learning
solutions

Unit Testing and TDD with C# and xUnit

Principles and Practice

NS40207

NS40207 Unit Testing and TDD with C# and xUnit

COURSE TEMPLATE

This book was created using the **Standard Course Template**, built on LaTeX. The template is an open-source project, released under the MIT license, and available at https://github.com/nextstepitt/standard-course-template.

999 Ponce de Leon Boulevard, Suite 830
Coral Gables, FL 33134
786-347-3155 / 800-418-0811

Licensed to

NextStep IT Training, LLC 3822 Tynemoore Walk SE Symrna, GA 30080	The Judge Group, INC 151 South Warner Rd. Suite 100 Wayne, PA 19087

Introduction

This is focused on unit testing and test-driven development using the xUnit framework. It features a holistic approach that uses a progression to layer related topics instead of focusing on one topic at a time. The participant will be introduced to the basic idea of unit testing, how xUnit and assertions work, strategies for approaching writing tests, test-driven development, using principles and patterns to build testable code, test-doubles and the Moq framework, mocking Entity Framework, testing web service controllers, and using xUnit for integration and system tests. Two chapters at the end introduce the participant to the concepts of ATDD and BDD, using Cucumber with Gherkin to build stakeholder-accepted tests, and performing end-to-end accpentance tests on web and desktop applications.

Preface

My goal over decades of programming has been to explore and develop better methods for software development, and to help other folks achieve the best possible results. To accomplish this I integrate aspects and tools from software engineering and project management. Over many years I have tried many techniques, discarded some, pioneered a few alongside of other people, and continue to actively contribute in this area. My focus has been on consulting, mentoring, and training company teams on techniques to build better software.

This course came about because of a lack of clear material for teaching unit testing and test-driven development. Often text books are constructed by topic, one aspect at a time: xUnit Framework, assertions, Moq, etc. This book uses a holistic approach and integrates the topics in a progression to start building tests right away, layering additional features from multiple topics at each step.

I hope you enjoy participating in this course as much as I enjoyed putting it together!

Joel Mussman

NS40207 Unit Testing and TDD with C# and xUnit

This course focuses on using the *xUnit framework* to build unit tests and introduce test-driven development. The real focus of the course is on how to create testable applications using design principles, and make sure that the code is tested.

Class Setup Requirements

- Visual Studio 2017 or 2019, community edition or higher.
- .NET Core projects and xUnit extensions for Visual Studio.
- Access to Github and NuGet repositories.

Prerequisites

- Experience writing .NET applications with C# in either Core or Famework
- Experience writing unit tests in MSTest, NUnit, or xUnit is a plus.
- JavaScript or TypeScript experience is a plus if web or desktop application testing will be explored.
- HTML5 and CSS3 are a benefit for working with web and desktop applications.

Agenda

- Chapter 1 - Unit Testing
- Chapter 2 - xUnit
- Chapter 3 - Test Requirements
- Chapter 4 - Data Driven Tests
- Chapter 5 - Test Driven Development
- Chapter 6 - Principles and Patterns
- Chapter 7 - Test Doubles
- Chapter 8 - Mocking Entity Framework
- Chapter 9 - Web Services
- Chapter 10 - Integration Testing

Facilities

- Please set your devices to vibrate in the classroom
- If you are on a conference bridge keep the connection muted unless you are talking
- The class has fixed hours and lunch, and there will be breaks; please be here on time
- Facilities: rest rooms, break rooms, coffee, vending, refrigerators, etc.

Contents

Chapter 1
Unit test

Unit Test

Objectives

- Level-set with the big-picture of software development
- Define unit testing, how it works, and where it fits into the big picture

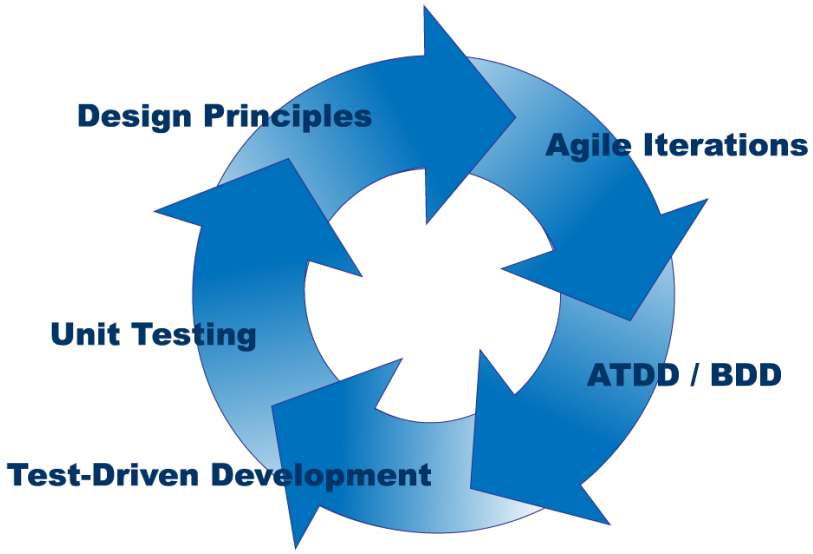
Overview

Over many decades different ideas and techniques have come together to make everything work smoothly in the software development ecosystem. This course focuses on unit testing for software developers, and it is crucial to start with a clear understanding of how the pieces fit together, as well as understanding how unit testing works.

1.1 Unit Testing

Unit Testing

1.1.1 The Big Picture



- Everything depends on everything else
- Acceptance TDD and behavior-driven development drive application design from application requirements
- TDD drives design from component requirements, unit tests verify compliance
- Design principles drive clean, adaptable designs to extend in the next iteration

1.1.2 Two Test Levels

- Application-level testing - driven by ATDD and BDD, usually a quality-assurance responsibility
- Component-level testing - making sure a piece of code works before moving on
- All programmers test code incrementally, unless they are still using punch cards

- Traditionally there were two ways to test code

1.1.3 Testing Code in the Application

- Hack up the application to set the state to test an embedded piece of code
- Easy to make a mistake, especially when backing out the hacks
- Impossible to repeat the test again in the future, unless you put the hacks back

1.1.4 Create a Test Application

- Create a separate application to build a state and run the code
- A lot of work to set up, a lot of work to maintain
- Both ways of testing code discourage real testing because of the effort

1.1.5 Unit Tests

- Developed by Kent Beck with SUnit in 1989 for Smalltalk; provides a scaffold to run tests on
- The scaffold does the heavy lifting, programmers focus on the code for the tests
- The scaffold is kept with the project, therefore the tests can be permanent and repeatable

1.1.6 Given, When, Then

```
[Fact]
public void CalculatorAddsNumbersInRangeSuccess() {

    Calculator calculator = new Calculator();

    int result = calculator.Add(1, 1000);

    Assert.Equal(1001, result);
}
```

- Most individual tests setup (given), do something (when), and check results (then)

- Test method names should be descriptive and are usually lengthy

1.1.7 Terminology

Term	Description
Test Runner	The program that finds and executes tests
Test Case	An individual test that checks a particular result
Test Suite	A set of related test cases
Test Fixture	Established a state that multiple test cases share

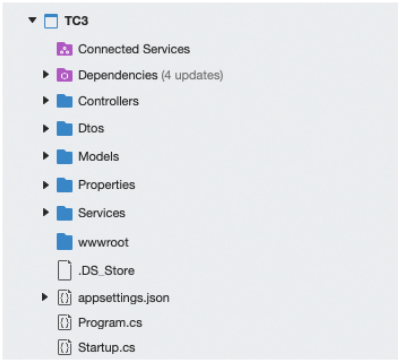
1.1.8 Testing Units

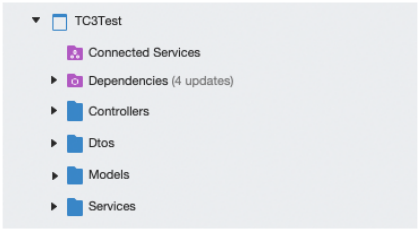
- Test process results, test the public methods in a class (the method, not the class, is the unit)
- Test with good and bad input: negative tests still check for "expected results"
- Negative tests expose a design principle: bad inputs needs to handled in an expected manner

1.1.9 Code-Coverage

- Code-coverage defines the amount of code checked by the tests
- In most circumstances 80-100% is good coverage
- Code not tested: properties without constraints, code not critical, just-in-case code

1.1.10 Test Structure





- .NET project unit tests are contained in a separate project
- Tests are methods of a class; the tests should be related to each other
- Normally establish one test class to parallel each class under test
- The test class structure, folder location and namespace, should parallel the classes under test

1.1.11 Checkpoint

- What does unit testing accomplish?
- How are unit tests organized?
- What defines a unit to test?
- Define code coverage. What defines a good result?
- Explain the normal flow of a unit test.
- What differentiates unit testing from acceptance testing?

Chapter 2
xUnit

xUnit
Lab - Card Number Validation

Objectives

- Explore the syntax of xUnit tests

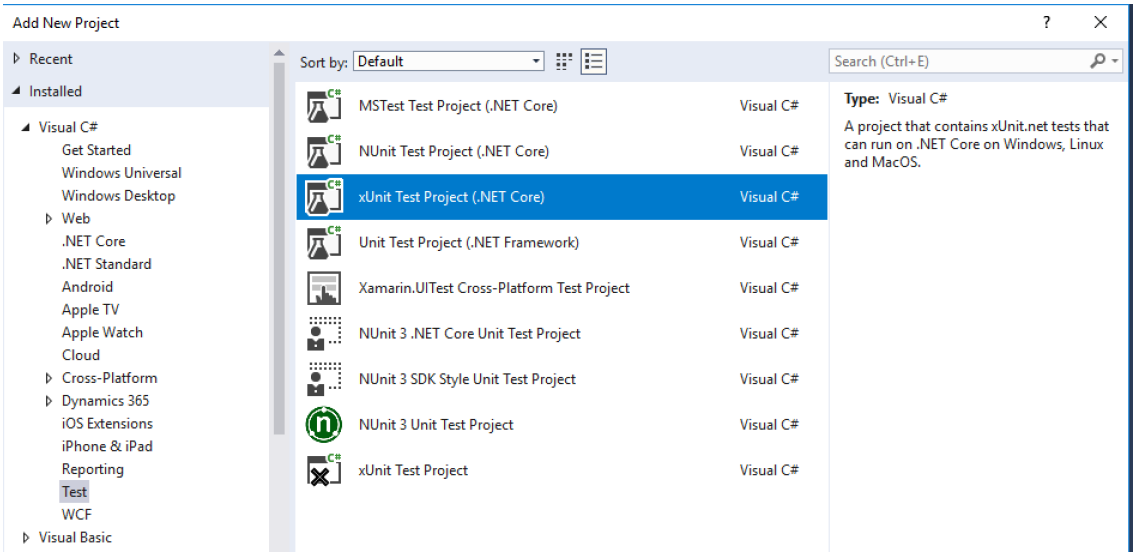
Overview

Explore the syntax of the xUnit framework and build initial tests.

2.1 xUnit

xUnit
Lab - Card Number Validation

2.1.1 xUnit



- A minimalistic unit test framework
- No setup or teardown methods
- Add a new xUnit Test Project to the solution; the interface may vary

2.1.2 Test Classes

```
using XUnit;

namespace TC3Test {

    public class UnitTest1 {
```

- Test classes usually have a one-to-one relationship with source code classes (same folder structure too)
- No attribute identifying a test class, xUnit searches for annotated test methods in all classes

2.1.3 [Fact]

```
[Fact]
public void TestMethod1() {
```

```
    Assert.Equal(1, 1);
}
```

- xUnit uses the *Fact* and *Theory* method attributes; *Theory* is used for data-driven tests (more later)
- The test runner executes each annotated method
- No guarantee what order methods will run in, or the order that classes are run in

2.1.4 Setup, Run, Evaluate, Teardown

```
[Fact]
public void CalculatorAddsNumbersInRangeSuccess() {

    // Given
    Calculator calculator = new Calculator();

    // When
    int result = calculator.Add(1, 1000);

    // Then
    Assert.Equal(1001, result);
}
```

- Build the environment for each test
- Execute the code
- Assert that some result is true (or false)
- Perform any necessary tear-down

2.1.5 Test Setup and Teardown Utility Methods

```
public class CalculatorTest : IDisposable {

    private Calculator calculator;

    private void GivenCalculator() {

        calculator = new Calculator();
    }

    [Fact]
    public void CalculatorAddsNumbersInRangeSuccess() {

        GivenCalculator();

        int result = calculator.Add(1, 1000);

        Assert.Equal(1001, result);
    }

    public Dispose() {
```

```
        // Teardown here
    }
}
```

- xUnit does not declare a setup or teardown utilities with attributes
- Setup methods must be private and called in the methods that need them; often named with "Given"
- A separate class instance is created to run each test method; this ensures a clean state for each test

2.1.6 Assertions

Assertion	Definition
True(actual)	Value is expected to be true
True(actual, message)	Override the default message in the report
False(actual)	Value is expected to be true
False(actual, message)	Value is expected to be false

```
[Fact]
public void TestMethod1() {

    Assert.True(false); // Force a failure
}
```

- Assertions are static helper methods of the Assert class, they verify the expected results
- Failure is detected by the test runner because the assertion throws an exception
- If an assertion fails, that test fails, but all of the tests run even if there are failures
- Generally the assertion methods have a form with a message parameter to override the default message

2.1.7 Equality

Assertion	Definition
Equal(expected, actual)	Are equal
Equal<T>(T expected, T actual, IEqualityComparer<T> comparer)	Are equal
NotEqual(expected, actual)	Are not equal
NotEqual<T>(T expected, T actual, IEqualityComparer<T> comparer)	Are not equal

```
Assert.Equal(1, 1);
```

- Assertions are built to accept any type of argument; both arguments should be of the same type
- Put the expected result first and the actual result second to keep the messages in the report correct

- Equality between objects depends on the *default comparator*, the *Equals* method being overridden, or a provided comparer

2.1.8 Floating-Point Numbers

Assertion	Definition
Equal(double expected, double actual, int precision)	Check floating point numbers
NotEqual(double expected, double actual, int precision)	Assert the numbers are not equal

```
Assert.Equal(1.0001, 1.0002, 3);     // True
Assert.Equal(1.0001, 1.0006, 3);     // False
```

- Doubles and float calculations can drift, the precision controls how many decimal points to check
- The numbers will be *rounded* to the correct number of decimal points before the comparison

2.1.9 Dates

Assertion	Definition
Equal(DateTime expected, DateTime actual, TimeSpan precision)	Check dates
NotEqual(DateTime expected, DateTime actual, TimeSpan precision)	Assert the dates are not equal

```
DateTime expected = DateTime.Now;
DateTime now = DateTime.Now; // A little bit in the future

Assert.Equal(expected, now, new TimeSpan(0, 0, 0, 5)); // Within five seconds
```

- DateTime objects could be a slightly off, the TimeSpan provides flexibility

2.1.10 Identity

Assertion	Definition
Same(expected, actual)	Are the same object
NotSame(expected, actual)	Are not the same object

2.1.11 Numbers

Assertion	Definition
True(x < y)	Less than
True(x <= y)	Less than or equal to
True(x > y)	Greater than
True(x >= y)	Greater than or equal to

- xUnit does not have assertions for numerical comparisons
- The logic for this is: why not just do the operation and check the boolean result?

2.1.12 Type, Assignment, and null

Assertion	Definition
IsAssignableFrom(type, object)	Expects and instance or subclass instance of a specific type
IsAssignableFrom<T>(object)	Expects and instance or subclass instance of a specific type
IsType(type, object)	Is exactly an instance of a specific type
IsType<T>(object)	Is exactly an instance of a specific type
IsNotType(type, object)	Expects anything but an instance of a specific type
IsNotType<T>(object)	Expects anything but an instance of a specific type
Null(reference)	The reference is null
NotNull(reference)	The reference is not null

- Be very wary of type in applications, including tests; it tightly couples components to classes
- If absolutely necessary to check that a method (like a factory) generated the correct subclass...

2.1.13 Exceptions

Throws(Action or func<object>)	The action or function throws some exception
ThrowsAsync(func<task>)	The asynchronous task throws some exception
ThrowsAny<T>(Action or func<object>)	The action or function throws T or a subclass of T
ThrowsAnyAsync<T>(func<object>)	The asynchronous action or function throws T or a subclass of T

```
[Fact]
public void CalculatorRejectsSecondArgumentOutOfRange() {

    Calculator calculator = new Calculator();

    Assert.Throws<ArgumentOutOfRangeException>(() => calculator.Add(1, 1001));
}
```

- One way to handle exceptions is to put the test code in try-catch blocks; that is a lot of work
- Use a lambda expression to set up the "then" part of the flow
- The lambda delays execution until the test runner has set up a handler for the expected exception

2.1.14 Checkpoint

- Which attribute identifies an xUnit test class?
- The four sections of any test are?
- The general flow of any test is?
- What does an assertion do?
- Which assertions compare numbers?
- Why not check an exception by just calling the method?

2.2 Lab - Card Number Validation

xUnit
Lab - Card Number Validation

Goals

- Set up the xUnit testing environment and build initial tests.

Synopsis

A credit card number should be checked for validity before attempting a purchase. Merchant services accounts provide purchase authorization services. Authorization services have a very short amount of time to authorize or deny a purchase (<= 80ms). Blocking bad credit card numbers before the attempt is made eliminates wasted network traffic and reduces the load on the payment authorizer.

Requirements

- An application-service must be provided to validate credit card numbers.
- Credit card numbers are passed to the validation as strings.
- Card numbers must contain only digits.
- The Luhn algorithm must be used to verify the card number.
- Invalid card numbers are expected, the service return value should indicate failure.
- A null card number is an unexpected event and a NullReferenceException exception should be thrown.
- There must be appropriate unit tests to check the validation method.
- Be prepared to demonstrate your work to the group.

Steps

1. Start with the skeleton retail service project TC3 (The Caribbean Coffee Company).
2. Create a *Services* folder (if necessary) to organize service classes that support the application.
3. Create a *CardValidator* class with a static boolean *ValidateCardNumber(string cardNumber)* method to check card numbers.
4. The method should throw an exception if the *cardNumber* is null.
5. The method should return null if the string is not all digits.
6. Check the *cardNumber* with the *Luhn Algorithm*. The *Luhn algorithm* at *Wikipedia.org*, with a short pesudo code function is on the page. Hint: use *Char.GetNumericValue(c)* returns a digit for a character as a floating point value, so cast it to an int. Watch out in the algorithm and make sure that you include every digit in the string.
7. Add a new *xUnit Test Project* to the solution and call it *TC3Test*.

8. Make sure that the *xUnit* and *xUnit Test Runner* NuGet packages were added as dependencies.
9. Create an *xUnit* test project named *EC3Tests*.
10. Identify and implement unit tests for this the *ValidateCardNumber* method. Dummy credit card numbers for testing may be found online. For example a search for "paypal test card numbers" will locate a list at https://developer.paypal.com/docs/payflow/payflow-pro/payflow-pro-testing/.

Summary

The focus of this lab was to use the xUnit framework to create unit tests.

Congratulations, you have completed this lab!

Chapter 3
Test Requirements

Objectives

- Learn how to identify what tests should be performed
- identify and test for boundary conditions
- Consider when an internal test is appropriate, and when to adapt code for a test

Overview

The chapter encourages focusing on what unit testing is really supposed to test: the interface of a unit. Starting with requirements, we create and implement candidates for tests and test data.

3.1 Test Requirements

Test Requirements
Boundaries
Boxes
Lab - Test Requirements

3.1.1 Component Requirements

Requirements	Description
Business Requirements	High-level requirements: main features, target audience, maybe a technology.
Stakeholder Requirements	What stakeholders need the system to do: quickly do processes, remove stumbling blocks.
Solution Requirements	
Functional Requirements	Specific behaviors, features, business rules.
Non-Functional Requirements	Quality of service: performance, maintainability, etc. Not assumed, are still explictitly stated.
Transition Requirements	Requirements to move from development to production
Technical Requirements	Ecosystem (platform, language, etc.)
Component Requirements	**Requirements of individual components to function in the system**

- Application requirements (business - technical) are defined by stakeholders, business analysts, system architects, etc. and *are not unit tested*!
- Component requirements are discovered/defined by programmers as the design evolves

3.1.2 Unit Testing

```
public class Calculator {
    public int Add(int x, int y) {
        return x + y;
    }
}
```

- Object-oriented programming and unit testing are made for each other
- Test suites are based on classes, but each test focuses on testing a method
- Remember: the unit is a *process*, not a class!

3.1.3 Test Class Properties?

```
public class Pair {
    private int y;
    public int X { get; set; }
    public int Y {
        get { return y; }
        set { y = value >= 0 ? value : y; }
    }
}
```

- Test only if the property has constraints; X has no constraints so skip it
- *Y* has a not-negative constraint, so that must be tested
- This lines up with the philosophy of testing requirements; Y is public and implements a requirement

3.1.4 Beware of False Positives

```
[Fact]
public void TwoNumbersInTheRange() {
    int result = calculator.Add(2, 2);
    Assert.Equal(4, result);
}
```

- Watch out for questionable data when designing a test
- What if the Add method did multiplication instead? The test would pass!

3.1.5 Avoid Testing Other Units

```
public class EmployeesController {

    IList<Employee> employees;

    public int LoadEmployees() {

        using (var dbCntext = new TC3Context()) {

            employees = dbCntext.Employees.where(e => e.Salary > 100000);
        }

        return employees.Count;
    }
}
```

```
[Fact]
public EmployeesMakingGreatorThanOrEqualTo100000() {

    EmployeesController controller = new EmployeesController();
    int employeeCount = controller.LoadEmployees();

    Assert.Equal(employeeCount, 10);
}
```

- Avoid testing what a dependency does, test how the class under test is working
- Subtle: test that the right query is fulfilled, not that the database runs it correctly
- Note the test principle violation: LoadEmployees returns a value to check (more later)

3.1.6 Stateful Dependencies

```
[TestFixture]
public class CalculatorTests {

    static Calculator calculator = new Calculator();
    TC3Context dbContext;

    private void GivenTC3Context() {

        dbContext = new TC3Context();
    }

    [Fact]
    public GetEmployeeTest() {

        GivenTC3Context();
        SalesOrder salesOrder = dbContext.SalesOrders.Find(1234);
        Assert.Equal(1, salesOrder.Id);
    }
}
```

- Calculator is used in every test, but is not stateful so instation at load is OK
- HRController probably has state, so it must be reset before each test

3.1.7 Negative Tests

```
[Fact]
public void CalculatorRejectsSecondArgumentOutOfRange() {

    Calculator calculator = new Calculator();

    Assert.Throws<ArgumentOutOfRangeException>(() => calculator.Add(1, 1001));
}
```

- Tendency is to focus on the happy-path scenarios
- Negative tests are not failures; test that bad input produces expected results
- An expected result could be a specific value, an exception...

3.2 Boundaries

Test Requirements
Boundaries
Boxes
Lab - Test Requirements

3.2.1 Calculator Class Requirements

1. The calculator shall have Add, Subtract, Multiply, and Divide methods that each accept two integers.
2. There shall be a Modulus method to return the remainder of division of two integers.
3. The methods accept positive integers in the range from 1 to 1000 (inclusive).
4. An *ArgumentOutOfRangeException* shall be thrown if the constraints are not met.

- Look to the class requirements: what tests can we identify from these?
- Strongly typed C# eliminates most worries about passing the correct data type as a parameter
- Pay attention to the constraints: many requirements define boundaries (edge conditions), focus on those

3.2.2 Calculator Tests

Test Description	Result	Why
Add 1 and 1000	1001	In bounds numbers work
Add 1000 and 1	1001	Numbers work in each argument
Add 0 and 1001	Exception	Both arguments are out of bounds
Add 1001 and 0	Exception	Both arguments are out of bounds
Add 1001 and 1	Exception	First argument out of bounds (high)
Add 1 and 1001	Exception	Second argument out of bounds (high)
Add 0 and 1	Exception	First argument out of bounds (low)
Add 1 and 0	Exception	Second argument out of bounds (low)
Add -1 and 1	Exception	First argument out of bounds (low)
Add 1 and -1	Exception	Second argument out of bounds (low)

```
[Fact]
public void TwoNumbersInTheRange() {

    int result = calculator.Add(1, 1000);
    Assert.Equal(1001, result);
}
```

- Ten tests identified for the Add method, times five calculator operations is 50 tests!
- Notice only the edges are tested, what would be the point of random numbers within the range?

3.2.3 Zero is Special

```
[Fact]
public void ZeroAndNumberInTheRange() {

    Assert.Throws<ArgumentOutOfRangeException>( () => calculator.Add(0, 1) )
}

[Fact]
public void NegativeAndNumberInTheRange() {

    Assert.Throws<ArgumentOutOfRangeException>( () => calculator.Add(-1, 1) )
}
```

- The requirements indicate that the edge to test for is zero
- Zero is special, programmers make mistakes with >, >=, <, and <= so test separately
- When zero is involved, always test around it with -1, 0, and 1

3.3 Boxes

Test Requirements
Boundaries
Boxes
Lab - Test Requirements

3.3.1 Avoid Testing Internals

```
public class Calculator {

    public int Multiply(int x, int y) {

        int result = 0;

        for (int i = 0; i < y; i++) {

            result += x;
        }

        return result;
    }
}
```

```
[Fact]
public void AddsTwoNumbersInTheRangeSuccess() {

    int result = calculator.Add(1, 1000);
    Assert.Equal(1001, result);
}
```

- *Single Responsibility*: I worry about my responsibility, not theirs
- A *behavioral* test; We are not ignorant of what the class does, we just don't care how it does it!

3.3.2 Avoid Interface Points for Test

```
    public class EmployeesController {

        IList<Employee> employees;

        public void LoadEmployees() {

            using (var dbContext = new TC3Context()) {

                employees = dbContext.Employees.where(e => e.Salary > 100000);
            }
        }
    }
```

- Stick to behavioral tests: do not add interface points to expose information for a test
- The interface point was eliminated, and it is pretty hard to test a *void* method
- So why was this method even public? Maybe it belongs in the constructor, maybe in a private helper method. Sometimes this situation reveals a design flaw...

3.3.3 Internal Behavior Tests

Hidden requirement: the max length of a name is 10 characters

```
public class Employee {

    private char lastname[11];
    public char* getLastName();
    public void setLastName(char* lastname);
}

public char* Employee::getLastName() {

    return lastname;
}

public void Employee::setLastName(char* lastname) {

    strncpy(this.lastname, lastname, 10);
}
```

- This is a C++ example; C# has eliminates many buffer overflow issues; when could it exhibit one?
- This is a hidden requirement because the programmer (or the database) dictated the length of the buffer
- What is the expected result: Reject the operation? Discard the extra data?
- What is a better solution for both the code and the problem?

3.3.4 Error on Invalid Data

```
private string lastName;

public LastName {
    get { return lastName; }
    set {
        if (value.Count > 10) {
            throw new ArgumentOutOfRangeException();
        }
        lastName = value;
    }
}
```

- This is not an unreasonable requirement: data has to fit in a field in a database
- Test for the exception

3.3.5 Gray Box Test

```
// Test that the name is truncated
employee.LastName = "Wolfeschlegelsteinhausenbergerdorff";
Assert.Equal("Wolfeschle", employee.LastName);
```

- Again, hhis is not an unreasonable requirement: data has to fit in a field in a database
- In this case the property silently truncates the data to fit
- We "know" the data will be truncated, so test for it

3.3.6 Testing Internals

The requirements state (for some silly reason) that a *List* must be used internally:

```
[assembly:InternalsVisibleTo("TC3Tests")]
public class Department {

    internal IList<Employee> employees = new List<Employee>();
}
```

```
[Fact]
public void EmployeeListIsArrayList() {

    Department department = new Department();

    Assert.IsType<List<Employee>>(department.employees);
}
```

- Requirements about internal implementaion are bad, but they happen, so we need a "glass-box" test
- Requires the internal state to be exposed either as a public property or with a compiler feature
- For signed assemblies add the test assemply public key: InternalsVisibleTo("TC3Tests", PublicKey=)

3.3.7 Checkpoint

- What should unit testing focus on?
- What are boundary conditions? Why are they important?
- Should we worry about the state of a class?
- What is a behavorial test?
- What is a gray-box test?
- What is a glass-box test?

3.4 Lab - Test Requirements

Test Requirements
Boundaries
Boxes
Lab - Test Requirements

Goals

- Refactor the project using our improved knowledge of unit testing

Requirements

A few new requirements have been added to our project:

- Credit card information should be passed around in instances of *CardInfo*: number (string), name (string), expires (date), ccv code (int). The card is expired at 00:00 (12:00AM) on first day of the month following the month the card expires in.
- The credit card must be a Visa, MasterCard, or American Express. Visa card numbers begin with a 4. MasterCard starts with 51-55. American Express is 34 or 37.
- The card must expire in the future, but not more than five years into the future.
- Add two new methods to the validation class: *ValidateExpirationDate(DateTime expires)* and *ValidateProvider(string cardNumber)*. These methods should throw new *InvalidExpirationDate* and *InvalidCardProvider* exceptions.
- There must be a method *ValidateCardInfo(CardInfo cardInfo)* that controls validation and delegates to the other three methods.
- All appropriate tests must be designed and implemented. Refactor in any tests identified but missed in the previous lab.

Steps

1. Create the *InvalidExpirationDate* and *InvalidCardProvider* exception classes.
2. Create the *CardInfo* class in *Models* with the public properties described in the requirements.
3. Add the three new validation methods to *CardValidator*: *ValidateExpirationDate(DateTime expires)*, *ValidateProvider(string cardNumber)*, and *ValidateCardInfo(CardInfo cardInfo)*. *ValidateCardInfo(CardInfo cardInfo)* leverages the other three methods and throws the same exceptions.
4. Identify and implement the tests for the new requirements.

Summary

This lab focused on checking boundary conditions in the credit card data. We will see more boundaries as the course progresses.

Congratulations, you have completed this lab!

Chapter 4
Data-Driven Tests

Parameterized Tests
Lab - Parameterized Tests

Objectives

- Set up data as a source for running a unit test

Overview

Using a data source for unit testing handles the circumstance where one test can be used repeatedly with different data.

4.1 Parameterized Tests

Parameterized Tests
Lab - Parameterized Tests

4.1.1 Data-Driven Tests

- There are often circumstances where a single test run with different data is useful
- The test runs with every data point, regardless if it fails some of the time

4.1.2 Parameterized Tests

```
[Theory]
[InlineData(1, 2, 3)]
[InlineData(2, 1, 3)]
public void AddNumbersIsReflexive(int a, int b, int expected) {

    Calculator calculator = new Calculator();

    Assert.AreEqual(expected, calculator.Add(a, b));
}
```

- Parameterized tests have the data embedded with the test
- The data is passed as parameters to the test method

4.1.3 MemberData and Properties

```
[Theory]
[MemberData(nameOf(TestData))]
public void AddNumbersIsReflexive(int a, int b, int expected) {

    Calculator calculator = new Calculator();

    Assert.AreEqual(expected, calculator.Add(a, b));
}

public static IEnumerable<object[]> TestData =>
    new List<object[]> {
        new object[] { 1, 2, 3 },
        new object[] { 2, 1, 3 }
    };
}
```

- MemberData is useful when a static property or method in the test class will generate the datapoints
- It works the same as InLineData, except the points come from the property
- Because the data is provided from code, the source could be a file, a database, anything!

4.1.4 MemberData and Methods

```
[Theory]
[MemberData(nameOf(TestData), parameters: 2)]
public void AddNumbersIsReflexive(int a, int b, int expected) {

    Calculator calculator = new Calculator();

    Assert.AreEqual(expected, calculator.Add(a, b));
}

public static IEnumerable<object[]> TestData(int count) {

    IList<object[]> data = new List<object[]> {
        new object[] { 1, 2, 3 },
        new object[] { 2, 1, 3 },
        new object[] { 2, 3, 5 }
    };

    return data.Take(count); // returns two data points
}
```

- A method may be passed parameters (it doesn't have to be)

4.1.5 MemberData and Classes

```
[Theory]
[MemberData(nameOf(TestDataContainer.TestData), MemberType=typeof(
    ↪ TestDataContainer))]
public void AddNumbersIsReflexive(int a, int b, int expected) {

    Calculator calculator = new Calculator();

    Assert.AreEqual(expected, calculator.Add(a, b));
}
```

```
public class TestDataContainer {

    public static IEnumerable<object[]> TestData(int count) {

        IList<object[]> data = new List<object[]> {
            new object[] { 1, 2, 3 },
            new object[] { 2, 1, 3 },
            new object[] { 2, 3, 5 }
        };

        return data.Take(count); // returns two data points
    }
}
```

- The property or method can be declared in another class
- Useful if the data needs to be shared across test classes

4.1.6 ClassData

```
[Theory, PropertyData(nameOf(TestData))]
public void AddNumbersIsReflexive(int a, int b, int expected) {

    Calculator calculator = new Calculator();

    Assert.AreEqual(expected, calculator.Add(a, b));
}
```

```
public class TestData : IEnumerable<object[]> {

    private IList<Object[]> data = new List<object[]> {
        new object[] { 1, 2, 3 },
        new object[] { 2, 1, 3 }
    };

    public IEnumerator<object[]> GetEnumerator() {

        yield return new object[] { 1, 2, 3 };
        yield return new object[] { 2, 1, 3 };
    }

    IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
}
```

- ClassData moves the source to another class, useful if several test classes need to share the data
- xUnit will look for the class to implement the IEnumerable interface

4.1.7 CSV

```
[Theory]
[CsvData(@"C:\MyDataPoints.csv")]
public void AddNumbers(int a, int b, int expected) {

    Calculator calculator = new Calculator();

    Assert.AreEqual(expected, calculator.Add(a, b));
}
```

- The first row is expected to be headers over the fields
- Careful here, externalized data may become inconsistent

4.1.8 Excel

```
[Theory]
[ExcelData(@"MyDataPoints.xls", "Select * from TestData")]
public void AddNumbers(int a, int b, int expected) {
    Calculator calculator = new Calculator();
```

```
    Assert.AreEqual(expected, calculator.Add(a, b));
}
```

- The name in the select statement is a named selection in the first workbook tab
- Careful here, externalized data may become inconsistent

4.1.9 Checkpoint

- Under what circumstances are data sources for unit tests useful?
- What do you need to consider when using a data source?
- Are boundaries and data sources compatible? Can I reduce the number of tests?

4.2 Lab - Parameterized Tests

Parameterized Tests
Lab - Parameterized Tests

Goals

- There are several places in the project that could benefit from a data set using the same test.

Requirements (revisited)

- There are no new requirements

Project Steps

1. The tests to check good and bad credit cards are repetitive, convert them to parameterized tests.
2. Run the altered tests to make sure that everything still works.

Summary

Cleaner code. We removed the redundancy in the unit tests.

Congratulations, you have completed this lab!

Chapter 5
Test-Driven Development

Test-Driven Development
Lab - Test-Driven Development

Objectives

- Define Test-Driven Development
- Understand how TDD works
- Implement TDD

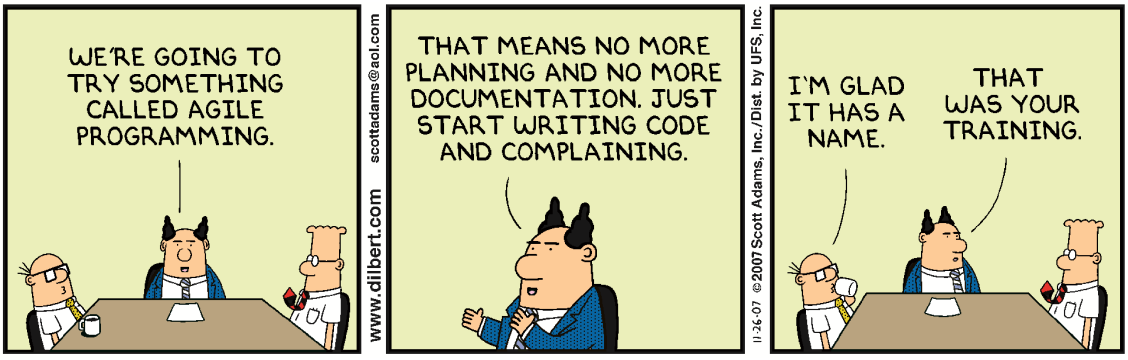
Overview

Test-Driven Development is a hot-button topic, viewed by many organizations as a fix-all-our-problems solution for software development. It is important to understand what TDD is, and what it is not, and how it fits with everything else that makes for a successful software development environment.

5.1 Test-Driven Development

Test-Driven Development
Lab - Test-Driven Development

5.1.1 Test-Driven Development

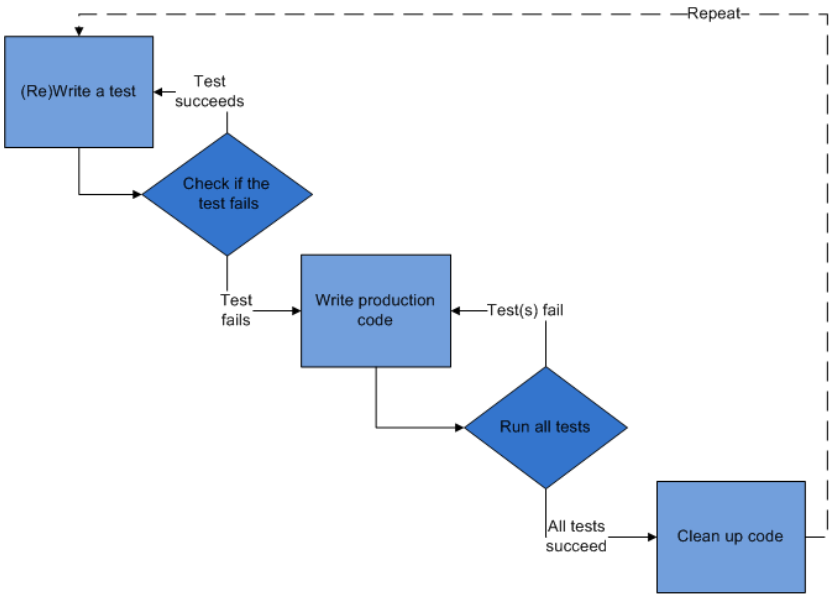


- Kent Beck introduced TDD along with *Unit Testing* and *Extreme Programming*
- Instead of drawing out a design, TDD drives the design in application code from the test code
- Write a test, now we understand the problem so write the code that solves it!
- Agile methodologies are iterative without monolithic designs, so TDD fits right in

5.1.2 Am I Forced into TDD?

- Heck no! Unless of course your manager says so...
- Everything doesn't have to be TDD: if it works for you then use it, even if your team rejects it!
- Fits with pair programming: write a test and switch drivers to write the code

5.1.3 TDD Flow



- Write a test. Of course it fails, nothing satisfies it because it won't even compile!
- Write code to satisfy the test, not done until the test is satisfied. OK, that is good
- Refactor: do not walk away until the code is clean and it "smells" right

5.1.4 Test First

```
[Fact]
public void AddTwoNumbersInTheRangeSuccess() {

    Assert.Equal(1001, calculator.Add(1, 1000));
}
```

- To write the test, the requirement must be understood
- Do not know (or care) where calculator comes from, have decided that an Add method is needed
- The *test* is out of the way!

5.1.5 IDE and IntelliSense

- All IDEs have context help (which in Visual Studio is called IntelliSense)
- "auto-completion mode" gets in the way with TDD, where you press space or . and it selectes an item
- Switch "suggestion mode" where only tab picks the selected item so space and . work normally

5.1.6 Code Second

```
public class Calculator {

    public int Add(int a, int b) {

        return 1001;
    }
}
```

- The *test* defined what needs to be created: the *Calculator* class and the *Add* method!
- We have a good idea of what we need to do, because we understand the requirement
- And, we are focused only on writing code that satisfies the test

5.1.7 Write the Minimum

```
public class Calculator {

    public int Add(int a, int b) {

        return 1001;
    }
}
```

- Kent Beck emphasied write the minumum code to satisfy the test, and returning 1001 does that
- Even if this accidently got left behind it will probably be discovered in another test
- Was it really that hard to write a + b?

5.1.8 New Requirement, New Test

```
public void RejectsZeroAndNumberInTheRange() {

    Assert.Throws<ArgumentOutOfRangeException>( () => calculator.Add(0, 1000) )
}
```

```
public class Calculator {

    public int Add(int a, int b) {

        if (a == 0) {

            throw new ArgumentOutOfRangeException();
        }

        return a + b;
    }
}
```

- Throw an ArugmentOutOfRangeExcpetion when a value is not between 1 and 1000 inclusive
- Write a test that checks at least part of that, then update the code to satisfy everything

- Of course there are multiple tests: (0, 1000), (1000, 0), etc.

5.1.9 Same Requirement, New test

```
public void RejectsNegativeOneAndNumberInTheRange() {

    Assert.Throws<ArgumentOutOfRangeException>( () => calculator.Add(-1, 1000) )
}
```

```
public class Calculator {

    public int Add(int a, int b) {

        if (a <= 0) {

            throw new ArgumentOutOfRangeException();
        }

        return a + b;
    }
}
```

- Add the next test, then add the code to support it

5.1.10 Same Requirement, New Tests

```
public void RejectsNumberInTheRangeAndZero() {

    Assert.Throws<ArgumentOutOfRangeException>( () => calculator.Add(1, 0) )
}

public void RejectsNumberInTheRangeAndNegativeOne()) {

    Assert.Throws<ArgumentOutOfRangeException>( () => calculator.Add(1, -1) )
}
```

```
public class Calculator {

    public int Add(int a, int b) {

        if (a <= 0) {

            throw new ArgumentOutOfRangeException();
        }

        if (b <= 0) {

            throw new ArgumentOutOfRangeException();
        }
```

```
        return a + b;
    }
}
```

- Do not forget to write the tests to mirror everything with the second argument.
- We are not done yet...

5.1.11 Refactor!

```
public void ZeroAndNumberInTheRange() {

    Assert.Throws<ArgumentOutOfRangeException>( () => calculator.Add(-1, 1000) )
}
```

```
public class Calculator {

    private IsArgumentInRange(value) {

        return value >= 1 && value <= 1000;
    }

    public int Add(int a, int b) {

        if (!IsArgumentInRange(a) || !IsArgumentInRange(b)) {

            throw new ArgumentOutOfRangeException();
        }

        return a + b;
    }
}
```

- Does the code smell? Is there a simpler way to code it?
- From the previous example the range check can be refactored into a helper method for DRY

5.1.12 Test-DRIVEN Development

- Often what we need is not identified until the test; we discovered that we need a calculator so that drives creating a Calculator class
- Balance some up-front design with letting the programmers build the design while testing and coding; up-front design should be the big picture
- That does not mean that what is designed in the test should not be backed into our any UML diagrams!

5.1.13 Company Benefits

- The tests are written and not forgotten; and tests are the best documenation of how the code works!
- Passing the tests reduces bugs in the program, and bugs are more expensive to track down than testing

- Eliminates any tendency to "gold-plate" the code and add "what-if" features on the fly

5.1.14 Programmer Benefits

- It is actually faster; the tests have to be written anyways and it makes coding clearer
- You write cleaner code because you are focused on satisfying the test
- You spend less time chasing bugs

5.1.15 Checkpoint

That was fast!

- What are the benefits of writing the test first?
- What is the opposition to writing the test first?
- Why was an emphasis placed on the word *driven* in TDD?

5.2 Lab - Test Driven Development

Test-Driven Development
Lab - Test-Driven Development

Goals

- Explore using Test-Driven Development to build project tests and code

Synopsis

Explore using TDD to identify design factors from tests implementing requirements.

Requirements

- There must be a class that manages sales orders, the *SalesOrderManager*.
- The class must verify that credit card info is valid before attempting to complete a sale to take the burden off the merchant services account.
- Before completing the sale the class must also verify that the total of the items in the sales order is greater than zero and less than $250.

Steps

- Write tests for the new class to verify the card info is checked. What requirements were identified? What design issues came up? What needs to be tested?
- What class or classes were discovered? What methods are necessary? Where do they go? Implement the code.
- Write tests for the new class to verify the total is handled propertly. Where does the total come from? Who is responsible for it?
- As tests are written, write the corresponding application code to satisfy them.

Summary

We have developed tests that now reveal what we needed to build: a service that has methods to check if a card number is valid, calculate the total purchase, and reject the sale if anything is amiss before requesting authorization (don't worry, authorization is coming).

Congratulations, you have completed this lab!

Chapter 6
Principles and Patterns

Design Principles
Design Patterns
Lab - Adapter Pattern

Objectives

- Introduce the principles of design
- Explore how the principles allow the development of reliable, adaptable, and maintainable software
- Understand how patterns relate to principles support testing and Agile

Overview

Even if unit testing and test-driven development help drive the design during development, there have to be good practices to implement a reliable, adaptable, and maintainable design. Understanding and following design principles and design patterns ensures that good, clean, and testable code is created.

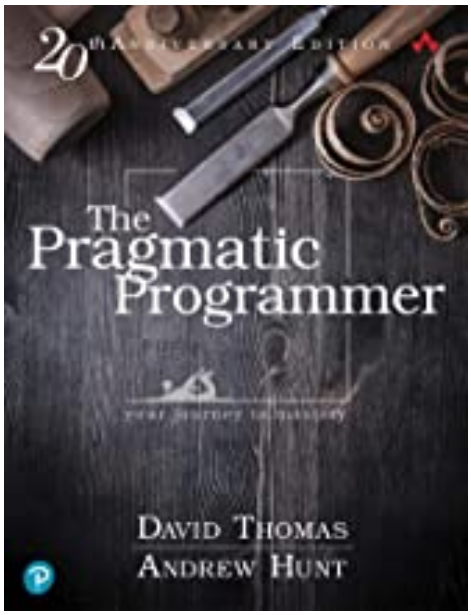
6.1 Design Principles

Design Principles
Design Patterns
Lab - Adapter Pattern

6.1.1 Design Principles

Don't Repeat Yourself	**DRY: Andy Hunt and Dave Thomas**
Single responsibility principle	**SOLID: Robert C. Martin**
Open for extension, closed for modification	
Liskov substitution principle	
Interface segregation	
Dependency inversion	
Inversion of Control	**Jackson Structured Programming: Michael Jackson**
Information Experts	**General Responsibility Assignment Patterns (GRASP): Craig Larman**
Creator	- related to inversion of control
Controller	
Indirection	
Low Coupling	- dependency inversion
High Cohesion	- enclosed by single responsibility
Polymorphism	- the open/closed and dependency inversion principles
Protected Variations	- open/closed
Pure Fabrication	

6.1.2 DRY



- Question: why were functions (and procedures) added to programming languages?
- Apply DRY to both code and data

6.1.3 Single Responsibility

- Single Responsibility is a super-set encompassing cohesion; it is stricter
- Uncle Bob original wrote about single responsibility in relation to classes
- But, it applies to all levels of programming: application, module, class, function, and statement

6.1.4 Open for Extension, Closed for Modification

- Once something is built, do not modify it without good reason
- Fundamental to OOP: extend a class or interface for new functionality
- Wraps polymorphism: the client does not care what class/functionality it uses

6.1.5 Liskov Substitution

- Barbara Liskov extended the basic principle of substitutability
- You cannot change the expected behavior: A car should go forward in drive, and backwards in reverse
- The focus is on substituability for polymorphism, and not breaking the client by behaving badly

6.1.6 Interface Segregation

- Similar to single responsibility: do not create huge classes (or interfaces) with a big API
- Large interfaces probably break single responsibility; worse implementers must build everything
- Build small classes (or interfaces) and combine them with composition or inheritance

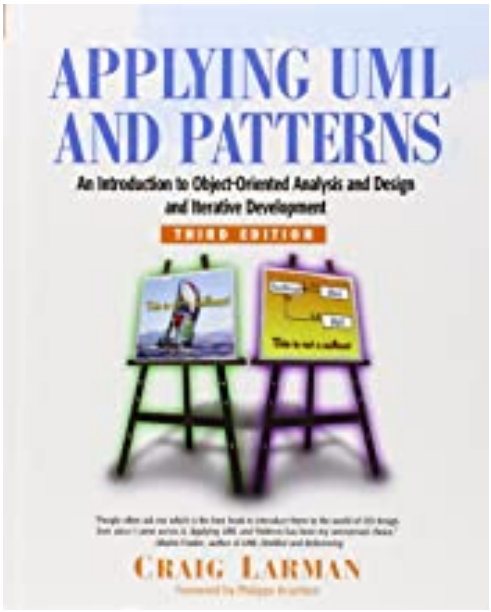
6.1.7 Dependency Inversion and Inversion of Control

- Depend on abstractions, not concretions
- The more abstract the dependency, the more things may be substituted where it is used
- Depending on a particular class, especially when the *new* operator is used, is tight coupling

6.1.8 Inversion of Control

- Inversion of Control (IoC) depends on *Dependency Inversion*
- A class should not be responsible for creating its dependency and implementing the processes that use it
- Something else should be responsible for the dependency

6.1.9 Information Expert



- Assign responsibility to the class that has the information to complete it

6.1.10 Creator

- Give class A creation responsibilities to:
    1. Has the responsibility of managing instances of A
    2. Records/persists instances of A
    3. Closely uses instances of A
    4. Has the information to instantiate A (a.k.a. information expert)
- Creator is mostly eclipsed now by inversion of control, where the responsibility belongs to the IoC container

6.1.11 Controller

- A component that responds to requests and controls the process
- In modern applications delegates work to service and view components

6.1.12 Indirection

- Create a class to act as an intermediary
- For example, a controller class processes a request and decouples model components from view components

6.1.13 Pure Fabrication

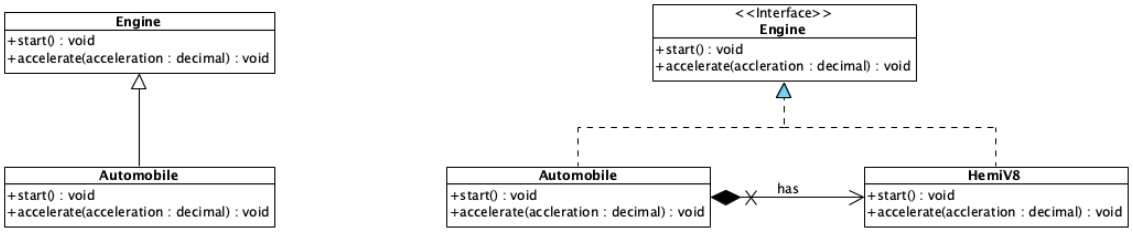
- A component that represents nothing in the problem domain, but is necessary to support other components
- For example, a factory that builds instances of employees; a service class
- OOP Rule #1: when in doubt, create a new class for the responsibility

6.1.14 Interfaces

```
public interface Employee {

    void decimal CalculateWeeklyPay();
}

public class HourlyEmployee : Employee {

    public decimal CalculateWeeklyPay(decimal hoursWorked) {

        return this.PayRate * hoursWorked;
    }
}
```

- Only use super-classes and inheritance when DRY needs to be supported, otherwise coose interfaces
- Salary employee is not bound by inheritance, and now chould have another super-class in the future

6.1.15 Composition over Inheritance

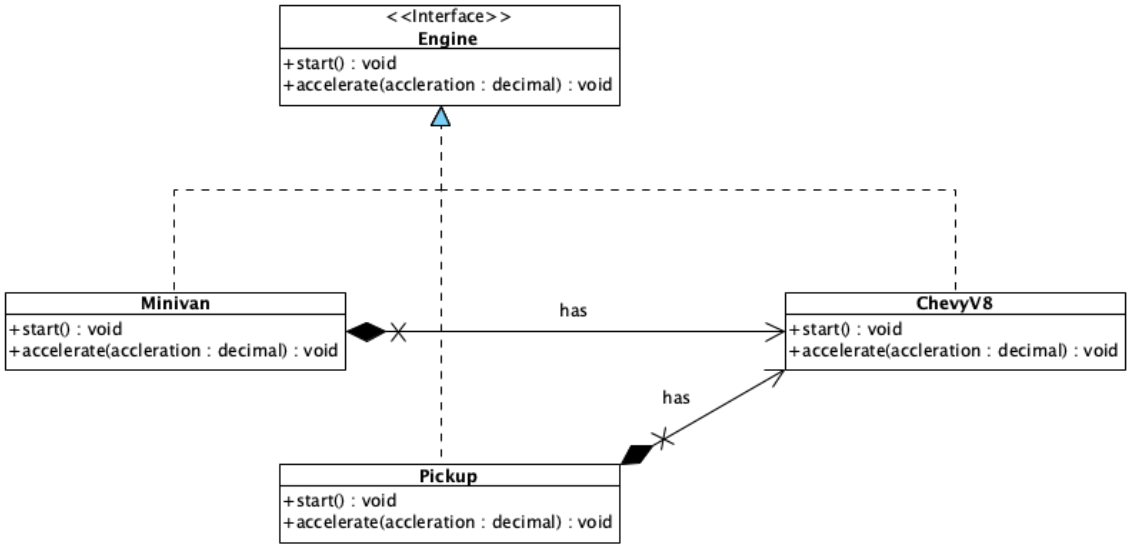



```
class Automobile : EngineClass {

    // You do not do this, what else could a car extend?
}

class Automobile : EngineInterface {

    // This is composition, better (but not great)
    private EngineInterface engine = new HemiV8():

    public void Start() { Engine.start(); }

    public void Accelerate(decimal accel) { engine.Accelerate(accel); }
}
```

- Inheritance is about DRY, not polymorphism

- Composition is more flexible, we can change what is used
- The Unified Modeling Language *class diagram* shows the static relationships

6.1.16 Sharing Composition




```
class Minivan : Engine {

    private Engine engine = new ChevyV6();

    public void Start() { _engine.start(); };
    public void Accelerate(decimal accel) { engine.Accelerate(accel); };
}

class Pickup : Engine {}

    private Engine engine = new ChevyV6();

    public void Start() { engine.Start(); };
    public void accelerate(decimal accel) { engine.Accelerate(accel); };
}
```

- Composition is "like" intheritance, because you can delegate work to the component
- To support dry across mutlipel classes implementing the same interface, use composition
- *Minivan* and *Pickup* effectivly "inherit" from the engine class!

6.1.17 Overriding Composition

```
class Minivan : Engine {

    public void Accelerate(decimal accel) {
```

```
        engine.Accelerate(accel < 100 ? accel : 100);
    };
}
```

- *Minivan* overrides the acceleration and limits it, this is like overriding an inherited method

6.1.18 Code Smell

```
public class SalesManager {

    private IList<SalesOrder> salesOrders = new List<SalesOrder>();
    private chaseAuthorizationService = new ChaseCardAuthorization();

    public string completeSale(SalesOrder sale, CardInfo cardInfo) {

        return chaseAuthorizationService.Authorize(salesOrder.Total, cardInfo);
    }
}
```

- The application uses a merchant service for credit card authorization
- The checkout method of the sales manager instantiates the service, and that "smells" bad
- A "code smell" is when something feels or looks like it code be done better

6.1.19 Dependency Injection

```
public class SalesManager {

    private IList<SalesOrder> salesOrders = new List<SalesOrder>();
    private CardAuthorizationService authorizationService;

    public SalesManager(CardAuthorizationService authorizationService) {

        this.authorizationService = authorizationService;
    }
}

public class Bootstrap {

    private CardAuthorizationService authorizationService;
    private SalesManager salesManager;

    public Bootstrap() {

        authorizationService = LoadAuthorizationService();
        salesManager = new SalesManager(authorizationService);
    }
}
```

- Dependency injection supports dependency inversion and inversion of control
- Someone else makes the decsion of what to use, and then that is *injected* into the class
- Injection has three forms: constructor, property, or Reflection (Unify or Spring)

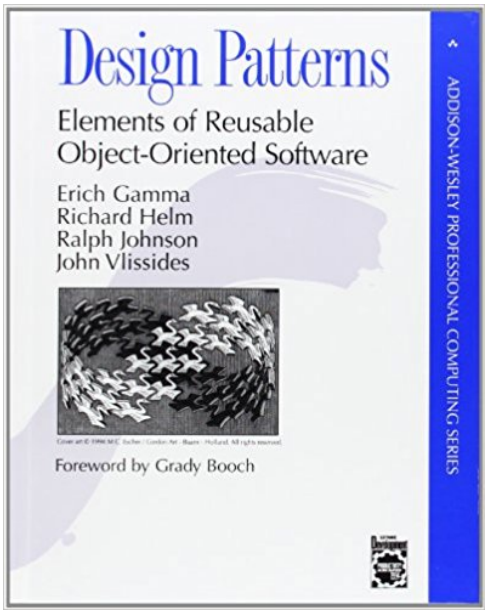
6.1.20 Testing and Dependency Injection

- Testing is not the primary reason for dependency injection, but DI is critical for testing
- With DI, one dependency can be changed out for another during testing

6.2 Design Patterns

Object-Oriented Design Principles
Design Patterns
Lab - Adapter Pattern

6.2.1 Erich Gamma & The Gang of Four

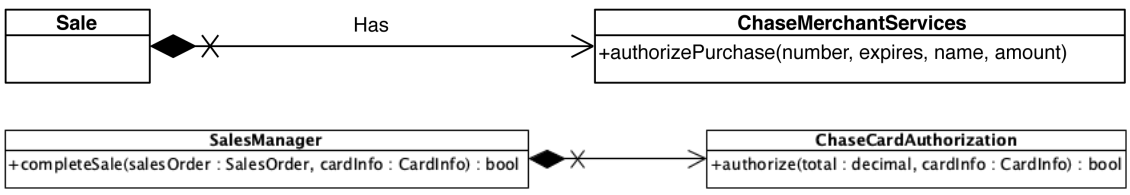


- The Gang of Four: Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides
- Identifies 23 Design Patterns that led to successful applications
- Published in 1994 and mostly C++, so don't get dogmatic about it

6.2.2 Design Patterns and OOP Design Principles

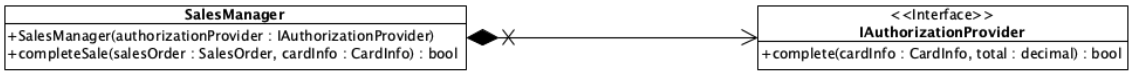
- Every pattern is a manifestation of the principles
- 67% of the 23 patterns use dependency injection and Open/Closed as their foundation!
- The patterns are good to recognize, identifying them in code means the principles were followed

6.2.3 Open/Closed



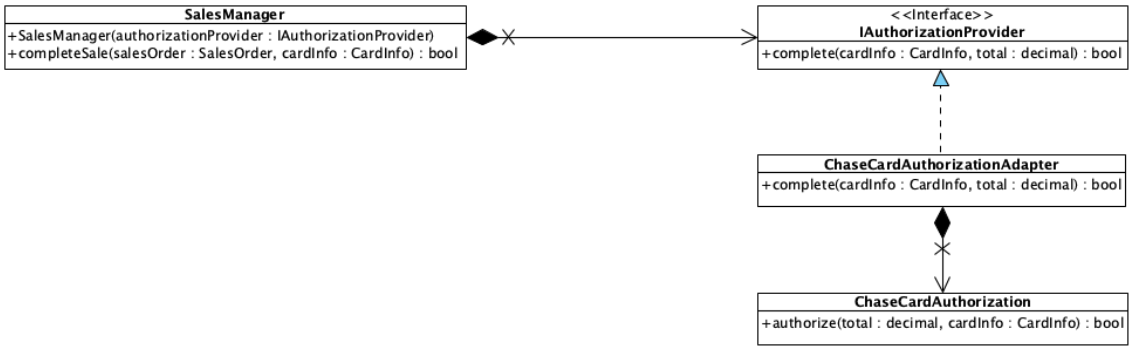
- Our *SalesManager* used an authorization provider to charge credit cards
- It is bad to instantiate it in the *SalesManager*; better to use dependency injection
- More than that, we can change what is injected when management changes the authorization provider

6.2.4 Inject an Interface Realization



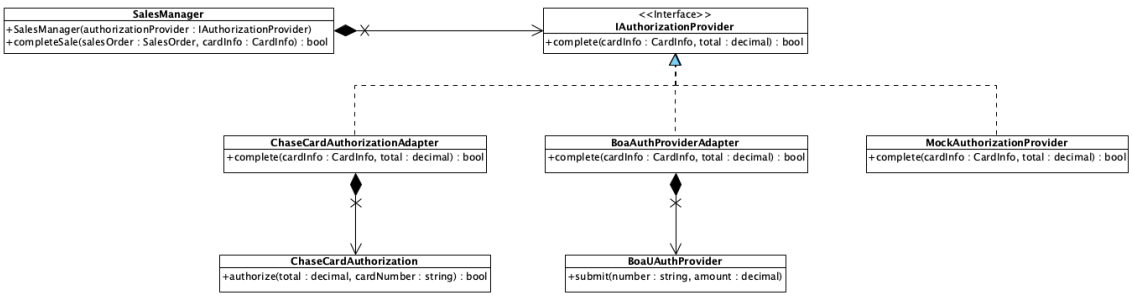
- Inject a realization of an inteface to swap it out
- For third-party code, we need to adapt from the interface to the code
- For testing we can inject a mocked-up class with consistent results

6.2.5 Injecting the Provider



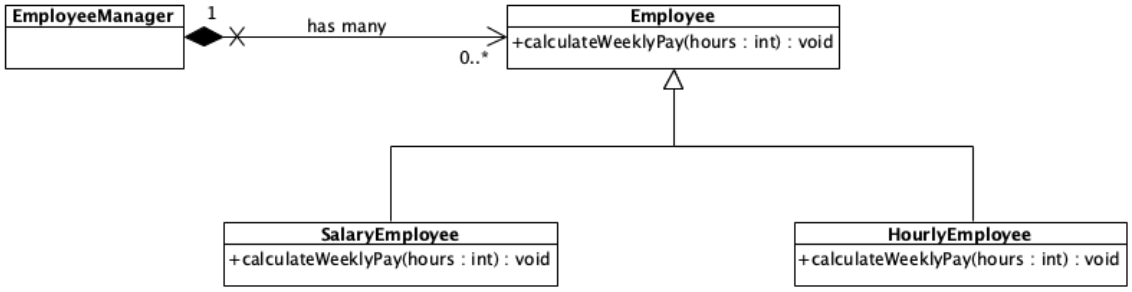
- The provider cannot be injected directly, wrong interface
- Create a class that translates

6.2.6 The Adapter Pattern



- Use dependency inject to pass the authorization object to Sale
- Follow Open/Closed to create new types of authorization objects

6.2.7 Strategy Pattern



- Use the *strategy pattern* to inject different classes with logic
- *if* and *switch* violate Open/Closed, put the logic into classes and extend
- *Strategy* has a similar pattern to *adapter*, just without the third-party code

6.2.8 Factory Method Pattern

```
public class SalesManager {

    public SalesManager() {

        GetAuthorizationProvider();
    }

    private GetAuthorizationProvider {

        // instantiate the authorization class
    }
}
```

- Isolate creation in a *factory method* away from the other methods in the class
- The factory method violates *Single Responsibility* when it is in the same class
- All the benefits of dependency injection are lost

6.2.9 Factory Class

```
public class ConfigurationFactory {

    public ConfigurationFactory() {

        // Read an externalized configuration to find the adapter
    }

    public GetAuthorizationProvider {

        // instantiate and return the correct adapter
```

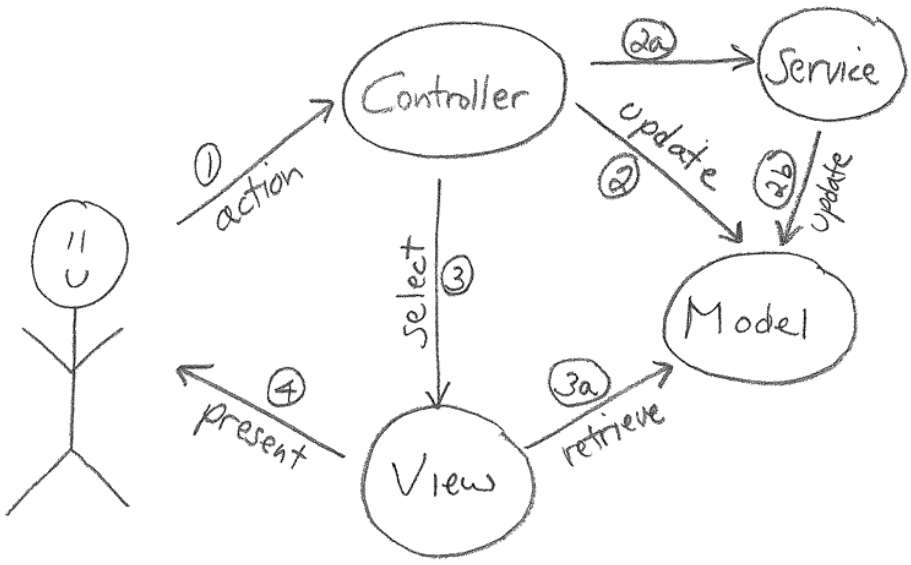
```
    }
}
```

- Putting component creation in a factory class and using dependency injection is the next level
- That follows Craig Larman's creator, information expert, and pure fabrication patterns

6.2.10 Inversion of Control

- IoC containers are factories that are configurable with lots of class instances to be injected when required
- .NET Core and Unity provide an IoC container and handle the injection

6.2.11 Model-View-Controller



- MVC is an architectural pattern, it defines the flow between components
- The actor is not necessarily a person, it could be another component (the client to a web service?)
- The view is a component that *produces* output, not the visual that a user sees as the result of a request
- Controllers have moved to delegating business rules to application services (single responsibility)

6.2.12 Checkpoint

- Why are the design principles so important?
- What does dependency injection accomplish?
- How does dependency injection work?
- What does Barbra Liskov say about substitution?
- What is the significance of the Open/Closed principle?
- Why UML?

6.3 Lab - Adapter

Design Principles
Design Patterns
Lab - Adapter Pattern

Goals

- Use TDD to extend the *SalesManager* to complete a sale.

Synopsis

The purpose of this exercise is to reinforce TDD and the importance of design principles and patterns. Management has changed merchant service providers in the past, and is likely to do so again in the future.

Requirements

- The company has two merchant service accounts, one with "The Bank of Random Credit", and another with "Everyone is Authorized". Look for their client classes in the project dependencies.
- The project requires the *SalesManager* to use an *IPaymentService* to complete a sale.
- The *IPaymentService* will have a method *string AuthorizePayment(CardInfo cardInfo, decimal amount);*.
- An adapter class is required to translate between the IPaymentService interface and the two merchant service accounts.
- Authorization takes place *after* the card info and total have been verified, to avoid passing bad information to the merchant services account.
- *AuthorizePayment* returns a string authorization code. Failure to authorize is expected and not an exception, null is returned.
- Create tests to instantiate each adapter and test the SalesOrderManager class.

Project Steps

1. Create the unit tests to instantiate the adapters, inject a SalesOrderManager, and test the class.
2. Build the interface, adapter classes, and refactor the SalesOrderManager to satisfy the test.
3. Can you complete all of the tests? If not, how many of the tests you identified can you complete?

Summary

We reinforced using TDD when we added yet another feature to the project. By now you should see how the design principles and patterns support the evolution of the design with unit testing, TDD, and Agile. And, we discovered that unit testing easily becomes unatainable if dependency injection is not realized.

Congratulations, you have completed this lab!

Chapter 7
Test Doubles (a.k.a. Mocks)

Test Doubles
Moq
Lab - Moq

Objectives

- Define test doubles, and dummies, fakes, stubs, spies, and mocks
- Use Moq as a framework for creating focused test doubles

Overview

Unit tests should be performed in isolation, dependencies should be decoupled. Integration and system tests are performed with dependencies. To test without dependencies, *test doubles* are used to stand in place of them.

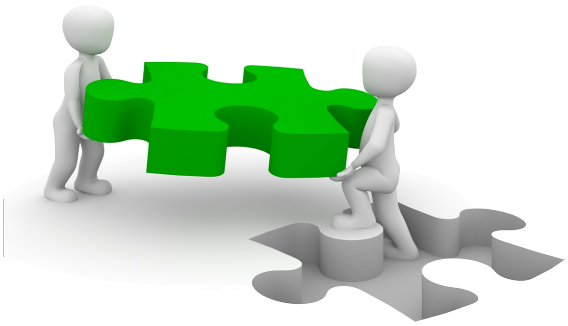
7.1 Test-Doubles

Test-Doubles
Moq Framework
Lab - Test-Doubles and Moq

7.1.1 What are Test-Doubles?

- Unit tests should be performed in isolation and test a process, not involve its dependencies
- A test-double is something that stands in for a dependency on another object during a test
- Tests need precise and consistent results, test-doubles provide this while real dependencies often do not

7.1.2 Always use Test-Doubles?



- Should test-doubles always be used? It depends on your point of view!
- One group argues that all tests should be done in complete isolation without exception
- The other group feels that if the dependency is stateless and consistent it is OK to use it

7.1.3 Types of Test-Doubles

Test Double Type	Description
Dummy	An object that just stands in for something else, often as a parameter filler and not actually used
Fake	A double that is a full implementation but consistent across tests, like a copy of a database to use for testing
Stub	A double that provides a consistent answer for specific circumstances, but does not work outside of those circumstances
Spy	A stub that records information about how and when it was called, suitable when there is a requirement to test that things are called in the correct sequence
Mock	Usually programmed through a framework, mocks provide a consistent result, throw exceptions if not called correctly, and can be checked to make sure they were actually called and called correctly

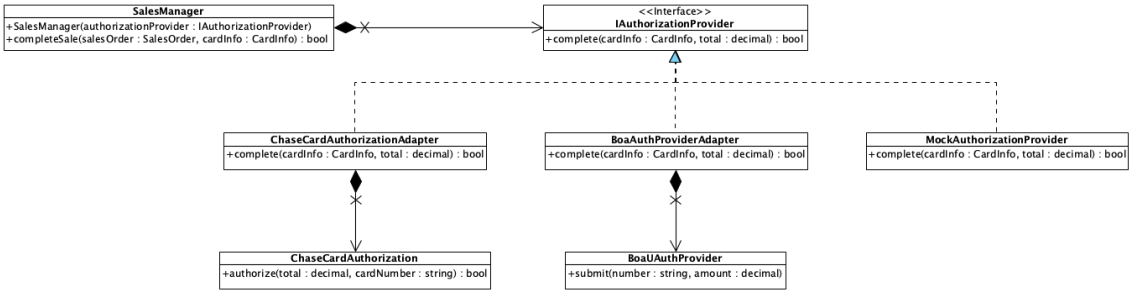
Adapted from Gerad Meszaros' test-double types on Martin Fowler's 1/17/2006 blog post "Test Double" @https://martinfowler.com/bliki/TestDouble.html.

7.1.4 Creating Test-Doubles

```
public class AuthorizationProviderStub : IAuthorizationProvider {

    public string Complete(CardInfo cardInfo, Decimal total) {

        string result = null;

        if (cardInfo.CardNumber == "") {

            result = Guid.NewGuid();

        } else if (cardInfo.cardNumber != "378282246310005") {

            throw ArgumentOutOfRangeException("Unexpected card number");
        }

        return result;
    }
}
```

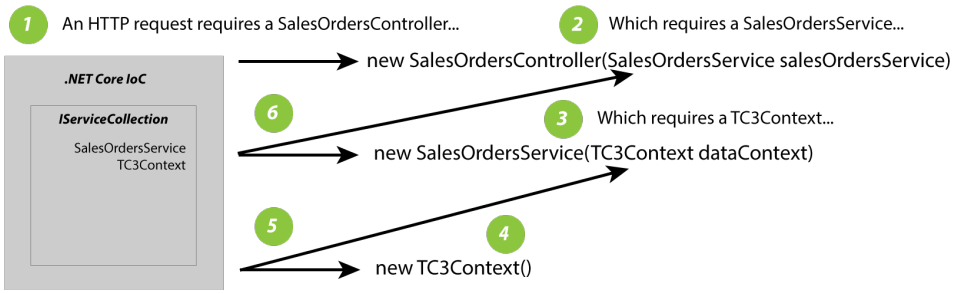
- Make a copy of a dependency, such as a copy of a database
- Create a new class that provides expected, consistent results to stand in for a dependency
- Leverage easy-to-use frameworks that provide test-doubles to stand in for dependecies
- *Dependency Injection* is the key to using test-doubles

7.1.5 Dependency Inversion (SOLID Principle #5)



- Simply put: client code should depend on the most abstraact version of a type possible
- *new* is evil, and client code depending on an *interface* (the language construct) cannot use *new* anyway
- The solution is to design client code to be passed any dependencies it needs: (*dependency injection*)
- Follow this and inject application constructs, but not data structure dependencies like a List<>

7.1.6 .NET Core Inversion of Control (IoC)



- What code instantiates the dependency that client code needs?
- Move the decision to the code that instantiates the client, it should know what the client needs
- Use an inversion of control *container* or *framework* to set the dependency when the client is instantiated
- IoC frameworks (Spring, Unity, etc.) typically control the instantiation of all objects in the program

7.1.7 Dependency Injection

```
public class SalesOrdersService {

    public SalesOrdersService(TC3Context dataContext) {

        DataContext = dataContext;
    }

    public TC3Context DataContext { get; set; }
}
```

- Implementing dependency injection supports IoC, but is also critical for testing code

- Three ways to insert a dependency into an object: *constructor injection*, *property injection*, and *reflection*
- *Reflection* allows runtime inspection of a class/method definition to decide how to pass in a dependency
- .NET Core specifically uses constructor injection

Acknowledgements

Original images are the property of NextStep IT Training.
Other images in this chapter are licensed for commercial use from https://pixabay.com.
Figure 2 (stick-figures and puzzle piece) by Peggy and Marco Lachmann-Anke.

7.2 Moq

Test Doubles
Moq
Lab - Moq

7.2.1 The Moq Framework

```
using Moq;
...
var moqCalculator = new Mock<ICalculator>();

moqCalculator.Setup(mock => mock.Radix)).Returns(10);
moqCalculator.Setup(mock => mock.Add(1, 2)).Returns(3);
```

- A mock framework defines the responses to specifid properties and methods in the current context
- Moq definitions may be inline in the test: simpler and cleaner than creating a test-double class
- Note: there may be more runtime overhead than using a test-double brought in at compile time

7.2.2 Moq and Interfaces

```
public interface ICalculator {

    int Radix { get; set; }
    int Add(int a, int b);
    ...
}
```

```
[Fact]
public void AddNumbersInRangeSuccess() {

    var moqCalculator = new Mock<ICalculator>();

    moqCalculator.Setup(mock => mock.Add(1, 2)).Returns(3);

    Assert.Equal(3, moqCalculator.Object.Add(1, 2));
    // Assert.Equal(3, moqCalculator.Object.Add(2, 1)); // Fails, not defined!
}
```

- Any or all of the interface may be mocked, it is only necessary to mock what will be used
- Moq is one reason so many interfaces are defined in libraries and application code

7.2.3 Moq and Classes

```
public class Calculator {

    public virtual int Add(int a, int b);
    public int Subtract(int a, int b);
    ...
}
```

```
[Fact]
public void AddNumbersInRangeSuccess() {

    var calc = new Mock<Calculator>();

    moqCalculator.Setup(m => m.Add(1, 2)).Returns(3);
    // moqCalculator.Setup(m => m.Subtract(3, 2)).Returns(1); // Fails!

    Assert.Equal(3, moqCalculator.Object.Add(1, 2));
}
```

- Moq can mock classes, but only virtual properties and methods in public classes, and only the get side of virtual properties

7.2.4 Mocking Methods and Results

```
moqCalculator.Setup(m => m.Add(1, 2)).Returns(3);
moqCalculator.Setup(m => m.Add(-1, 2)).Throws(new ArgumentException());
moqCalculator.Setup(m => m.Radix = 10));
```

- Three possible results: *value*, *exception*, or *void*
- void doesn't require any work

7.2.5 Matching Parameters to Setup Definitions

```
SalesOrder salesOrderA = new SalesOrder();  // Equals is not defined
SalesOrder salesOrderB = salesOrderA;
SalesOrder salesOrderC = new SalesOrder();

var moqSalesOrderManager = new Mock<SalesOrderManager>();

moqSalesOrderManager.Setup(m => m.Checkout(salesOrderA)).Returns("A");

string r1 = moqSalesOrderManager.Object.Checkout(salesOrderA);  // "A"
string r2 = moqSalesOrderManager.Object.Checkout(salesOrderB);  // "A"
string r3 = moqSalesOrderManager.Object.Checkout(salesOrderC);  // null
```

- Moq matches parameters to mocked methods by *equality*, not type
- Primitive values (int, double, etc.) work they way you think they will
- Calling a method without using Setup returns the default value for the method return type

7.2.6 Matching and Equality

```
SalesOrder salesOrderA = new SalesOrder();  // Equals is overridden
SalesOrder salesOrderB = salesOrderA;
SalesOrder salesOrderC = new SalesOrder();

var moqSalesOrderManager = new Mock<SalesOrderManager>();

moqSalesOrderManager.Setup(m => m.Checkout(salesOrderA)).Returns("A");

string r1 = moqSalesOrderManager.Object.Checkout(salesOrderA);  // "A"
string r2 = moqSalesOrderManager.Object.Checkout(salesOrderB);  // "A"
string r3 = moqSalesOrderManager.Object.Checkout(salesOrderC);  // "A"
```

- Moq uses the *Equals* override to check for Equality
- Moq does not use the == operator, even if overridden

7.2.7 Matchers for Flexibility

```
// any value
mock.Setup(foo => foo.DoSomething(It.IsAny<string>())).Returns(true);

// any value passed in a 'ref' parameter:
mock.Setup(foo => foo.Submit(ref It.Ref<Bar>.IsAny)).Returns(true);

// matching Func<int>, constrained by the lambda (evaluated lazily)
mock.Setup(foo => foo.Add(It.Is<int>(i => i % 2 == 0))).Returns(true);

// matching ranges
mock.Setup(foo => foo.Add(It.IsInRange<int>(0, 10, Range.Inclusive))).Returns(
    ↪ true);

// matching regex
mock.Setup(x => x.DoSomethingStringy(It.IsRegex("[a-d]+", RegexOptions.
    ↪ IgnoreCase))).Returns("foo");
```

- The matcher *It* allows for flexible definitions, anything matching the matcher is accepted
- Four forms: any value, matches if a function returns true, matches a range, a regular expression matches

7.2.8 Verify (Spy)

- Use the moq object *Verify* method to spy on what happened

7.2.9 Moq and Async Methods

7.2.10 Property Get and Set

```
moqCalculator.Setup(m => m.Radix)).Returns(10);

int radix = moqCalculator.Object.Radix; // 10

moqCalculator.SetupSet(m => m.Radix = 2)
moqCalculator.VerifySet(mock => m.Radix = 2)     // Note the assignment operator
```

- Properties are matched the same way methods are, the *Returns* provides the value
- Properties do not have parameters, so there is only one way to call them

7.2.11 Properties and Matchers

7.2.12 Property Stub

```
moqCalculator.SetupProperty(m => m.Radix);
moqCalculator.SetupProperty(m => m.Radix, 10);  // With a default value

moqCalculator.Object.Radix = 2;
Assert.Equal(2, moqCalculator.Object.Radix);
```

- The property stub manages the assigned value, so it can be assigned any value

7.2.13 Moq Quickstart

- This has just been a taste of Moq
- The full definitions are at https://github.com/Moq/moq4/wiki/Quickstart

7.2.14 Checkpoint

- Why use test doubles?
- What are Dummies, Fakes, Stubs, Spies, and Mocks?
- Why is it easier to mock an interface?

7.3 Lab - Moq

Test Doubles
Moq
Lab - Moq

Goals

- Refine unit tests with Moq

Synopsis

Test in isolation! Replace the dependencies on the card validation and authorization provider with test doubles.

Requirements

Part 1

- Add dependency injection to set the CardValidator dependency in the SalesOrderManager.
- Use a test-double class to stand in for an authorization adapter that provides expected results.

Part 2

- Use Moq to set up and test the SalesOrderManager class.

Project Steps

1. Set up dependency injection to push a CardValidator into the SalesManager class from the tests.
2. Create a new folder *mocks* in the test project.
3. In the mocks folder create a new test-double adapter class that provides consistent results for good and bad CardInfo instances.
4. Fix the tests to use the new test-double and check the results.
5. Use the NuGet project manager to add the Moq framework to the test project.
6. Create a mock object for the adapter, set up the method results, and use it to test the SalesOrderManager.
7. Make sure all unit tests still work after the refactoring.

Summary

Cleaner code, now the classes are tested in isolation.

Congratulations, you have completed this lab!

Chapter 8
Moq and Data Sources

Entities and Web Services
Moq and Entity Framework
Moq and MongoDB
Lab - Mock and Entity Framework

Objectives

- Explore mocking the Entity Framework

Overview

When tests need to be run in isolation, Moq can be used to create test doubles to stand in for the Entity Framework. In this case Moq is used with the DBSet collections that manage entities and the DBContext classes that manage the collections. Moq can also be used to mock NoSQL databases like MongoDB.

8.1 Entities and Web Services

Entities and Web Services
Moq and Entity Framework
Moq and MongoDB
Lab - Mock and Entity Framework

8.1.1 Entities

- Entities represent data from a persistent data source, often relational databases or NoSQL sources
- Entity classes are plain-old classes that define the data in an entity
- Entity classes map to table rows in RDBMS or documents in NoSQL sources

8.1.2 Entity Security and Data Transfer Objects

- Many table rows or NoSQL documents contain sensitive information
- Data Transfer Objects (DTOs) are used to prevent the leakage of sensitive information
- Inbound information can move freely, but outbound information can be restricted

8.1.3 Data Transfer Objects

- DTOs are used to carry information between user interfaces and business logic
- DTOs are used to carry information between clients and web service controllers

8.1.4 Designing Data Transfer Objects

```
public class SalesOrderEnabledDto : SalesOrder {

    public SalesOrderEnabledDto() {
    }

    public SalesOrderEnabledDto(SalesOrder salesOrder) : base(salesOrder) {
    }
}

public class SalesOrderRestrictedDto : SalesOrder {

    public SalesOrderRestrictedDto() {
    }

    public SalesOrderRestrictedDto(SalesOrder salesOrder) : base(salesOrder) {

        CardNumber = null;
        CardExpires = null;
```

```
        }
}
```

- Extend the model class into an DTO class, duplicating and extending the constructors
- Create "enabled" and "restricted" versions of the DTO class to pass or filter information

8.2 Mocking Entity Framework

Mocking Entity Framework
Lab - Mocking Entity Framework

8.2.1 Mocking Entity Framework

- Mocking the Entity Framework is much faster than using the database, and allows the client functionality to be checked
- The mock may be used to make sure update data is successfully passed in
- Consistent data may be provided to see if Linq queries work correctly

8.2.2 Mocking DBContext

```
public partial class KaleidoscopeCruiseLinesEntities : DbContext
{

    public KaleidoscopeCruiseLinesEntities()
        : base("name=KaleidoscopeCruiseLinesEntities")
    {
    }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {

        throw new UnintentionalCodeFirstException();
    }

    public virtual DbSet<Passenger> Passengers { get; set; }
}
```

- EF has three pieces, a DBContext, a DBSet, and the entity (model) classes
- The *DBContext* class may be mocked because the DBSet property is virtual; we can ignore set and return anything that we like
- So, set up what the *DBSet* collections are to return when the *DBContext* is mocked

8.2.3 Mocking DBSet

```
[Fact]
public void AddPassenger() {

    var passengers = new ArrayList<Passenger>().AsQueryable();
    var passengersSet = new Mock<DbSet<Passenger>();

    passengersSet.As<IQueryable<Passenger>>().Setup(m => m.Provider).Returns(
        ↪ passengers.Provider);
    passengersSet.As<IQueryable<Passenger>>().Setup(m => m.Expression).Returns(
        ↪ passengers.Expression);
    passengersSet.As<IQueryable<Passenger>>().Setup(m => m.ElementType).Returns(
        ↪ passengers.ElementType);
    passengersSet.As<IQueryable<Passenger>>().Setup(m => m.GetEnumerator()).
        ↪ Returns(passengers.GetEnumerator());

    Passenger p = new Passenger();

    passengersSet.Setup(m => m.Add(p));

    var kclContext = new Mock<KaleidoscopeCruiseLineEntities>();

    kclContext.Setup(x => x.Passengers).Returns(passengersSet.Object);
}
```

- Create an empty list of passengers as *IQueryable*, mock the set operations to use it, and create a passenger for mocking Add
- Add the *DBSet* mock to the *context* mock as what it returns

8.2.4 Using the Mock

```
[Fact]
public void AddPassenger() {
    ...
    var kclContext = new Mock<KaleidoscopeCruiseLineEntities>();

    kclContext.Setup(x => x.Passengers).Returns(passengersSet.Object);
    var passengersService = new PassengersService(kclContext.Object);

    p.Id = 1;
    passengersService.Add(p);
    Assert(1, passengers[0].Id);
}
```

8.2.5 Going Further

- This covered most of what we need; we can step in for Entity Framework and serve up a collection of objects, and we can step in and intercept modifications to that set of objects
- A more comprehensive tutorial may be found at https://www.jankowskimichal.pl/en/2016/01/mocking-dbcontext-and-dbset-with-moq/

8.2.6 Checkpoint

- Why mock the Entity Framework?
- What pieces of the framework must be mocked?
- Why are so many methods mocked for the DBSet?

8.3 Lab - Mocking Entity Framework

Mocking Entity Framework
Lab - Mocking Entity Framework

Goals

- Get Entity Framework out of the picture, just mock it up!

Requirements (revisited)

- No new requirements

Project Steps

1. Build mocks that provide or accept the data in place of using Entity Framework.
2. Run the tests and prove that everything still works.

Summary

Cleaner code. Entity Framework and databases slow down tests drastically, they can add hours to unit and integration tests. Especially integration tests. So, anything we can do to keep them out of the picture we want to do!

Congratulations, you have completed this lab!

Chapter 9
Testing Web Services

Testing Web Services
Lab - Web Services

Objectives

- Explore the framework for testing web services
- Test web services without using HTTP

Overview

We assume that the HTTP server the web service is deployed to will provide the correct interface for HTTP. Our task is to test the functionality of the web service, and we can do that by calling the methods directly.

9.1 Testing Web Services

Testing Web Services
Lab - Web Services

9.1.1 Testing Web Services

- A web service method returns: *void*, *HttpResponseMessage*, *IHttpActionResult*, or something else (such as *IEnumerable<T>* or *IQueryable<T>*)
- We have to know what to expect when we use a method
- Skip HTTP and go direct for unit tests, testing through HTTP would be an E2E test

9.1.2 Setting the Stage

```
[fact]
public void GetReturnsThePassengerList() {
    ...
    var service = new Mock<PassengerService>();

    service.Mock(mock => mock.GetPassengers().Return(passengers));
    var controller = new PassengersController(service.Object);
}
```

- This example looks at a controller that uses a service, none one that drives the DBContext directly
- Create a mock service if necessary, then initialize the controller
- To test a controller it may be necessary to add a constructor to allow the injection of the service

9.1.3 Testing Input

```
[ResponseType(typeof(Passenger))]
public IHttpActionResult GetPassenger(int id)
{
```

```
[Fact]
public void GetReturnsThePassenger() {
    ...
    var result = controller.GetPassenger(1) as OkNegotiatedContentResult<
        ↪ Passenger>;
```

- Testing input is straightforward, just pass the input to the controller method
- Do not worry about passing JSON or XML through HTTP, if GetPassenger wants an int then call it with an int

9.1.4 GET and IEnumerable/IQueryable

```
[Fact]
public void GetReturnsThePassengerList() {
    ...
    var results = controller.GetPassengers() as IQueryable<Passenger>;

    Assert.NotEqual(null, results);
}
```

- Controller method returns IEnumerable<T> or IQueryable<T>
- Cast the result into the expected type; this example uses *as* which returns null on failure

9.1.5 GET and HttpResponseMessage

```
[Fact]
public void GetReturnsThePassenger() {
    ...
    var result = controller.GetPassenger(1) as HttpResponseMessage;

    Assert.Equal("Smith", result.Content.LastName);
}
```

- Give a response message, the data (object) returned is in the *Content* property

9.1.6 Get and IHttpActionResult

```
[Fact]
public void GetReturnsThePassenger() {
    ...
    var result = controller.GetPassenger(1) as OkNegotiatedContentResult<
        ↪ Passenger>;

    Assert.NotEqual(null, result);
    Assert.Equal("Smith", result.Content.LastName);
}
```

- IHttpActionResult was introduced in Web API 2; it simplifies creating the HttpResponse, and it simplifies testing
- But, IHttpActionResult must be converted into a response by the test
- The best way is to cast it to *OkNegotiatedContentResult* (use *as*)

9.1.7 Get and *void*

- A controller method returning void is pesky, you cannot see the result of calling the method
- So, just test for failure (exceptions) or go around it and use the mock to see if the method called the mock

- Or, not very clean, modify the class to allow peeking

9.1.8 Http Status Code

```
[Fact]
public void GetReturnsThePassenger() {
    ...
    var result = controller.GetPassenger(0) as OkNegotiatedContentResult<
        ↪ Passenger>;

    Assert.NotEqual(null, result);
    Assert.Equal(404, result.StatusCode);
}
```

- With *IHttpActionResult* it is more normal to call a method like *NotFound* from *APIController* which will set the response status code
- Check the status code of the response after calling *OkNegotiatedContentResult*

9.1.9 Http Status Code

```
[Fact]
public void GetReturnsThePassenger() {
    ...
    var result = controller.GetPassenger(0)

    Assert.Equal(TypeOf(NotFountResult), result);
}
```

- Or avoid the cast and the first Assert by looking at the *IHttpActionResult* directly

9.1.10 Exceptions

```
[Fact]
public void GetReturnsThePassenger() {
    ...
    Assert.Throws<HttpResponseException>(() => controller.GetPassenger(0));
}
```

- An HttpResponseException caught in the controller is turned into an error in the HTTP protocol
- But that doesn't happen in the direct call to the controller method, so look for the exception in the test

9.1.11 Checkpoint

- Why do we avoid using HTTP for unit testing the web service?

- Is there anything special about calling a web service method?
- How about handling the result of a web service method?
- Why is *as* preferable as the type casting mechanism?
- What does OKNegotiatedContentResult do?
- How are expected errors handled in the unit test?

9.2 Lab - Web Services

Testing Web Services
Lab - Web Services

Goals

- Test the controller classes.

Synopsis

Requirements (revisited)

- There are no new requirements.

Project Steps

1. Build the tests for the controller classes.
2. Oh yeah, you need to use TDD to build the controller class for bookings!

Summary

Hopefully we have the controllers set and tested.

Congratulations, you have completed this lab!