Xpirit

Deepdive Unit Testing in .NET

Day 1

Xpirit

Agenda Day 1

- Check-in
- Unit testing 101
- .NET Unit Test Frameworks
- Test Driven Development
- Exceptions
- Testable code
- Mocking dependencies
- Checkout

Format

- Slides
- Demos
- Hands-On-Labs
  → TDD katas
  → Pair programming
  → Ask for help
  → Central review afterwards

Unit Testing 101

Unit test definition

- A piece of code that invokes a unit of work in the system and then checks a single assumption about the behavior of that unit of work
- Test code verifying application code

Xpirit

Impact of unit testing

Functional design

- Requires (re)formulating functionality in testable way
- Exposes flaws and holes in design
- Uncovers invalid and implicit assumptions

Technical design

- Simpler class hierarchies
- Component oriented
- Evolves through refactoring

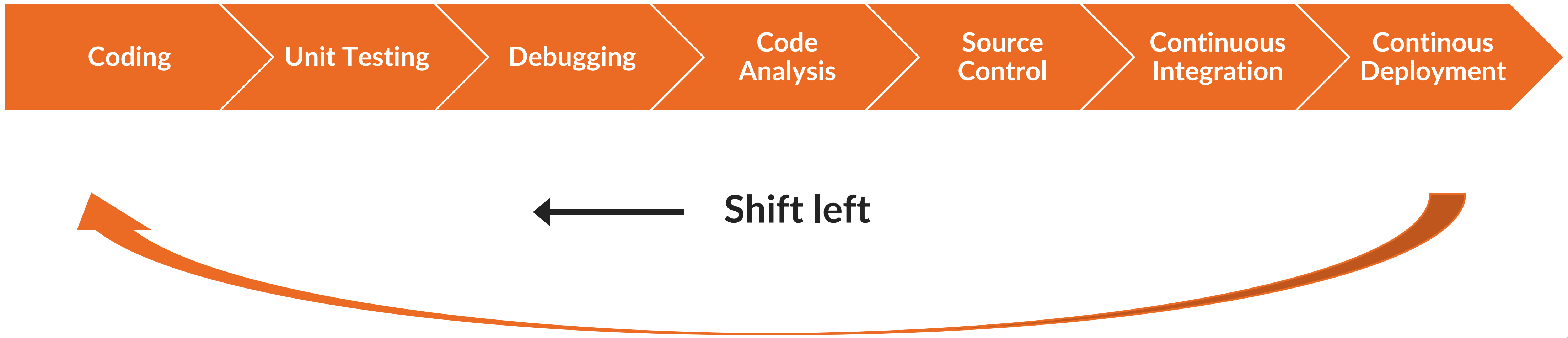
Coding

- Less code
- Smaller functions
- Less conditional code
- A lot more testing code

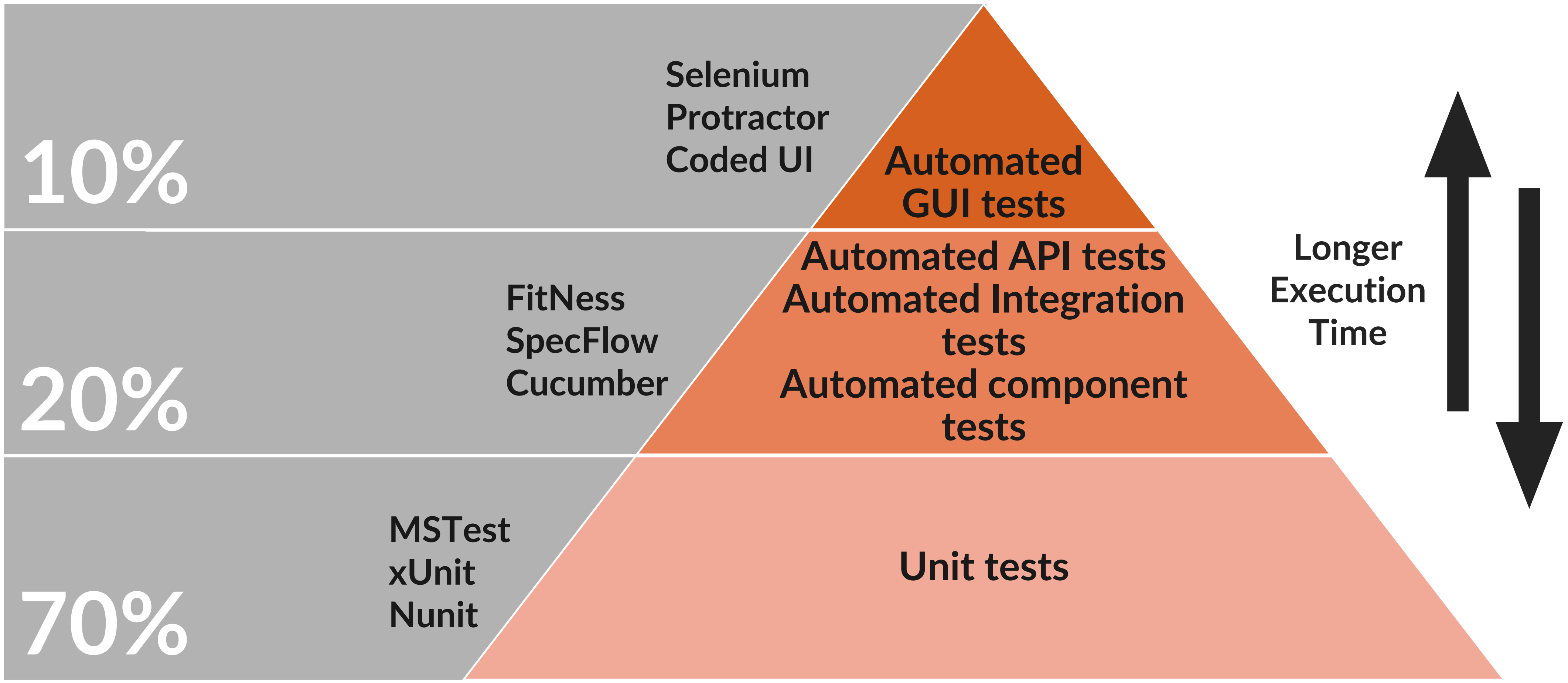
Bugs and debugging

- Bugs are more easily reproduced
- Debugging becomes simpler
- Solving bugs requires mostly local code changes

Unit testing in the Software Development Lifecycle



Testing Pyramid



Xpirit

A good unit test is

- Independent
  - →No shared state
  - →No order between tests
  - →No external dependencies
- Consistent
- Fast to execute
- Readable
- Maintainable
- Trustworthy

Naming convention

Use a clear naming convention for unit tests such as

```
[UnitOfWork]_[Scenario]_[ExpectedBehaviour]
```

Examples

```
RegisterUser_WithValidUser_ShouldReturnUserId

RegisterUser_WithInvalidUser_ShouldThrowInvalidUserException
```

Xpirit

BDD style naming convention

Alternative Behaviour Driven Development style

```
[GivenPreconditions_WhenStateUnderTest_ThenExpectedBehavior]
```

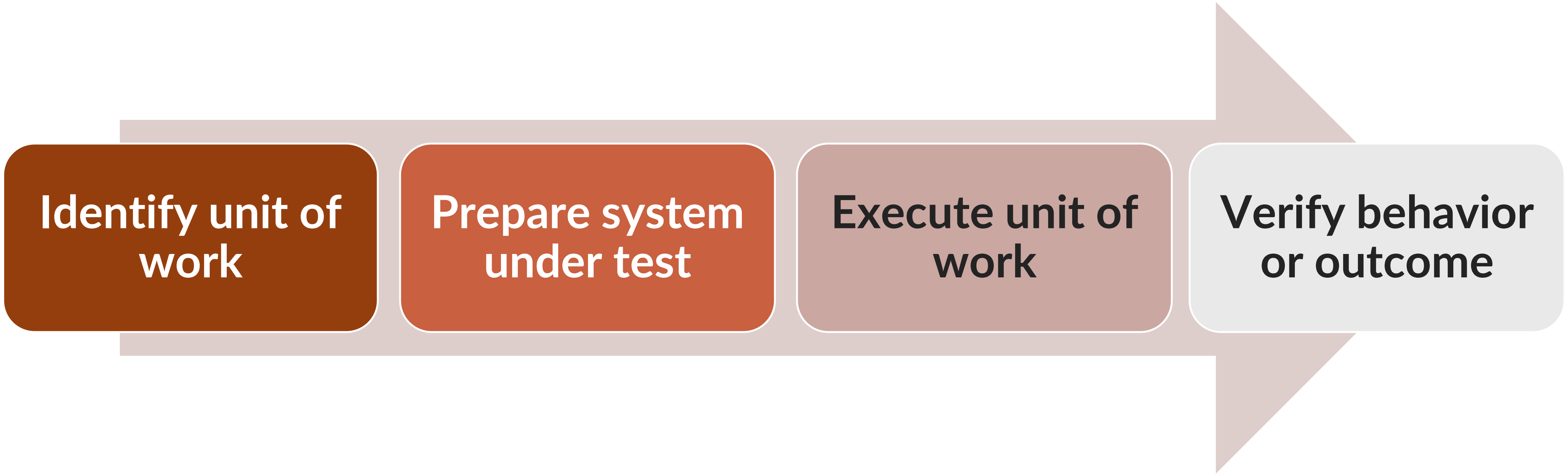
Examples

```
GivenAValidUser_WhenUserIsRegistered_ThenReturnUserIdShouldBeReturned

GivenAnInvalidUser_WhenUserIsRegistered_
ThenAnInvalidUserExceptionShouldBeThrown
```

Xpirit

Unit Test Responsibility



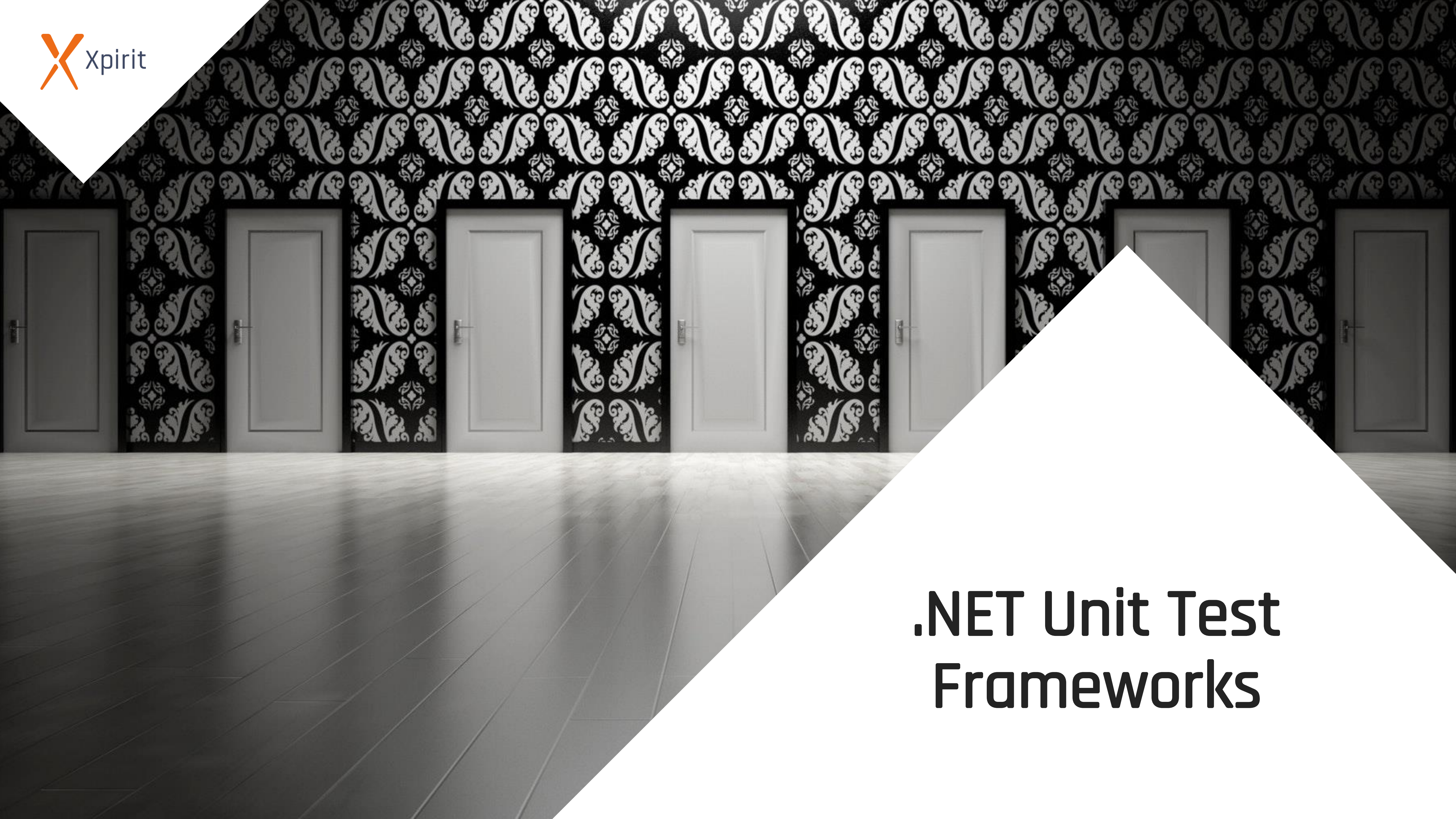
Xpirit

Unit test structure: AAA

```
// Arrange
var registrationService = new RegistrationService();
var newUser = new User(“Grace Hopper”, “grace@hopper.org”)

// Act
var result = registrationService.RegisterUser(newUser);

// Assert
Assert.NotNull(result, “It is expected that the result is the
registered user.”);
```



.NET Unit Test Frameworks

Xpirit

.NET Unit Test Frameworks

- Provides an API to structure tests
- Assert outcomes
- Examples:
  → MSTest
  → NUnit
  → xUnit

Xpirit

Unit Test Framework Attributes

	MSTest	NUnit	xUnit
Identify test class	[TestClass]	[TestFixture]	-
Identify test method	[TestMethod]	[Test]	[Fact]
Parameterized test method	[DataTestMethod]	[TestCase]	[Theory]
Run before every test in a class	[TestInitialize]	[SetUp]	-
Run after every test in a class	[TestCleanup]	[TearDown]	-
Run once before any tests are run in a class	[ClassInitialize]	[OneTimeSetUp]	-
Run once after all tests are run in a class	[ClassCleanUp]	[OneTimeTearDown]	-

Frameworks & Libraries

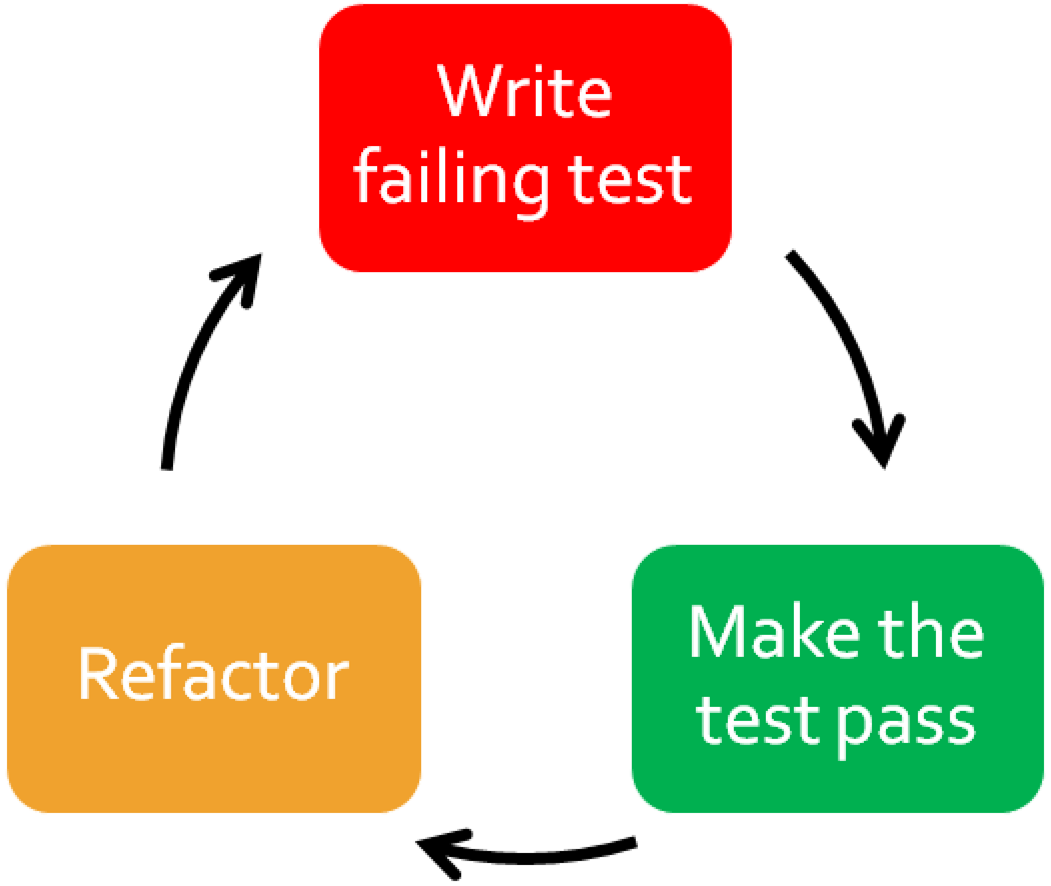
Unit testing	Mocking	Arrange	Assert
• MSTest • Nunit • xUnit	• NSubstitute • Moq • Nmock • FakeItEasy	• AutoFixture	• FluentAssertions • Shouldy

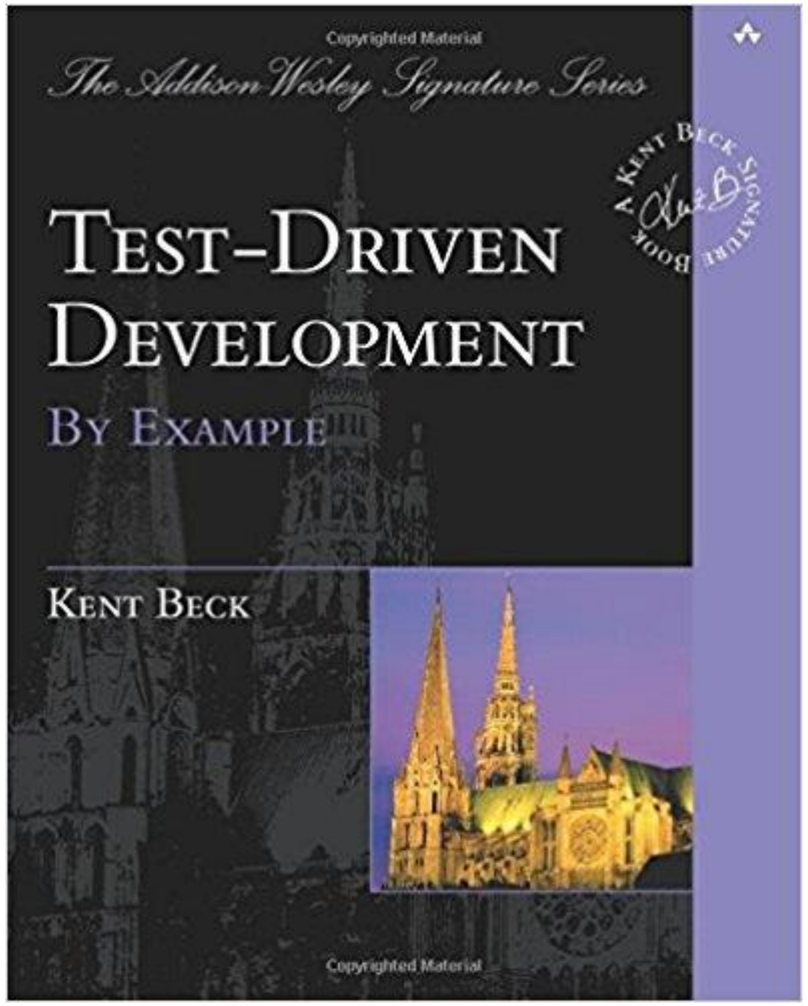
DEMO

- Unit testing with xUnit
- Assertions

Test Driven Development

Test Driven Development




Copyrighted Material

The Addison-Wesley Signature Series

A Kent Beck Signature Book

TEST-DRIVEN DEVELOPMENT

BY EXAMPLE

KENT BECK

Copyrighted Material

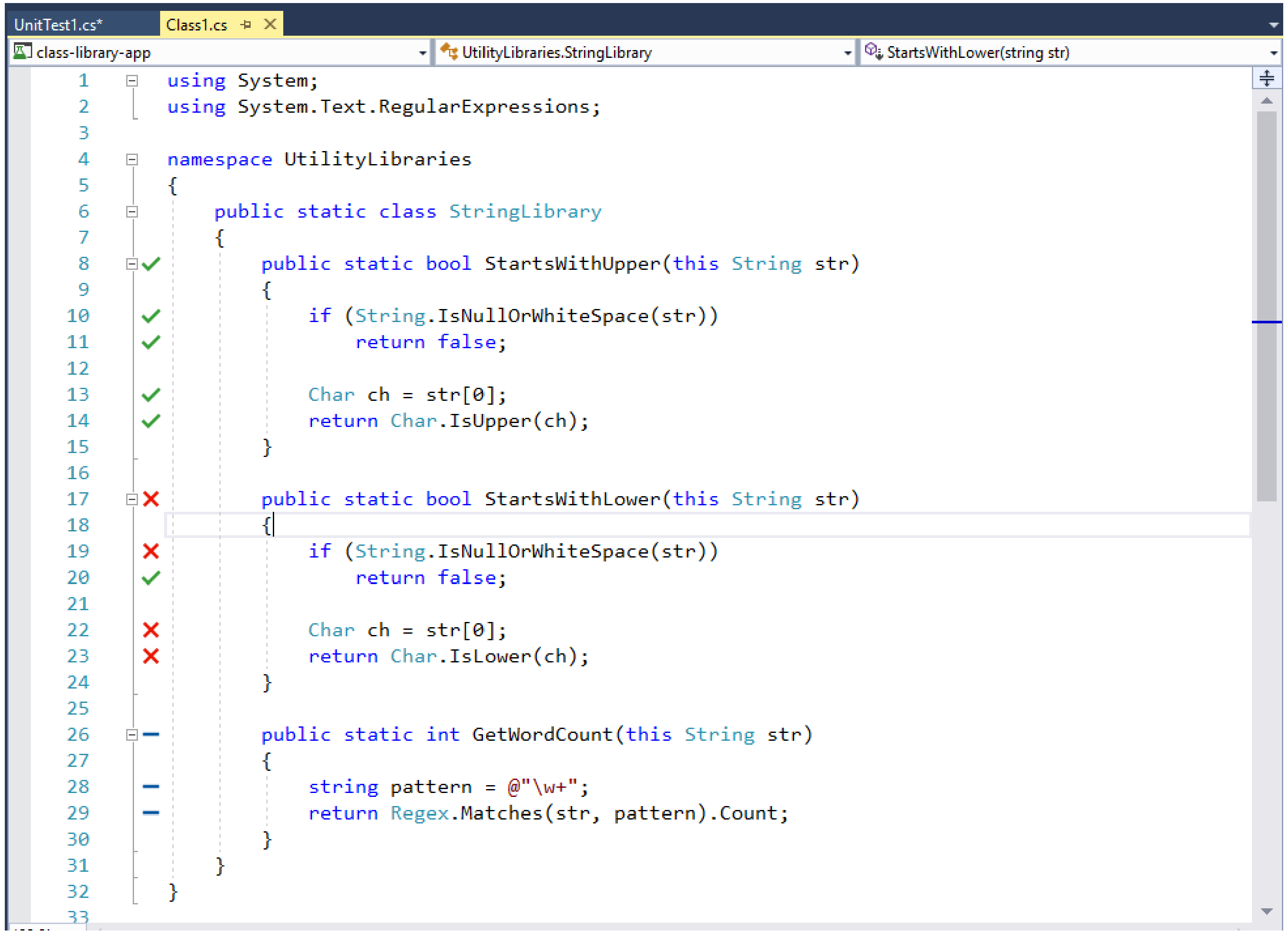
TDD

- **Write the minimum amount of code in order to pass the test.**

→ Having a small unit of work makes the debugging easier

→ Makes test code easier to read & understand

Live Unit Testing in VS2017 Enterprise



```
using System;
using System.Text.RegularExpressions;

namespace UtilityLibraries
{
    public static class StringLibrary
    {
✓       public static bool StartsWithUpper(this String str)
        {
✓           if (String.IsNullOrWhiteSpace(str))
✓               return false;

✓           Char ch = str[0];
✓           return Char.IsUpper(ch);
        }

✗       public static bool StartsWithLower(this String str)
        {
✗           if (String.IsNullOrWhiteSpace(str))
✓               return false;

✗           Char ch = str[0];
✗           return Char.IsLower(ch);
        }

–       public static int GetWordCount(this String str)
        {
–           string pattern = @"\w+";
–           return Regex.Matches(str, pattern).Count;
        }
    }
}
```

TDD kata

“A kata is a form of *deliberate practice*, with its roots in the martial arts world. It describes a choreographed pattern of movements used to *train yourself* to the level of *muscle memory*.

In the world of programming, katas are small coding exercises that a programmer completes on a daily basis. ”

Xpirit

Hands-on-Labs Solution & Docs

- Clone repo: https://github.com/XpiritBV/UnitTesting
- Read tdd_katas/tdd_kata_start.md
- Open the Xpirit.UnittestingWorkshop.sln

Xpirit

Hands-on-Labs

- **Perform TDD Kata 1– Player (15 min timebox)**
  → Pair programming
  → Ask for help
  → Central review afterwards

Hands-on-Labs

- Perform TDD Kata 2 – Moving (20 min timebox)
  - → Pair programming
  - → Ask for help
  - → Central review afterwards

Hands-on-Labs

- **Perform TDD Kata 3 – Winning (20 min timebox)**

→ Pair programming

→ Ask for help

→ Central review afterwards

Testable Code

SOLID principles

- S - Single-responsibility principle
- O - Open-closed principle
- L - Liskov substitution principle
- I - Interface segregation principle
- D - Dependency Inversion Principle

Single responsibility principle

“A class should have one and only one reason to change, meaning that a class should have only one job.”

Open-closed principle

“Objects or entities should be open for extension, but closed for modification.”

Liskov substitution principle

“Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.”

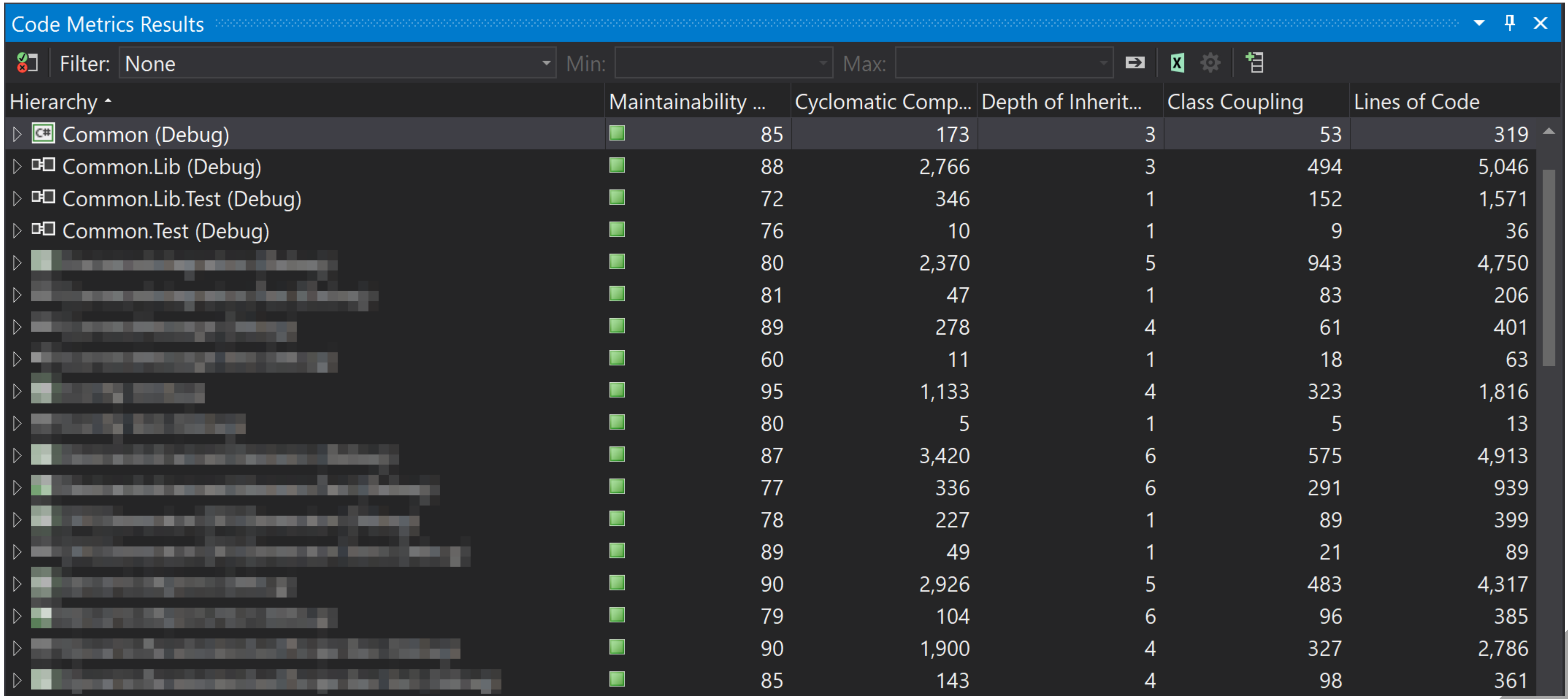
Interface segregation principle

“Many client-specific interfaces are better than one general-purpose interface.”

Dependency inversion principle

“One should "depend upon abstractions, not concretions.”

Analyze > Calculate Code Metrics



Code Metrics Results

Filter: None   Min: Max:

Hierarchy	Maintainability ...		Cyclomatic Comp...	Depth of Inherit...	Class Coupling	Lines of Code
Common (Debug)	🟩	85	173	3	53	319
Common.Lib (Debug)	🟩	88	2,766	3	494	5,046
Common.Lib.Test (Debug)	🟩	72	346	1	152	1,571
Common.Test (Debug)	🟩	76	10	1	9	36
[redacted]	🟩	80	2,370	5	943	4,750
[redacted]	🟩	81	47	1	83	206
[redacted]	🟩	89	278	4	61	401
[redacted]	🟩	60	11	1	18	63
[redacted]	🟩	95	1,133	4	323	1,816
[redacted]	🟩	80	5	1	5	13
[redacted]	🟩	87	3,420	6	575	4,913
[redacted]	🟩	77	336	6	291	939
[redacted]	🟩	78	227	1	89	399
[redacted]	🟩	89	49	1	21	89
[redacted]	🟩	90	2,926	5	483	4,317
[redacted]	🟩	79	104	6	96	385
[redacted]	🟩	90	1,900	4	327	2,786
[redacted]	🟩	85	143	4	98	361

Seams

- Seams are places in the code where behavior can be changed to allow unit testing
  - → Do use dependency injection using interfaces
  - → Do use virtual methods or properties
  - → Don’t use static classes

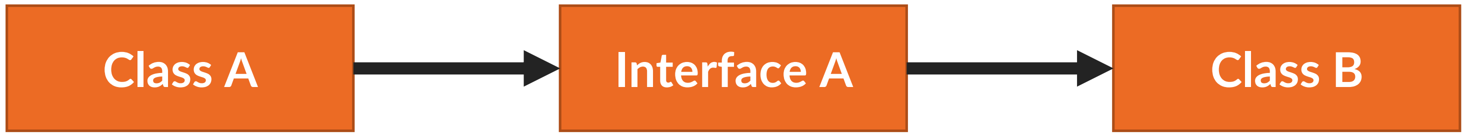
Xpirit

Unit Isolation

- Unit tests do not use actual implementation of dependencies
  - → Databases
  - → Web services
  - → File system
- Replace dependencies with fake objects
- Choose size of your unit:
  - → Unit isolation testing
  - → Unit integration testing

Dependency Inversion

- High-level modules should not depend on low-level modules. Both should depend on abstractions.
- Abstractions should not depend on details. Details should depend on abstractions



Dependency Injection

- Make internal dependencies accessible to outside world
  → Convenient in a lot of situations, e.g. changes, testing
  → Be explicit about dependencies
- Three common ways to inject a dependency
  → Constructor
  → Property
  → Method call

Xpirit

Constructor Injection

```
public class Game

{

   private IHighScoreService highScoreService;
   public Game(IHighScoreService highScoreService)
   {
      this.highScoreService = highScoreService;
   }

   public void Play()
   {
      highScoreService.Start();
      …
   }

}
```

Constructor Injection with default

```
public class Game
{
   private IHighScoreService highScoreService;
   public Game(IHighScoreService highScoreService = null)
   {
      this.highScoreService = highScoreService ?? new HighScoreService();
   }

   public void Play()
   {
      highScoreService.Start();
      …
   }
}
```

Property Injection

```
public class Game

{

   public IHighScoreService HighScoreService { get; set; }

   public void Play()
   {
      HighScoreService.Start();
      …
   }

}
```

Property Injection with default

```
public class Game

{

   private IHighScoreService highScoreService;
   public IHighScoreService HighScoreService
   {
      get => highScoreService ?? new HighScoreService();
      set => highScoreService = value;
   }

   public void Play()
   {
      HighScoreService.Start();
      …
   }

}
```

Mocking dependencies

Mocking Frameworks

- A mocking framework helps to isolate the behavior of the system under test . It can replace dependencies with mocked objects which can simulate the behavior of the dependencies.
- Examples:
  - → Moq
  - → NMock
  - → Nsubstitute
  - → FakeItEasy

DEMO: Mocking framework

Xpirit

State & Interaction based tests

	State-based	Interaction-based
When to use	Care about state of an object.	Care about the behavior of an object.
What is asserted	The system under test itself.	A (mocked) object the system under test is dependent on.
Examples	Assert a property or method result has a certain value.	Verify that a method has been called. Verify the arguments of a method.

Hands-on-Labs

- Perform TDD Kata 5 - Logging (15 min timebox)
  - → Pair programming
  - → Ask for help
  - → Central review afterwards

Xpirit

Testing Exceptions

DEMO

- Testing exceptions

Xpirit

Unit Testing Exceptions

xUnit

```
// Act
Action action = () => new Game(null);
// Assert
Assert.Throws<ArgumentNullException>(action)


// Act & Assert
Assert.Throws<ArgumentNullException>(
       () => new Game(highscoreService));
```

Xpirit

Unit Testing Exceptions

FluentAssertions

```
// Act
Action action = () => new Game(null);
// Assert
action.ShouldThrow<ArgumentNullException>();
```

Hands-on-Labs

- Perform TDD Kata 4 – Boundaries (20 min timebox)
  - → Pair programming
  - → Ask for help
  - → Central review afterwards

Check out

- What did you like best?
- What could be improved?
- Which topics should we cover next lesson?

Xpirit