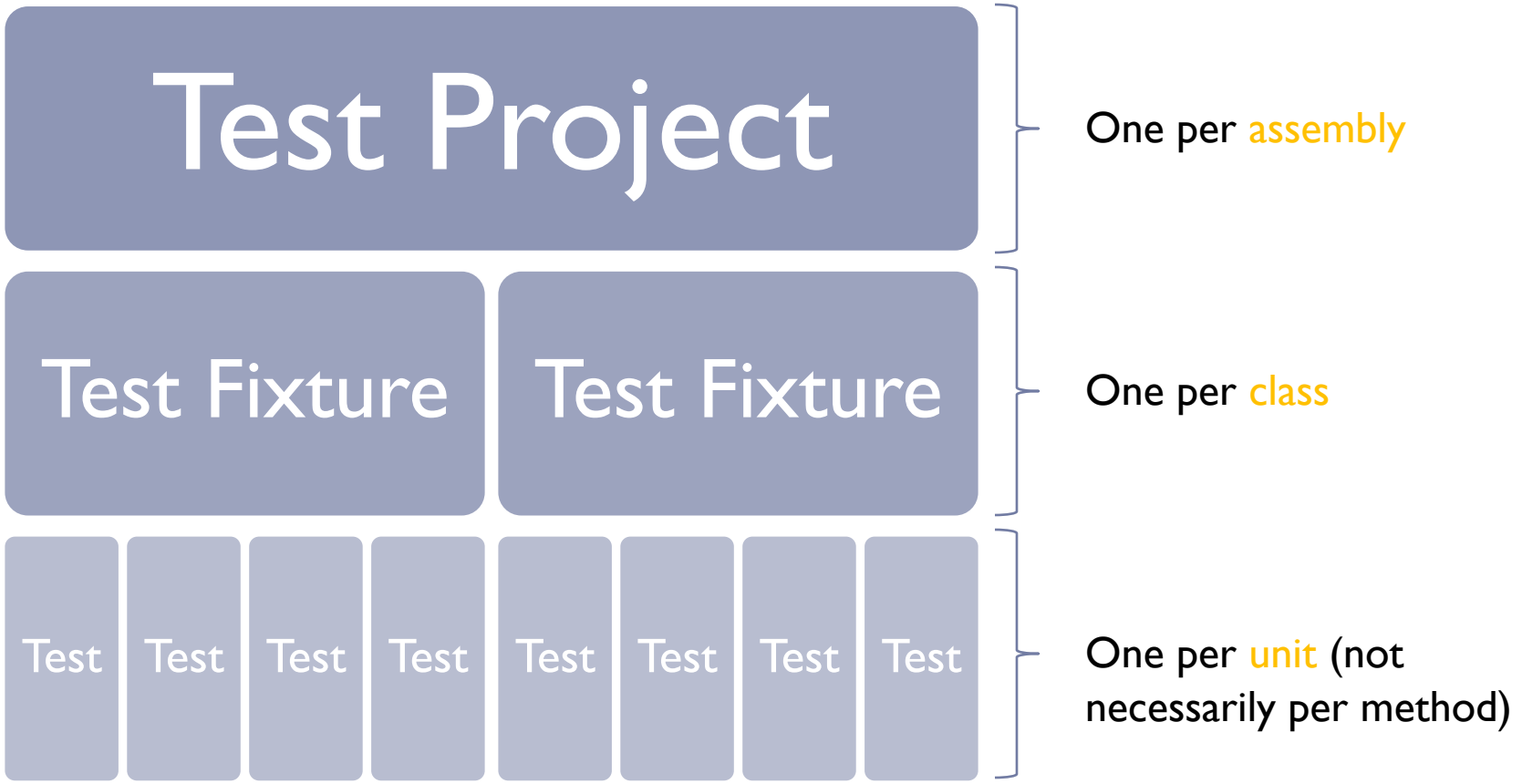


Unit Testing Tools

- Testing Frameworks
  - NUnit
- Test Runner
  - GUI
  - Command line
- Automation
  - CruiseControl.NET



Unit Test Hierarchy



Structure of A Unit Test

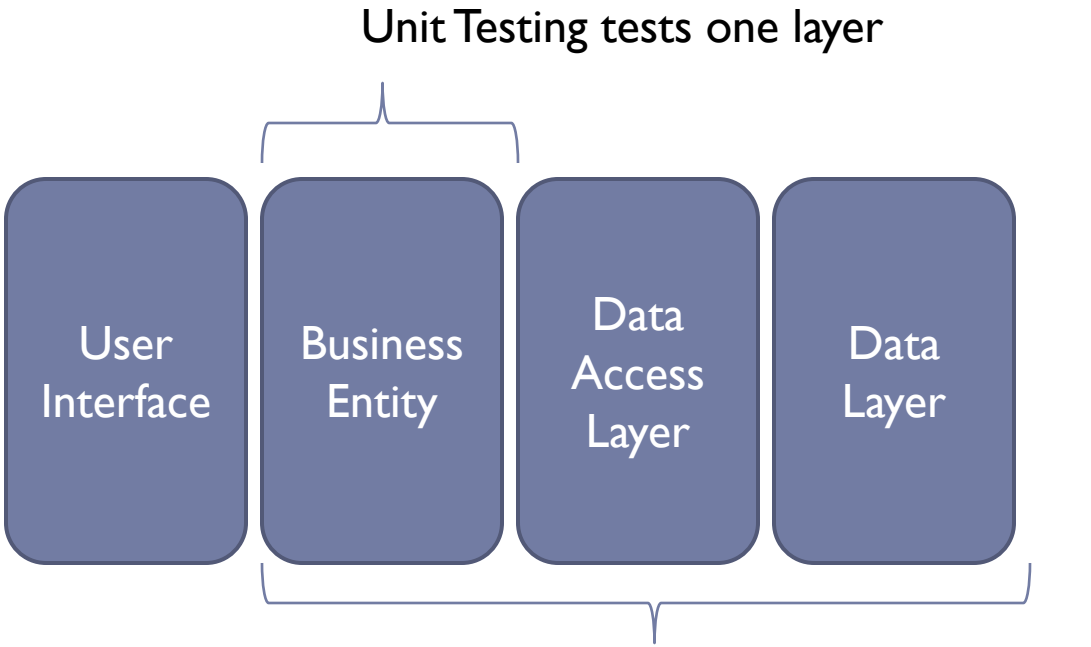
- Setup
- Prepare an input
- Call a method
- Check an output
- Tear down

```
Imports NUnit.Framework

<TestFixture()> _
Public Class PatronManagerTest

    <TestFixtureSetUp()> _
    Public Sub TestSetup()

    End Sub

    <Test()> _
    Public Sub TotalBillEqualsSumOfMenuItems()

        Dim patron As New Patron

        patron.DrinkOrder.Add(New DrinkMenuItem With {.Name = "Coke", .Price = 1.29})
        patron.DrinkOrder.Add(New DrinkMenuItem With {.Name = "Sweet Tea", .Price = 1.29})

        Dim expected As Decimal = 2.58
        Dim actual As Decimal = patron.TotalDrinksBill()

        Assert.AreEqual(expected, actual, "Drink total does not equal sum of drink items.")

    End Sub

    <TestFixtureTearDown()> _
    Public Sub TestTearDown()

    End Sub
End Class
```

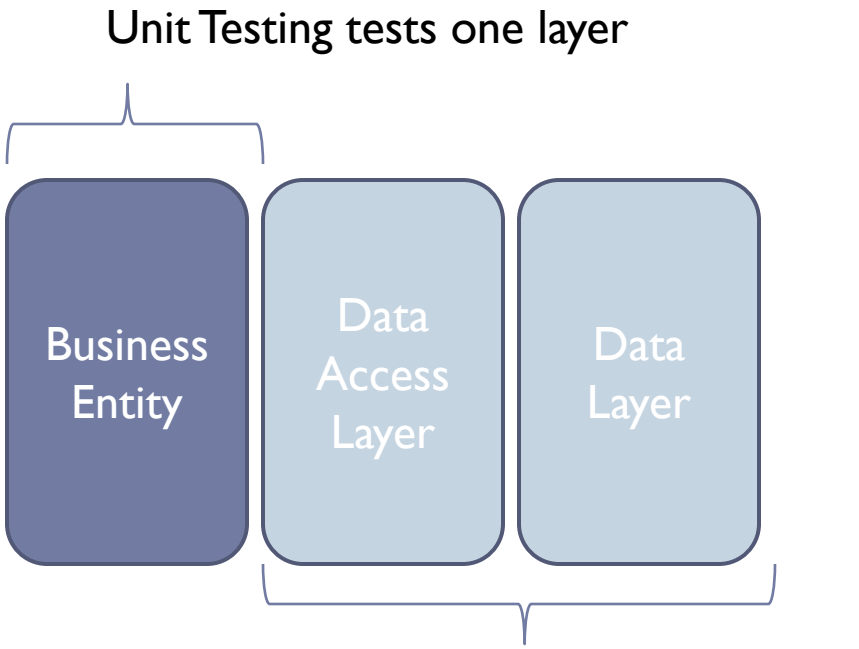
Test Assertions

- Assertions are the 'checks' that you may perform to determine if a test passes or fails.
- For instance:
  - Assert.IsTrue()
  - Assert.IsInstance()
  - Assert.AreEqual()
- Generally speaking, you want ONE assertion per test.

Unit Testing vs. Integration Testing



Unit Testing with Mocks



Executing Tests

Manually:

1. Compile Test project (to .dll or .exe)
2. Open in Test runner.
3. Select and execute tests.

Automatically:

1. Build server compiles and runs tests as part of nightly build operation.
2. Any test failures = entire build fails.

Sample Unit Test

Best Practices

Unit Test Best Practices

1. Consistent
2. Atomic
3. Single Responsibility
4. Self-descriptive
5. No conditional logic or loops
6. No exception handling
7. Informative Assertion messages
8. No test logic in production code
9. Separation per business module
10. Separation per type

Consistent

- Multiple runs of the test should consistently return true or consistently return false, provided no changes were made on code

Code that can cause problems:

```
Dim currentDate as Date = Now()

Dim value as Integer = New Random().Next
```

Unit Test Best Practices

1. Consistent
2. **Atomic**
3. Single Responsibility
4. Self-descriptive
5. No conditional logic or loops
6. No exception handling
7. Informative Assertion messages
8. No test logic in production code
9. Separation per business module
10. Separation per type

Atomic

- Only two possible results: PASS or FAIL
- No partially successful tests.

- Isolation of tests:
  - Different execution order must yield same results.
  - Test B should not depend on outcome of Test A
    - Use Mocks instead.

Unit Test Best Practices

1. Consistent
2. Atomic
3. **Single Responsibility**
4. Self-descriptive
5. No conditional logic or loops
6. No exception handling
7. Informative Assertion messages
8. No test logic in production code
9. Separation per business module
10. Separation per type

Single Responsibility

- One test should be responsible for one scenario only.

- Test behavior, not methods:
  - One method, multiple behaviors → Multiple tests
  - One behavior, multiple methods → One test

Single Responsibility

```
Sub TestMethod()
  Assert.IsTrue(behavior1)
  Assert.IsTrue(behavior2)
  Assert.IsTrue(behavior3)
End Sub
```

```
Sub TestMethodCheckBehavior1()
  Assert.IsTrue(behavior1)
End Sub

Sub TestMethodCheckBehavior2()
  Assert.IsTrue(behavior2)
End Sub

Sub TestMethodCheckBehavior3()
  Assert.IsTrue(behavior3)
End Sub
```

Unit Test Best Practices

1. Consistent
2. Atomic
3. Single Responsibility
4. **Self-descriptive**
5. No conditional logic or loops
6. No exception handling
7. Informative Assertion messages
8. No test logic in production code
9. Separation per business module
10. Separation per type

Self Descriptive

- Unit test must be easy to read and understand
  - Variable Names
  - Method Names
  - Class Names
  - No conditional logic
  - No loops

Self descriptive

- Name tests to represent PASS conditions:
  - Public Sub CanMakeReservation()
  - Public Sub TotalBillEqualsSumOfMenuItemPrices()

Unit Test Best Practices

1. Consistent
2. Atomic
3. Single Responsibility
4. Self-descriptive
5. **No conditional logic or loops**
6. No exception handling
7. Informative Assertion messages
8. No test logic in production code
9. Separation per business module
10. Separation per type

No conditional logic or loops

- Test should have no uncertainty:
  - All inputs should be known
  - Method behavior should be predictable
  - Expected output should be strictly defined
  - Split in to two tests rather than using "If" or "Case"

- Tests should not contain "While", "Do While" or "For" loops.
  - If test logic has to be repeated, it probably means the test is too complicated.
  - Call method multiple times rather than looping inside of method.

No conditional logic or loops

```
Sub TestBeforeOrAfter()
  If before Then
    Assert.IsTrue(behavior1)
  ElseIf after Then
    Assert.IsTrue(behavior2)
  Else
    Assert.IsTrue(behavior3)
  End If
End Sub
```

```
Sub TestBefore()
   Dim before as Boolean = true
  Assert.IsTrue(behavior1)
End Sub

Sub TestAfter()
   Dim after as Boolean = true
  Assert.IsTrue(behavior2)
End Sub

Sub TestNow()
   Dim before as Boolean = false
   Dim after as Boolean = false
  Assert.IsTrue(behavior3)
End Sub
```

Unit Test Best Practices

1. Consistent
2. Atomic
3. Single Responsibility
4. Self-descriptive
5. No conditional logic or loops
6. **No exception handling**
7. Informative Assertion messages
8. No test logic in production code
9. Separation per business module
10. Separation per type

No Exception Handling

- Indicate expected exception with attribute.
- Catch only the expected type of exception.
- Fail test if expected exception is not caught.
- Let other exceptions go uncaught.

No Exception Handling

```
<ExpectedException("MyException")> _
Sub TestException()

    myMethod(parameter)
    Assert.Fail("MyException expected.")

End Sub
```

Unit Test Best Practices

1. Consistent
2. Atomic
3. Single Responsibility
4. Self-descriptive
5. No conditional logic or loops
6. No exception handling
7. **Informative Assertion messages**
8. No test logic in production code
9. Separation per business module
10. Separation per type

Informative Assertion Messages

- By reading the assertion message, one should know why the test failed and what to do.
- Include business logic information in the assertion message (such as input values, etc.)
- Good assertion messages:
  - Improve documentation of the code,
  - Inform developers about the problem if the test fails.

Unit Test Best Practices

1. Consistent
2. Atomic
3. Single Responsibility
4. Self-descriptive
5. No conditional logic or loops
6. No exception handling
7. Informative Assertion messages
8. **No test logic in production code**
9. Separation per business module
10. Separation per type

No test logic in Production Code

- Separate Unit tests and Production code in separate projects.
- Do not create Methods or Properties used only by unit tests.
- Use Dependency Injection or Mocks to isolate Production code.

Unit Test Best Practices

1. Consistent
2. Atomic
3. Single Responsibility
4. Self-descriptive
5. No conditional logic or loops
6. No exception handling
7. Informative Assertion messages
8. No test logic in production code
9. **Separation per business module**
10. Separation per type

Separation per Business Module

- Create separate test project for every layer or assembly
- Decrease execution time of test suites by splitting in to smaller suites
  - Suite I - All Factories
  - Suite II - All Controllers
- Smaller Suites can be executed more frequently

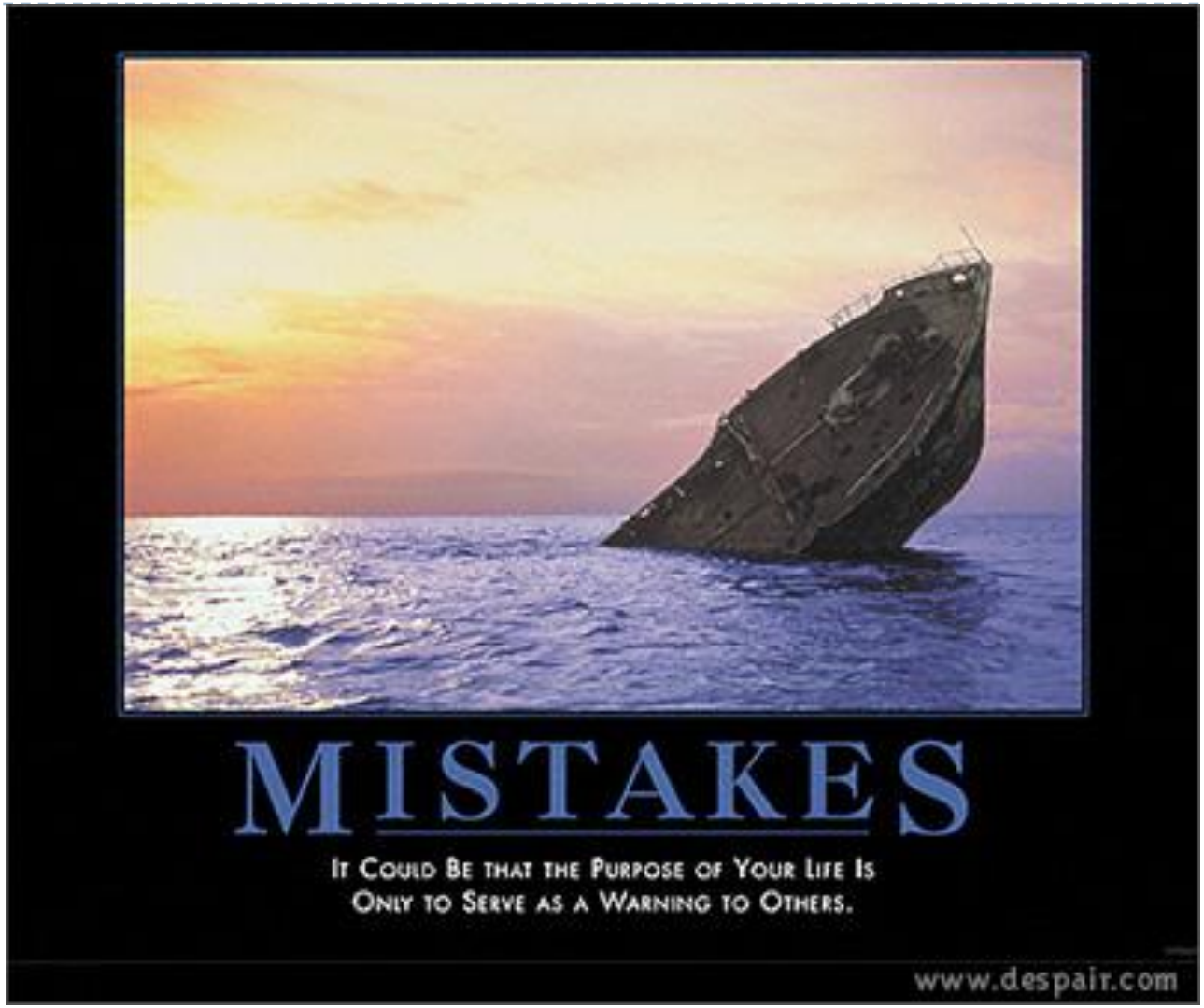
Unit Test Best Practices

1. Consistent
2. Atomic
3. Single Responsibility
4. Self-descriptive
5. No conditional logic or loops
6. No exception handling
7. Informative Assertion messages
8. No test logic in production code
9. Separation per business module
10. **Separation per type**

Separation per Type

- Align Test Fixtures with type definitions.

- Reminder: Unit tests are separate from integration tests!
  - Different purpose
  - Different frequency
  - Different time of execution
  - Different action in case of failure


MISTAKES
It Could Be that the Purpose of Your Life Is
Only to Serve as a Warning to Others.
www.despair.com

- Images from Flickr
- Best practices from www.nickokiss.com