TESTING

Patrick Drechsler
@drechsler
Redheads Ltd.

2020-01-xx

WHY DO WE TEST?

interaction!!

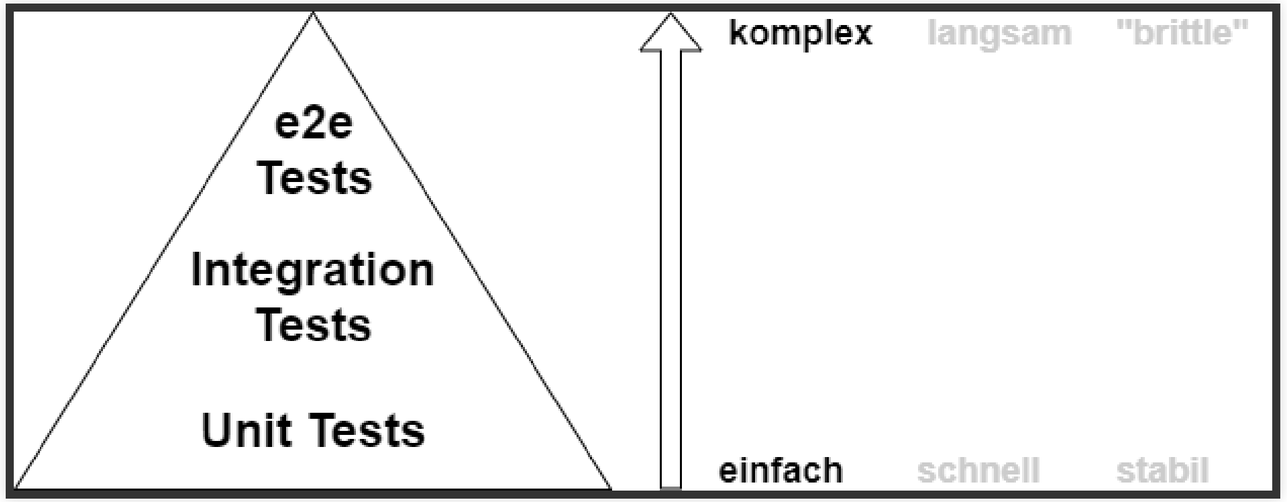
pen & paper -> write sticky notes

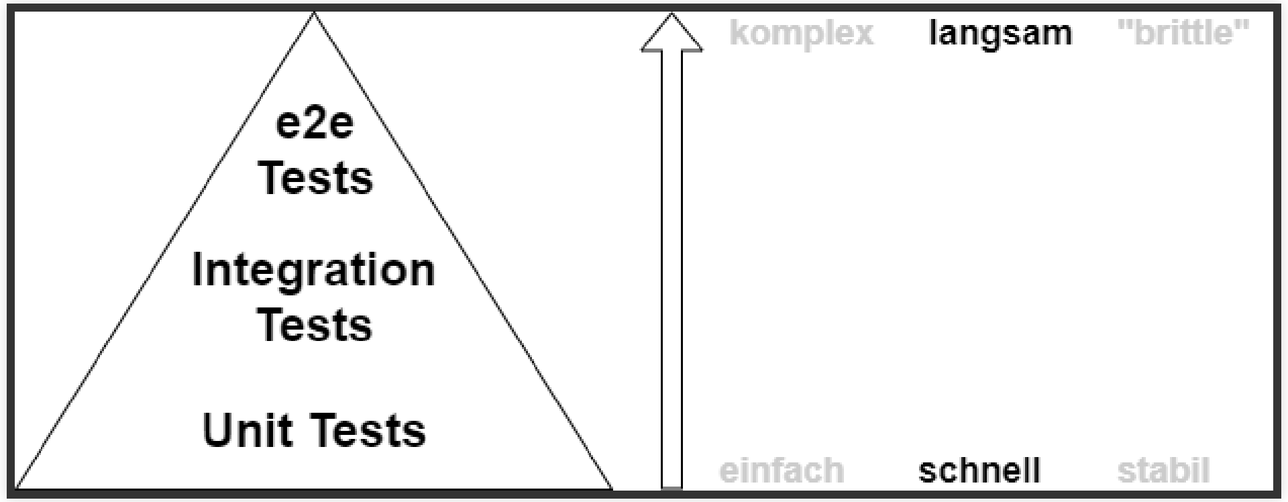
- prevent regression bugs
- improve quality / exposes edge cases
- find bugs early
- documentation
- simplifies debugging
- forces us to think about design (for example integration with other components)

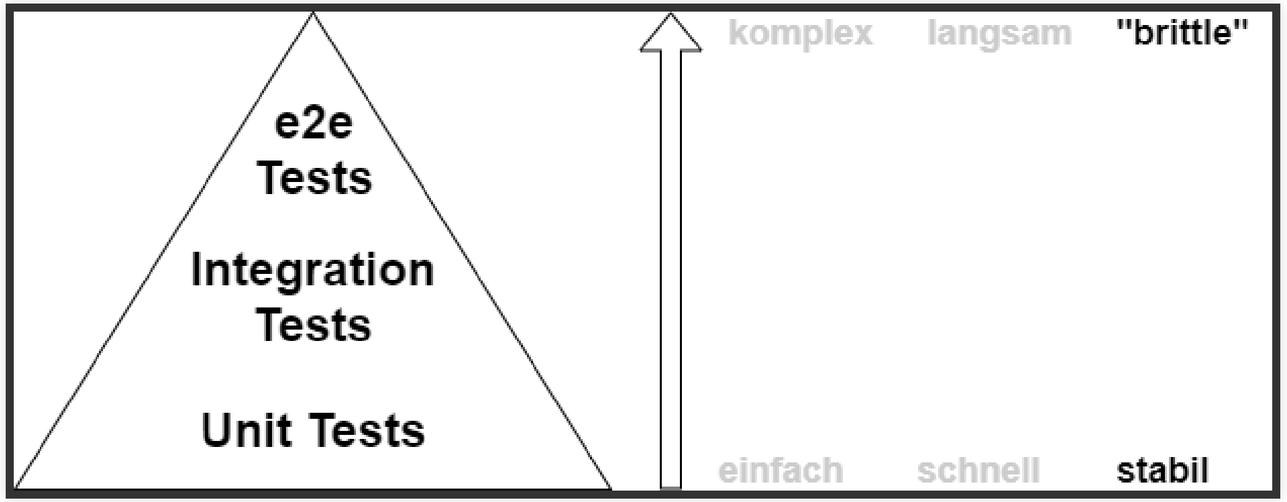
DIFFERENT KINDS OF TESTS

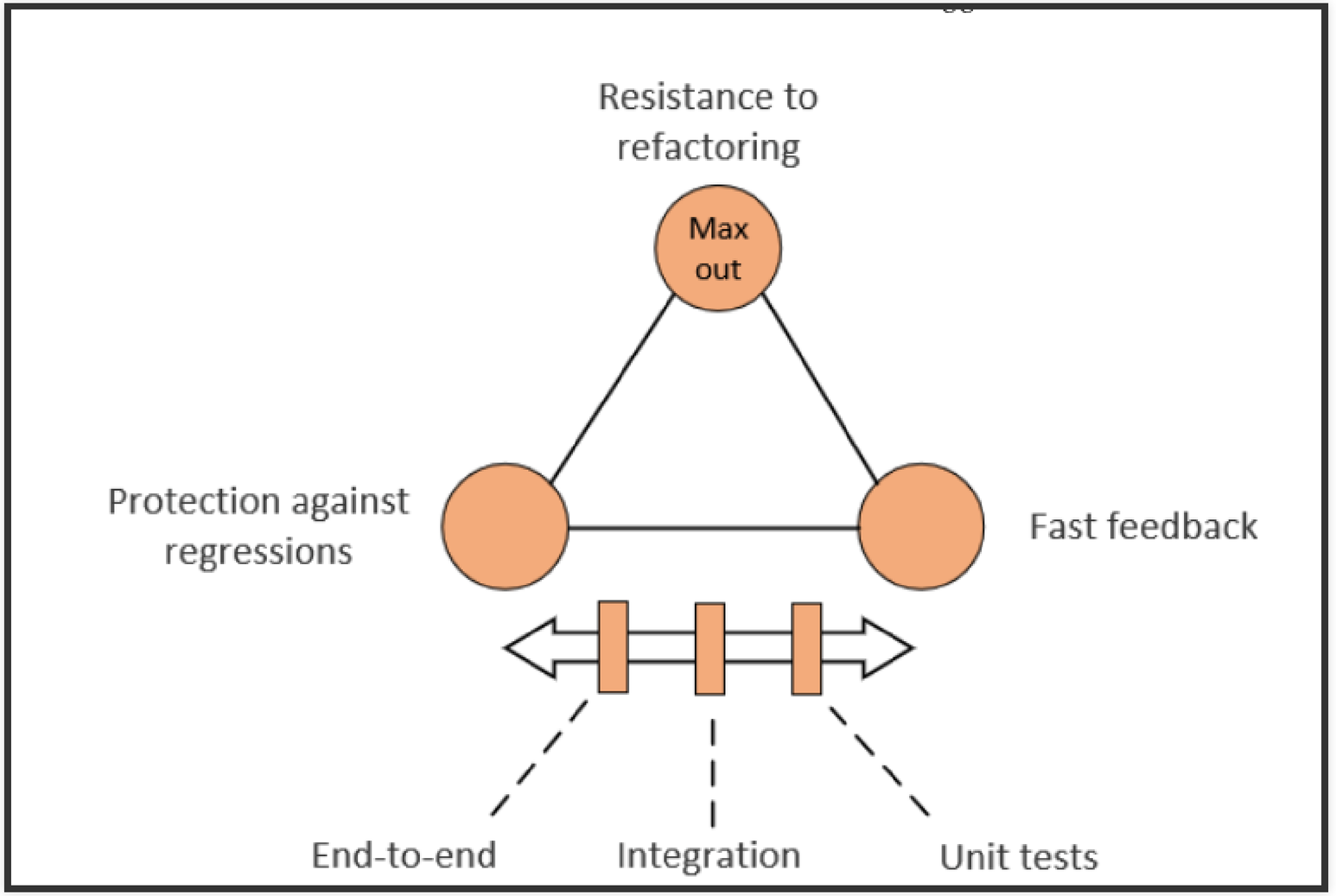
- unit tests
- integration tests
- smoke tests
- end-to-end tests
- system tests
- acceptance tests
- ui tests
- ...

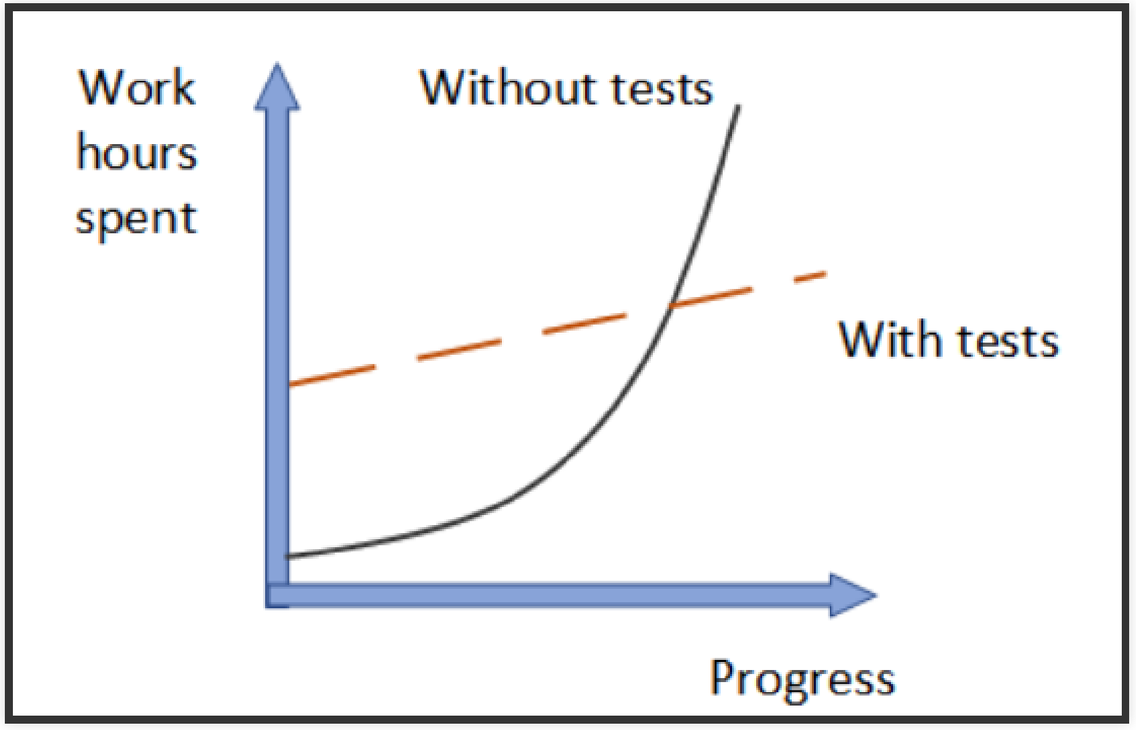
TEST PYRAMIDE

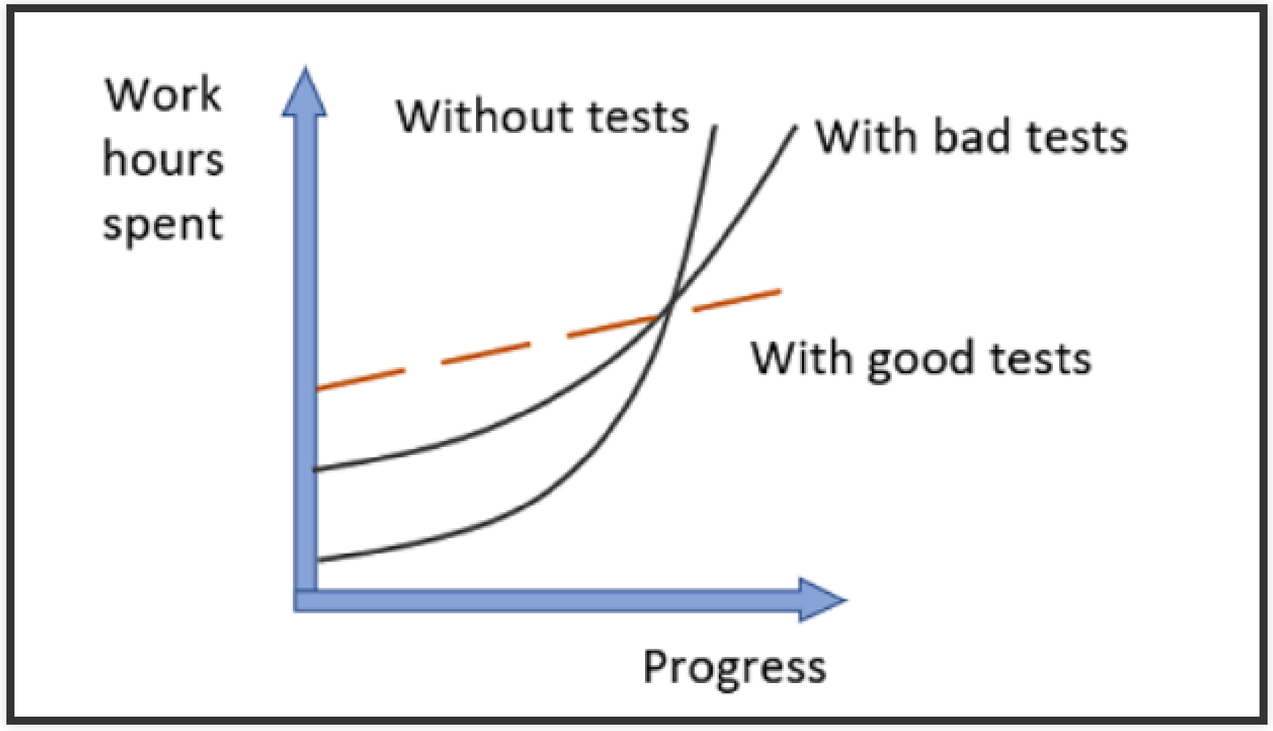

e2e
Tests
Integration
Tests
Unit Tests
komplex
langsam
"brittle"
einfach
schnell
stabil


e2e
Tests
Integration
Tests
Unit Tests
komplex
langsam
"brittle"
einfach
schnell
stabil


e2e
Tests
Integration
Tests
Unit Tests
komplex
langsam
"brittle"
einfach
schnell
stabil


Resistance to refactoring
Max out
Protection against regressions
Fast feedback
End-to-end
Integration
Unit tests


Work
hours
spent
Without tests
With tests
Progress


Work
hours
spent
Without tests
With bad tests
With good tests
Progress

AGENDA

- purpose of testing
- testing frameworks basics
- assertion libraries
- **we code!**
- integration testing & mocking
- **we code!**
- introduction to TDD

HELLO WORLD UNIT TEST

```
// production code
public int Add(int a, int b) ⇒ a + b;

// test code
[Test]
public void Add_1_and_1_returns_2()
    ⇒ Assert.That(Add(1, 1), Is.EqualTo(2));
```

C# TESTING FRAMEWORKS

- MSTest
  - the "IE" of testing frameworks
- NUnit
  - port of JUnit
  - very stable & actively maintained
- xUnit
  - successor of NUnit
  - modern, stable & very modular
  - Microsoft uses xUnit

NUNIT BASICS

- `TestFixture` attribute on test class
- `Test` attribute on method

```
[TestFixture] // ←
public class CustomerRepositoryTests
{
    [Test]    // ←
    public void GetById_bla(){}
}
```

NUNIT ASSERTIONS

https://github.com/nunit/docs/wiki/Assertions

- Classic model (deprecated!): https://github.com/nunit/docs/wiki/Classic-Model
- Constraint model: https://github.com/nunit/docs/wiki/Constraint-Model

NUNIT CLASSIC ASSERTIONS

```
var result = "foo";

// (1)
Assert.IsTrue(result == "foox");

// (2)
Assert.AreEqual(result, "foox");
```

- Both assertions fail.

Error message from (1) is useless:

```
Expected: True
But was:  False
```

Error message from (2):

```
Expected string length 3 but was 4. Strings differ at index 3.
Expected: "foo"
But was:  "foox"
--------------^
```

PROBLEM WITH ALL ASSERTION EQUAL TESTS

```
Assert.AreEqual(result, "foox");
```

I can never remember the order of parameters, because each framework is different.

- some use `(actual, expected)`
- others use `(expected, actual)`

This makes reading the error message difficult.

NUNIT CONSTRAINT MODEL

- always uses the `Assert.That( ... )` syntax

Example:

```
var result = "foo";
Assert.That(result, Is.EqualTo("foox"));
```

- readability improvement
- sane error message

WHY ARE ERROR MESSAGE IMPORTANT?

- local machine: Quick feedback
  - continous testing
  - just read the error message and know what's wrong
- CI server
  - Fail email: why?
- Test naming & error message should tell a story

TEST NAMING CONVENTIONS

- always name test so that error message is clear
- readability tip: use snake case for tests

```
Adding_1_and_1_returns_2
Creating_customer_with_missing_name_throws
```

- use inner classes for methods needing multiple tests

```
CustomerProvider with method GetCustomerById
```

Example

```
[TestFixture]
public class CustomerProviderTests
{
    private class GetById
    {
        [Test]
        public void With_valid_id() {}

        [Test]
        public void With_invalid_id_returns_null() {}
    }
```

Error message:

```
CustomerProviderTests
    → GetById
        → With_valid_id
        → With_invalid_id_returns_null
```

ALTERNATIVE ASSERTION LIBRARIES

- FluentAssertions: https://fluentassertions.com/
- NFluent: http://www.n-fluent.net/

Goal: improve readability and error messages

```
// FluentAssertions
"foo".Should().Be("foox");
```

Bonus: **works with all test frameworks**

FLUENTASSERTIONS

https://fluentassertions.com/

ERROR MESSAGE: REASON

Most methods have an optional "reason" string

```
// wrong implementation
IEnumerable<int> GetEvenNumbers(IEnumerable<int> input)
    ⇒ input.Where(x ⇒ x % 2 ≠ 0).ToList();

[Test]
public void Parse_even_numbers()
{
    var input = new List<int>{0, 1, 2, 3, 4};
    var result = GetEvenNumbers(input);
    result.Should()
        .BeEquivalentTo(
            new List<int>{0, 2, 4},
            "odd numbers are wrong");
}
```

```
Expected result to be a collection with 3 item(s)
because odd numbers are wrong, but {1, 3}
contains 1 item(s) less than {0, 2, 4}.
```

COLLECTIONS

```
// FluentAssertionDemos
var list = new List<string>
{
    "foo",
    "bar",
    "baz"
};

list.Should()
    .HaveCount(3)
    .And.Contain("foo")
    .And.Contain("bar")
    .And.Contain("baz")
    .And.NotContain("42");
```

EXCEPTIONS

```
// FluentAssertionDemos
private static void Throws()
    => throw new Exception("Ups");

[Test]
public void Exceptions_tests()
{
    Action action = () => Throws();
    action.Should()
        .Throw<Exception>()
        .WithMessage("Ups");
}
```

EQUIVALENTTO

Shameless self plug:

http://draptik.github.io/blog/2016/05/09/testing-objects-have-same-properties/

TEST ANATOMY

- Arrange, Act, Assert (AAA)
- Given, When, Then

ANTIPATTERNS

- Avoid multiple AAA sections
  - keep test small
  - try to only test one thing
- Avoid if statements
- Avoid "Act" statement with more than 1 line
- Avoid high coupling in setup methods

```
// Arrange
var num1 = 1;
var num2 = 1;
var sut = new Calculator();
var expectedResult = 2;

// Act
var result = sut.AddNumbers(num1, num2);

// Assert
result.Should().Be(expectedResult);
```

- `Arrange` can be very long!
  - redesign?
- `Act` should only be 1 line!
- `Assert` can be very long!
  - write dedicated assertion?
  - use `AssertionScope`

- Test code should be treated with the same care as production code

ARRANGE: PARAMETERIZED TESTS (1/2)

```
// FluentAssertionDemos
[TestCase(-1, Bar.Undefined)]
[TestCase(0, Bar.All)]
[TestCase(1, Bar.Beer)]
[TestCase(2, Bar.Whiskey)]
[TestCase(3, Bar.Water)]
[TestCase(4, Bar.Undefined)]
[TestCase(99, Bar.Undefined)]
public void MapIntToBar(int input, Bar expected)
{
    MapIntToBar(input).Should().Be(expected);
}
```

ARRANGE: PARAMETERIZED TESTS (2/2)

- https://github.com/nunit/docs/wiki/TestCaseSource-Attribute

```
// FluentAssertionDemos
[TestCaseSource(typeof(CustomerTestData))]
public void CustomerIsValid(Customer customer, bool expected)
{
    customer.IsValid().Should().Be(expected);
}

private class CustomerTestData : IEnumerable
{
    public IEnumerator GetEnumerator()
    {
        yield return new object[] {
            new Customer {FirstName = null, LastName = null}, false};
        yield return new object[] {
            new Customer {FirstName = "a", LastName = "b"}, true};
    }
}
```

RUN CODE BEFORE / AFTER EACH TEST

- Useful when code repeats itself

RUN CODE BEFORE / AFTER EACH FIXTURE

- Useful for integration tests
  - example: DB setup/teardown

TEST SUITES AND OTHER ANNOTATIONS

- slow running tests
  - nightly build

ENCAPSULATION

- avoid testing private methods
  - maybe refactor?
- C# fallback:
  - make method `internal`
  - `InternalsVisibleTo` in assembly

AUTOMATION

- Automation is a critical part of testing!
- within IDE: continous test runner
  - NCrunch
  - R#
- CI (i.e. Jenkins)

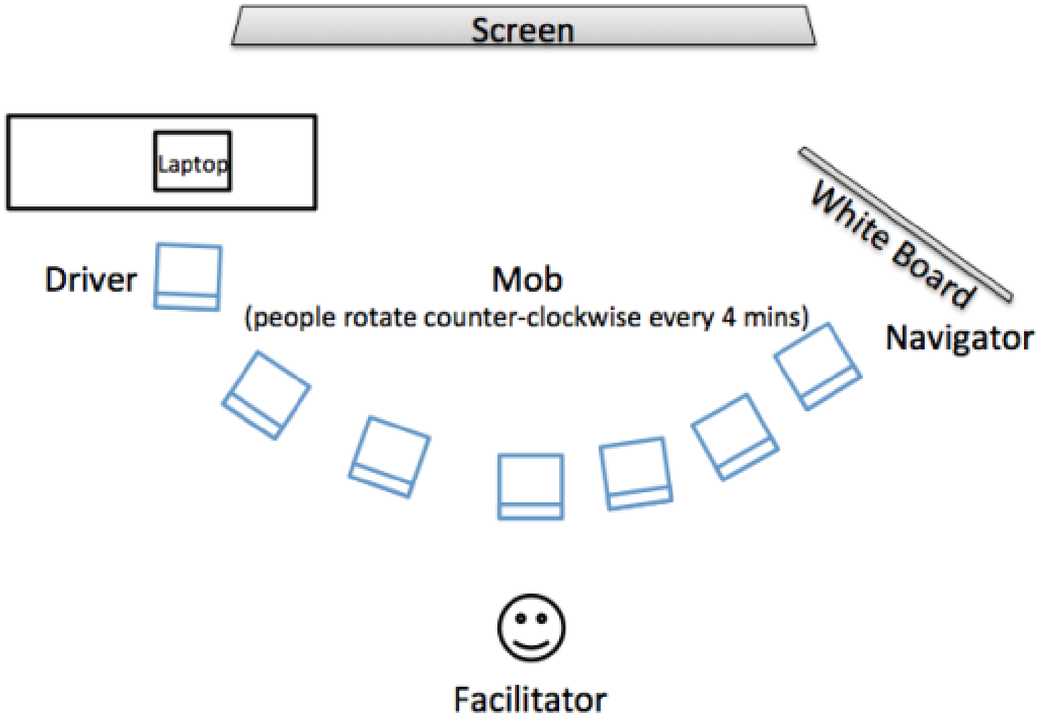
let's code

(One more thing... Mobbing)

MOB PROGRAMMING

- wir lernen gemeinsam
- Pair Programming in der Gruppe

Mob Programming Setup



- Driver: Sitzt an der Tastatur (darf nicht denken)
- Navigator: Sagt dem Driver, was zu tun ist
- Mob: Unterstützt den Navigator
- Regelmäßiger Wechsel (3-5min)

"ASSISTED" MOB PROGRAMMING

- Facilitator unterstützt den Navigator

let's code

Gilded Rose Kata

WHAT SHOULD WE TEST?

WHAT SHOULD WE TEST?

- **focus on user stories**, not on technical details
- **often evokes better design!**

Let's code!

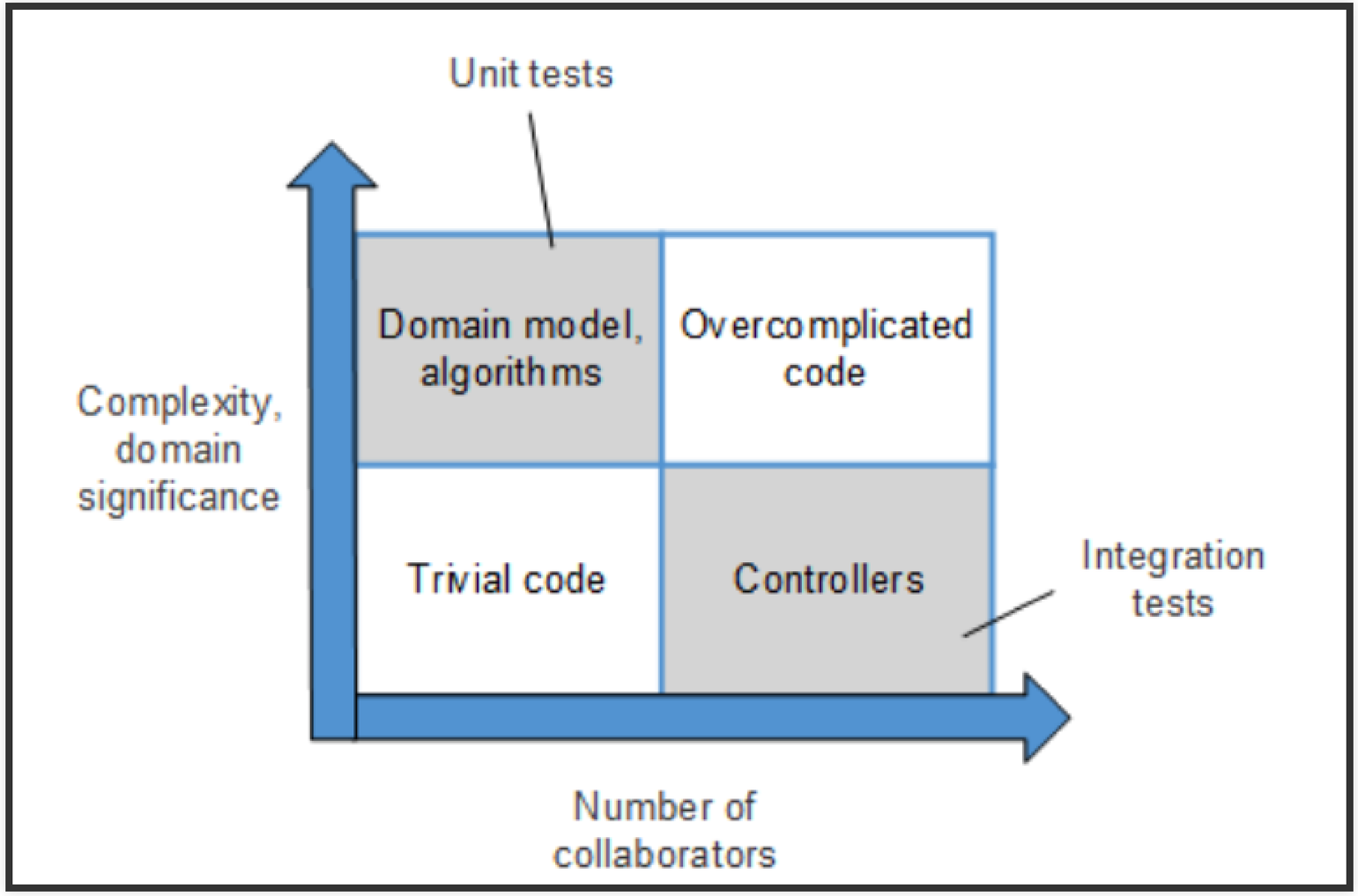
INTEGRATION TESTING

- testing how different components interact
- composing interactions

INTEGRATION TESTING

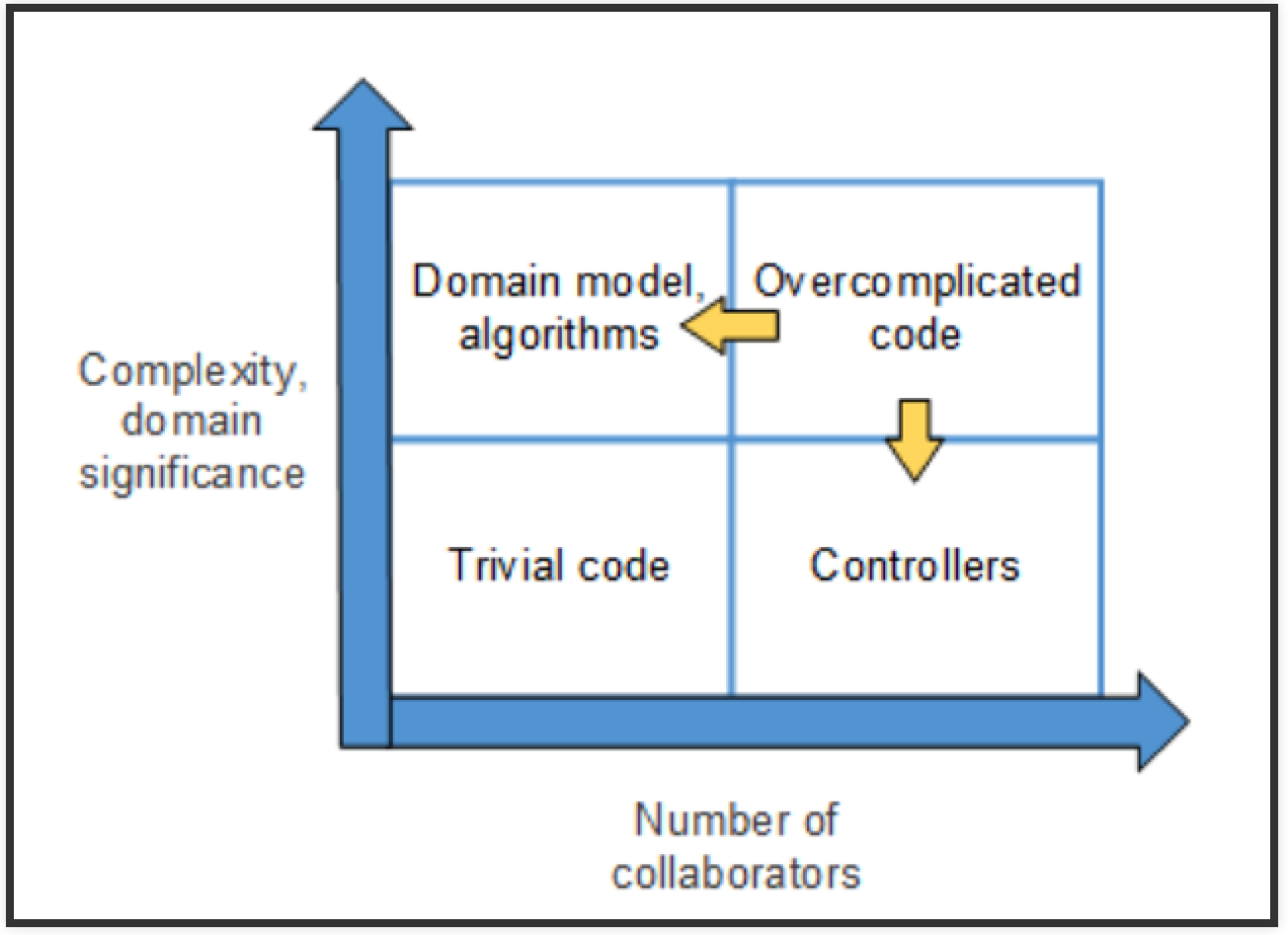
- Always a good indicator for
  - decoupling
  - correct layer of abstraction

TEST MATRIX


Unit tests
Domain model, algorithms
Overcomplicated code
Complexity, domain significance
Trivial code
Controllers
Integration tests
Number of collaborators

TEST MATRIX - CODE SMELL

- "overcomplicated code" quadrant
  - wrong layer of abstraction
  - missing encapsulation
- solution: move logic to domain object


Domain model, algorithms
Overcomplicated code
Trivial code
Controllers
Complexity, domain significance
Number of collaborators

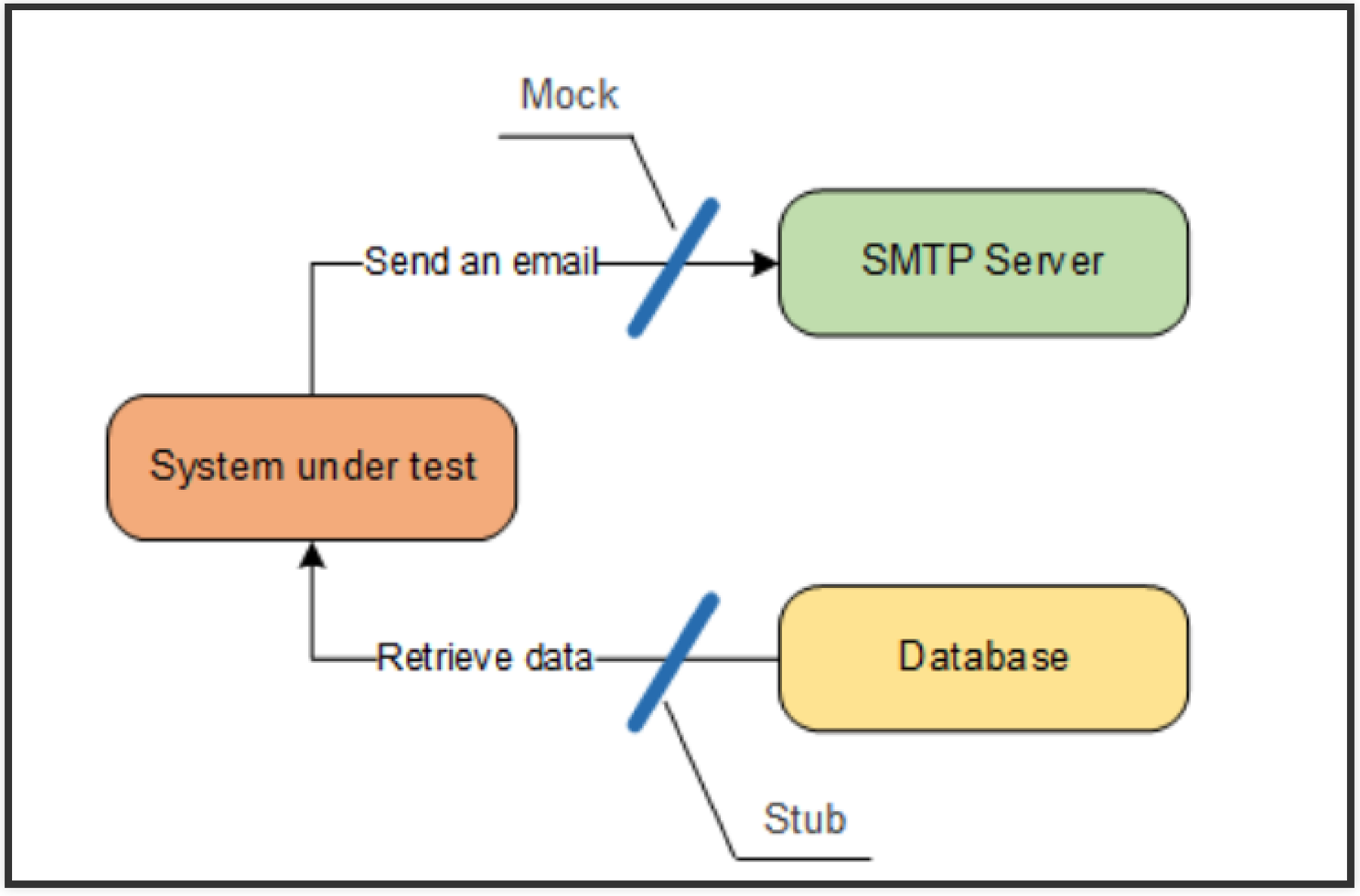
WHAT SHOULD WE TEST?

- **focus on user stories**, not on technical details
- **often evokes better design!**
- when testing technical details:
  - focus on happy path
  - bug report: write a test demonstrating the bug; then fix the bug
  - keep techn. tests maintainable!

TEST DOUBLES

- Mock (spy)
  - emulate **outcoming** interactions
- Stub (dummy, fake)
  - emulate **incoming** interactions

Many modern frameworks don't make this distinction.


Mock
Send an email
SMTP Server
System under test
Retrieve data
Database
Stub

MANUALLY CREATING MOCKS

- only useful for simple objects
- time consuming

MOCKING FRAMEWORKS

- often require interface or virtual methods (C#)
- focus on methods required in test
- setup pre conditions
- verify post conditions

C# MOCKING FRAMEWORKS

- Moq
  - can also mock classes without interfaces
- NSubstitute
  - nice API
- FakeItEasy
  - nice API

NSUBSTITUTE

```
// Arrange
var mailer = Substitute.For<IMailer>(); // ←

var validCustomer = new ValidCustomer();

// We inject an interface to the system-under-test!
// Mocking frameworks can control the behaviour of the injected object!
var sut = new RegistrationService(mailer);

// Act
sut.Register(validCustomer);

// Assert
mailer.Received().Send(); // ←
```

examples in code

GOLDEN MASTER

- use actual program behaviour as reference
- useful when working with legacy code you don't understand
- compares final ouptut with previously "recorded" output (record analogy)

now you know the **tooling**!

Let's use **testing** to **DRIVE** our **development**!

RECAP: TRADITIONAL APPROACH

- Design
- Implementation
- Test

DRAWBACKS WHEN WRITING TESTS AT END

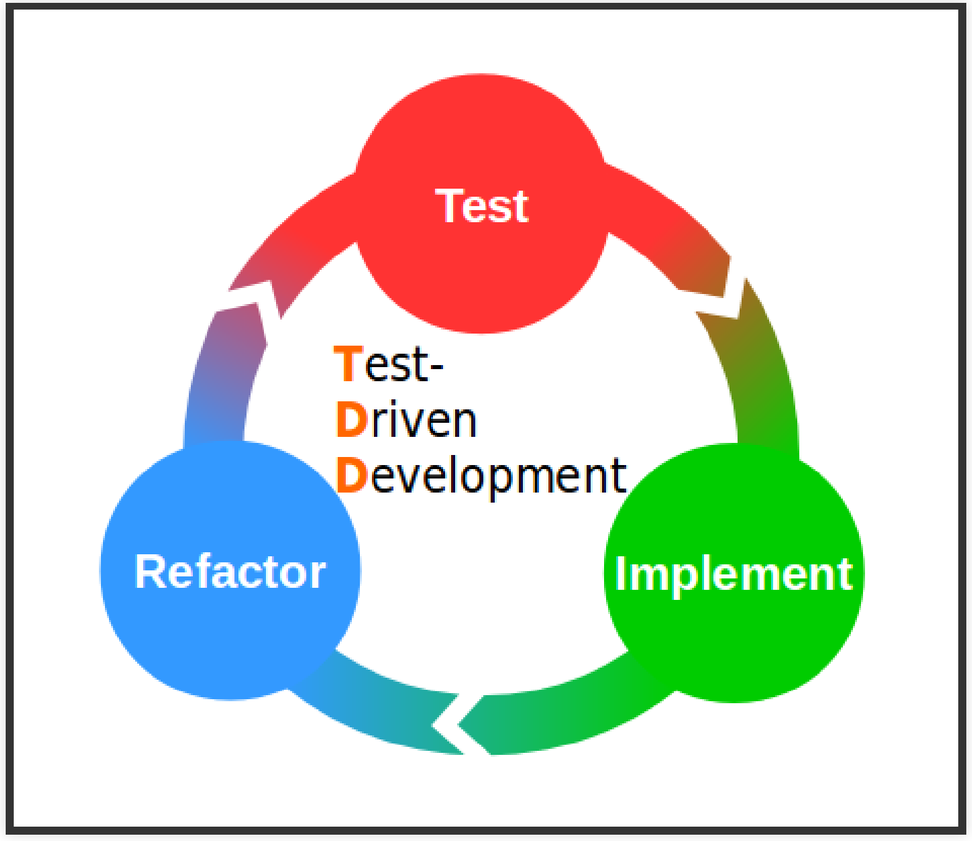
- low test coverage
- late testing: rethink problem
- knowledge of implementation: changes how tests are written
- not automated

TEST-DRIVEN DEVELOPMENT (TDD)

- Design
- Test
- Implementation

TDD WORKFLOW

- Red
- Green
- Refactor


Test
Test-
Driven
Development
Refactor
Implement

- no code without test
- untested code should not go live
- baby steps: only write code to make test pass

TDD PATTERNS

- isolated tests
- Test List (on paper, not in code -> small steps)
- Test first -> less "stress"
- Assert first
- use Test data (vs realistic data)

RED BAR PATTERNS

Patterns decribing

- when to write tests
- where to write tests
- when to stop writing test

RED BAR PATTERNS

- Starter test
- Explaining test
- Regression test
- One step test

STARTER TEST

- Output equals input
- input as small as possible

EXPLAINING TEST

- Test should explain what is happening

REGRESSION TEST

- small test reproducing error
- why wasn't the test there before?
- Redesign?

ONE STEP TEST

- Categorize tasks by difficulty
  - Obvious
  - no idea
  - WAT?
  - **this I can do**

GREEN BAR PATTERNS

Patterns describing how to go from red to green

- obvious implementation
- triangulation
- "fake it til you make it"

OBVIOUS IMPLEMENTATION

- simple enough -> just implement it (driving in 2nd gear)
- BUT: if you encounter surprising red bars -> take smaller steps ("be prepared to downshift")

TRIANGULATION

- Abstract only if you have 2 or more examples
- Use when you are really unsure about correct abstraction.
- Prefer "fake it til you make it" or "Obvious Implementation"

FAKE IT TIL YOU MAKE IT

TRANSFORMATION PRIORITY PREMISE

- ...gradually replace constants with variables...
- taken one step further
- Example using Transformation Priority Premise

ADVANTAGES OF TDD

- high test coverage
- increases modularity
- improves maintainability
- implicit documentation

let's code

Fizz-Buzz Kata

SCHOOLS OF TDD

- **Classical**
  - aka Detroit, Chicago, "inside-out", "fake it til you make it"
  - K. Beck: Test-Driven Development By Example
- **London**
  - aka moquist, "outside-in"
  - Freeman & Pryce: Growing Object Oriented Software Guided by Tests
- Munich: David Voelkl
- Hamburg: Ralph Westpfahl

RESSOURCEN

- Kent Beck: Test-Driven Development By Example
- Roy Osherove: The Art of Unit Testing
- Steve Freeman and Nat Pryce: Growing Object Oriented Software Guided by Tests
- Vladimir Khorikov: Unit testing. Principles, Practices and Patterns