



Unit

RED RIVER
COLLEGE


- Smallest unit of testable code
- Typically a method
 - Single piece of functionality
- In C# this would also include property accessors

Unit Testing

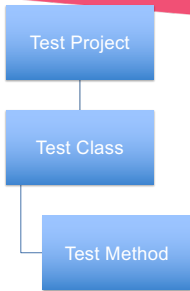
RED RIVER
COLLEGE

- Testing a unit in isolation
- Goal:
 - Unit produces expected behavior (outlined in the requirements) under specific conditions

Unit Test

RED RIVER
COLLEGE

- Code that tests a unit



```
graph TD; TP[Test Project] --- TC[Test Class]; TC --- TM[Test Method]
```

Programming 1 Testing



1. Code a method
 2. Invoke method in a program
 3. Run the program
 4. Execute to the point the method is invoked
 5. Compare expected result to actual
- This process is repeated for all outcomes for each method

Testing Problem #1



- Requirements change
 1. Update method (unit)
 2. Re-test all outcomes of the unit
 3. Re-test all methods that are dependent on the updated method (regression testing)

Testing Problem #2



- Outcome of many methods are to check state
 - Can't access private fields
- Method is declared as private
 - Can't access private methods
 - Can't isolate it from other code

Solution #1



- Unit Testing automates testing
- Tests are performed without interaction from a user
- Testing Frameworks provide functionality you wouldn't have to program yourself
 - Examples:
 - Reporting
 - Test timing and timeouts

Solution #2

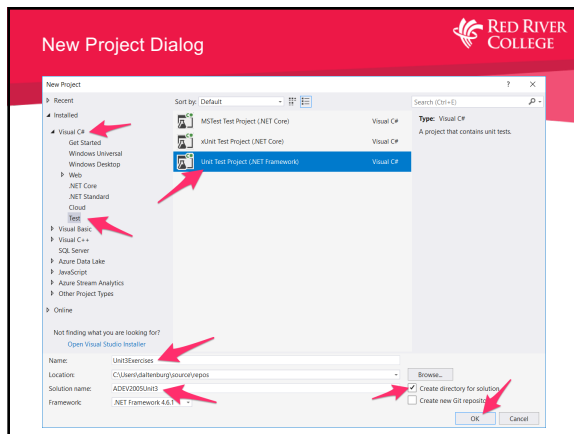


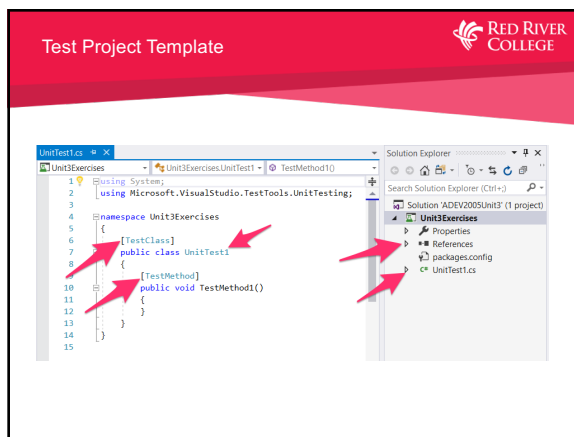
- Reflection
- Allows access to private members of an object

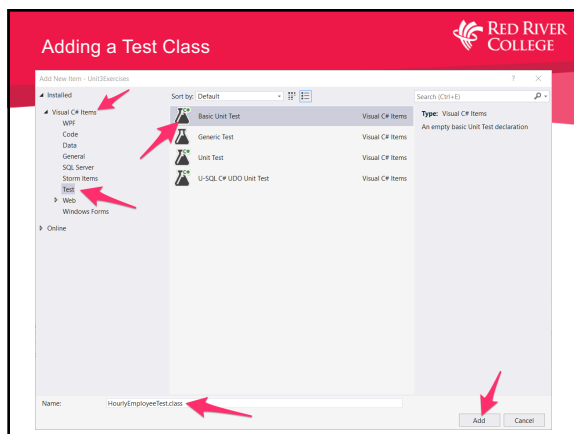
Test Project




- Unit testing is done in a Test Project









Test Class Standards


- Identifier: `{ClassName}Test`
 - Example: SalesQuoteTest
- One Test Class for each class you are testing
 - In its own code file
- One test method per outcome
 - DO NOT combine tests




Unit 3 – Unit Testing
CODING UNIT TESTS

Software Engineering Tips



- Plan your tests prior to unit testing
 - Review the requirements document
 - List all units and outcomes prior to coding tests
- Always test the constructor(s) first

AAA Pattern




1. Arrange
 - Define test data
 - Construct instance (if necessary)
2. Act
 - Invoke unit being tested
 - capture result if necessary
3. Assert
 - Compare actual result to expected result

Before you code – Evaluate Requirement



- Constructor
 - Requirement: Set initial state of the object.
 - What is the outcome?

Arrange – Define Test Data



```
[TestMethod]
public void Constructor_Valid_Test()
{
    string name = "Dan";
    decimal amountOfMoney = 777m;
}
```

Test Name Standards



- Test Method name must include:
 - The name of the unit being tested
 - The conditions of the test
 - Ends with “_Test”

Act – Construct Instance



```
[TestMethod]
public void Constructor_Valid_Test()
{
    string name = "Dan";
    decimal amountOfMoney = 777m;

    Person person = new Person(name, amountOfMoney);
}
```

PrivateObject Class



- Used to access private members of an object
 - PrivateObject(Object)
- Methods
 - GetField(string) : Object

Act – Capture result (actual)



```
[TestMethod]
public void Constructor_Valid_Test()
{
    string name = "Dan";
    decimal amountOfMoney = 777m;

    Person person = new Person(name, amountOfMoney);

    string expectedName = "Dan";
    decimal expectedAmountOfMoney = 777m;

    PrivateObject target = new PrivateObject(person);

    string actualName = (string)target.GetField("name");
    decimal actualAmountOfMoney = (decimal)target.GetField("amountOfMoney");
}
```

Assertions



- Used for test verification
- Assert class (static)
 - contains static methods for various evaluations
 - Examples:
 - AreEqual(Object, Object) : void
 - IsFalse(bool) : void
 - IsTrue(bool) : void

Assert – Compare Expected/Actual



```
[TestMethod]
public void Constructor_Valid_Test()
{
    string name = "Dan";
    decimal amountOfMoney = 777m;

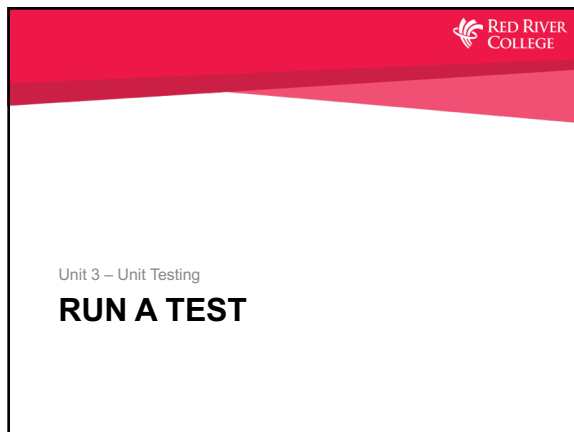
    Person person = new Person(name, amountOfMoney);

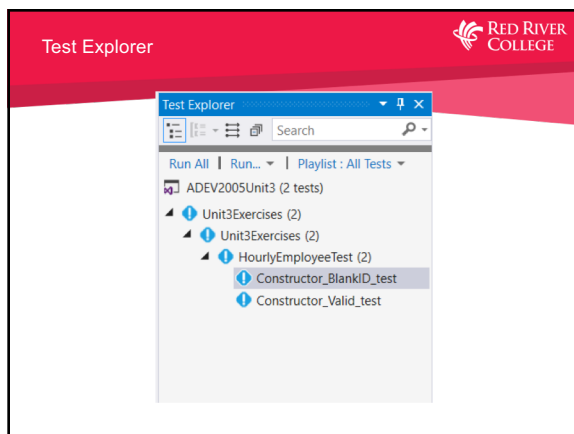
    string expectedName = "Dan";
    decimal expectedAmountOfMoney = 777m;

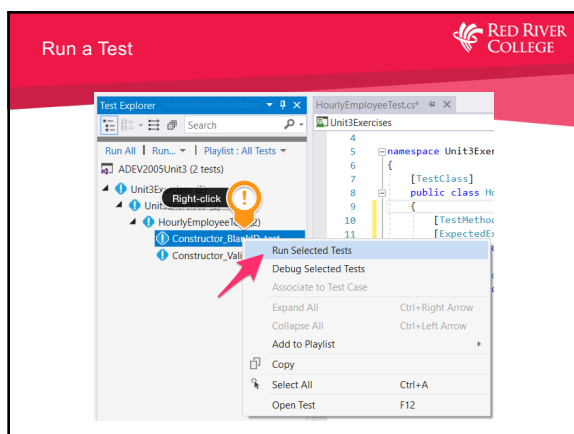
    PrivateObject target = new PrivateObject(person);

    string actualName = (string)target.GetField("name");
    decimal actualAmountOfMoney = (decimal)target.GetField("amountOfMoney");


    Assert.AreEqual(expectedName, actualName);
    Assert.AreEqual(expectedAmountOfMoney, actualAmountOfMoney);
}
```












Test Pass/Fail/Error




- Fail
 - Any assertion in the test method throws an *AssertionFailedException*
- Pass
 - Gets to the end of the test method
- Error
 - Unhandled exception generated in unit test
 - Unhandled exception in code

Test Explorer Icons

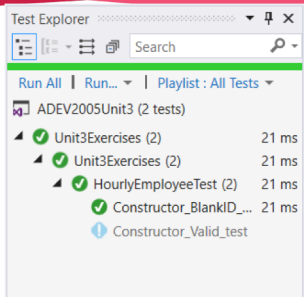


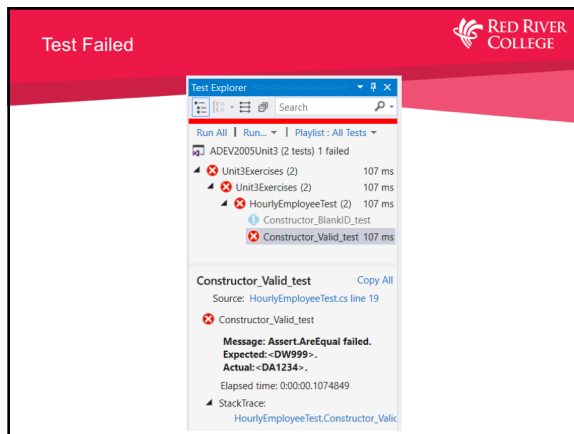
| | |
|---|---|
|  | if group contains failed tests |
|  | if group contains passed tests and NO failed tests |
|  | If group contains skipped tests and NO failed or passed tests |
|  | if group contains not run tests and NO failed, passed, or skipped tests |

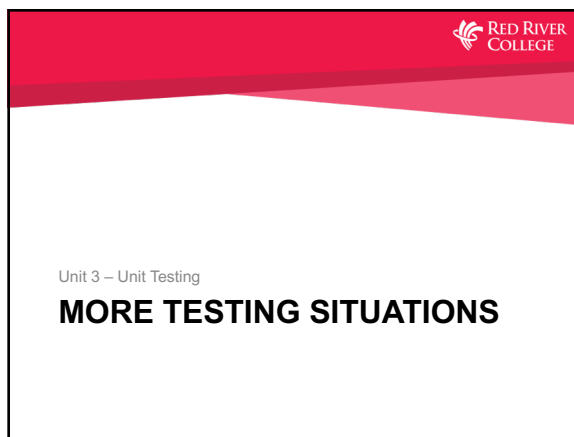
Test Passed

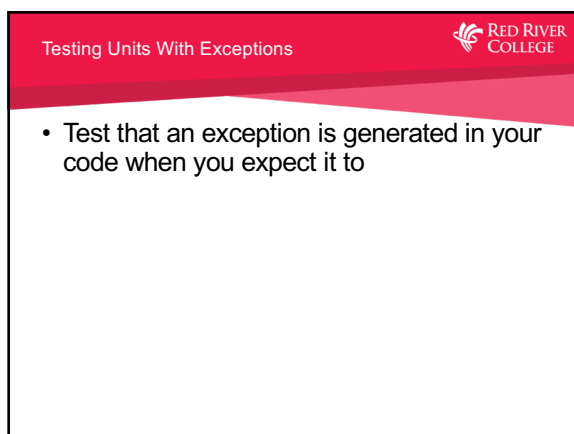


Test Explorer











Unit Test Exception Example

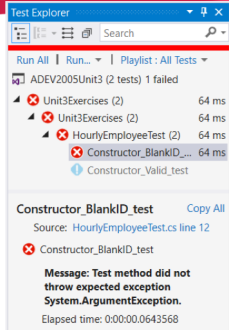


```
[TestMethod]
[ExpectedException(typeof(ArgumentException))]
public void Constructor_PersonWithName_Test()
{
    string name = "";
    decimal amountOfMoney = 777m;


    Person person = new Person(name, amountOfMoney);
}
```

Exception Test Failed





Before you code - Evaluate Requirement



- AmountOfMoney Property
 - Get
 - Requirement: returns amount of money state
 - Set
 - Requirement: modifies amount of money state

Arrange – Define Test Data



```
[TestMethod]
public void GetAmountOfMoney_Test()
{
    string name = "Dan";
    decimal amountOfMoney = 777m;
}
```

Arrange – Construct Instance



```
[TestMethod]
public void GetAmountOfMoney_Test()
{
    string name = "Dan";
    decimal amountOfMoney = 777m;

    Person person = new Person(name, amountOfMoney);
}
```

Act – Invoke Unit



```
[TestMethod]
public void GetAmountOfMoney_Test()
{
    string name = "Dan";
    decimal amountOfMoney = 777m;

    Person person = new Person(name, amountOfMoney);

    decimal expectedAmountOfMoney = 777m;
    decimal actualAmountOfMoney = person.AmountOfMoney;
}
```

Assert – Compare Expected/Actual



```
[TestMethod]
public void GetAmountOfMoney_Test()
{
    string name = "Dan";
    decimal amountOfMoney = 777m;

    Person person = new Person(name, amountOfMoney);

    decimal expectedAmountOfMoney = 777m;
    decimal actualAmountOfMoney = person.AmountOfMoney;
    Assert.AreEqual(expectedAmountOfMoney, actualAmountOfMoney);
}
```

Set Accessor Unit Test



```
[TestMethod]
public void SetAmountOfMoney_Valid_Test()
{
    string name = "Dan";
    decimal amountOfMoney = 777m;

    Person person = new Person(name, amountOfMoney);
    decimal expectedAmountOfMoney = 333m;
    person.AmountOfMoney = 333m;

    PrivateObject target = new PrivateObject(person);
    decimal actualAmountOfMoney = (decimal)target.GetField("amountOfMoney");
    Assert.AreEqual(expectedAmountOfMoney, actualAmountOfMoney);
}
```

State Not Updated On Exception Test



```
[TestMethod]
public void SetAmountOfMoney_NegativeValue_Test()
{
    string name = "Dan";
    decimal amountOfMoney = 777m;

    Person person = new Person(name, amountOfMoney);
    decimal expectedAmountOfMoney = 777m;
    try
    {
        person.AmountOfMoney = -333m;
        Assert.Fail("Did not throw ArgumentOutOfRangeException as expected.");
    }
    catch (ArgumentOutOfRangeException)
    {
        PrivateObject target = new PrivateObject(person);
        decimal actualAmountOfMoney = (decimal)target.GetField("amountOfMoney");
        Assert.AreEqual(expectedAmountOfMoney, actualAmountOfMoney);
    }
}
```

Before you code – Evaluate Requirement



- AddMoney Method
 - Requirement: increment the amount of money state by a specified amount

Unit Test Method Example



```
[TestMethod]
public void AddMoney_Valid_Test()
{
    string name = "Dan";
    decimal startingMoney = 777m;
    decimal additionalMoney = 333m;

    Person person = new Person(name, startingMoney);
    decimal expectedAmountOfMoney = 1443m;

    person.AddMoney(additionalMoney);
    person.AddMoney(additionalMoney);

    PrivateObject target = new PrivateObject(person);
    decimal actualAmountOfMoney = (decimal)target.GetField("amountOfMoney");
    Assert.AreEqual(expectedAmountOfMoney, actualAmountOfMoney);
}
```

Unit Testing Private Methods




```
[TestMethod]
public void NameToUppercase_Valid_Test()
{
    string name = "Dan";
    decimal startingMoney = 777m;

    Person person = new Person(name, startingMoney);
    string expected = "DAN";

    PrivateObject target = new PrivateObject(person);
    string actual = (string)target.Invoke("nameToUppercase");
    Assert.AreEqual(expected, actual);
}
```

Units with Multiple Outcomes

RED RIVER
COLLEGE

```
public static String FizzBuzz(int number)
{
    string message = "";


    if (isDivisibleBy(number, 3))
    {
        message += "Fizz";
    }

    if (isDivisibleBy(number, 5))
    {
        message += "Buzz";
    }

    if (message.Length == 0)
    {
        message += number;
    }


    return message;
}
```

Evaluate Requirements

RED RIVER
COLLEGE

- FizzBuzz Method
 - Requirements:
 - Number divisible by 3 – Result “Fizz”
 - Number divisible by 5 – Result “Buzz”
 - Number divisible by 3 & 5 – Result “FizzBuzz”
 - All other scenarios – Result number as String
- 4 Unit Tests

Unit Testing Static Members

RED RIVER
COLLEGE

```
[TestMethod]
public void ToFahrenheit_Valid_Test()
{
    decimal celsius = 19m;

    decimal expected = 66.2m;

    decimal actual = Conversion.ToFahrenheit(celsius);

    Assert.AreEqual(expected, actual);
}
```

Unit Testing Abstract Classes



- Can't instantiate instance
- Test via concrete class
 - Unit test will be done slightly different; i.e. private fields

Accessing Abstract Class Members



```
public class BaseClass
{
    private int one = 1;
}

public class DerivedClass : BaseClass
{
    private int two = 2;
}

[TestClass]
public class DerivedClassTest
{
    [TestMethod]
    public void Constructor_Valid_Test()
    {
        DerivedClass test = new DerivedClass();

        PrivateObject privateDerived = new PrivateObject(test);
        PrivateObject privateBase =
            new PrivateObject(test, new PrivateType(typeof(BaseClass)));


        Assert.AreEqual(1, (int)privateBase.GetField("one"));
        Assert.AreEqual(2, (int)privateDerived.GetField("two"));
    }
}
```

When a Unit Test Fails




1. Understand why the test is fail
[IMPORTANT]
2. Trace your unit test code
3. Debug method
 - if no error, debug unit test

Test Pass Does Not Mean Correct




```
[TestMethod]
public void SuperComplicatedMethod_Test()
{
    Assert.AreEqual(true, true);
}
```

Best Practices



- Always start with the constructor
- Complete one unit at a time
- Perform tests in order of dependency (when known or possible)
- No unit is insignificant to test
- Test to the requirements, not the code
- Keep unit tests small and simple
- If a test fails, fix code and retest
 - Halt further testing until code is fixed
- Name tests properly

Summary



- Unit = Method
- Unit Test = Code to test code
- Goal = Automation and Accessibility
- Reflection = Access private interface

