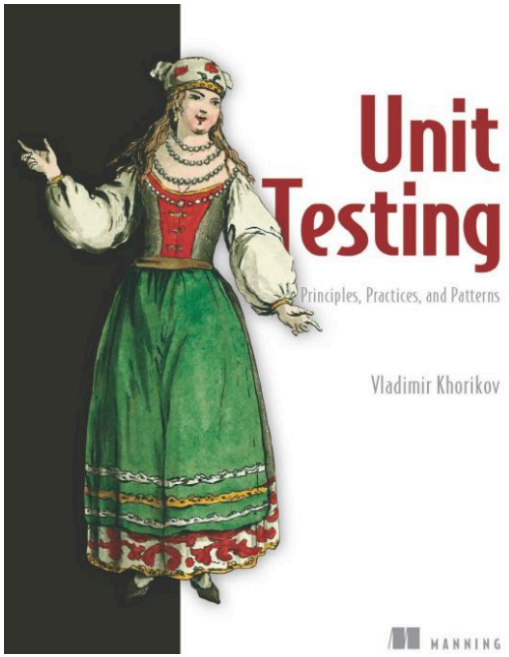
Unit Testing

An attempt to finding a common terminology and testing strategy for our team

Crelder, 29.6.2021



All the content (ideas, images, text) from the following presentation is taken from this book.

Other sources are explicitly mentioned.

Goals and Motivation

- Clarity about terms regarding testing
  mock, stub, fake, dummy, spy, dependency, unit and integration test, etc.

- Ideally: Team agreement on a common test strategy or school of testing

- Not covered in this presentation:
  testing the database,
  styles of unit testing,
  different coverage metrics

Questions

1. What is a good test?
2. What is an ideal software test?
3. What is the difference between the testing in the classical school and the Mockist (London) school?
4. What is the anatomy of a unit test?
5. How can you refactor toward good unit tests?
6. What is the relationship between mocks and test fragility (brittle tests, many false positives)?
7. What kind of dependencies are there?
8. What are out-of-process dependencies and how do they relate to shared dependencies?
9. What is the difference between managed and unmanaged out-of-process dependencies?
10. What kind of test doubles exist? What are their relations to the CQS-principle?
11. Which test double terms fall under the category 'mock' and 'stub'?
12. What is the value of a good testing suite over the long run of a project?
13. What is the relationship of false positives and false negatives in the test suite over the duration of a project?

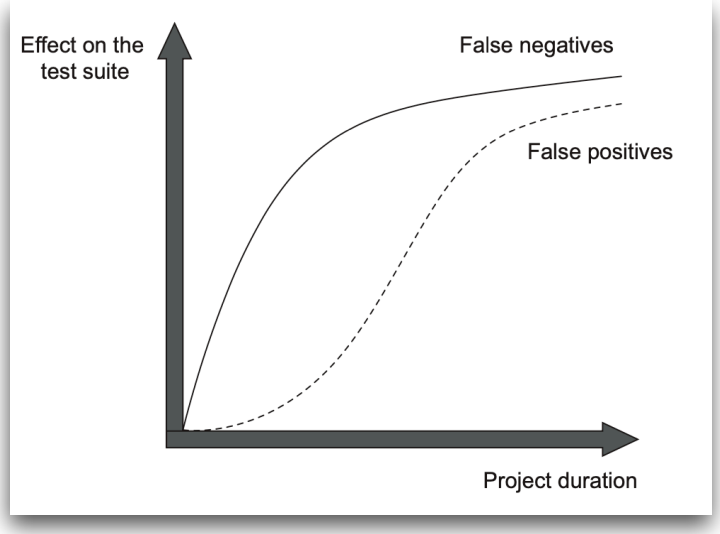
I. Theoretical Part

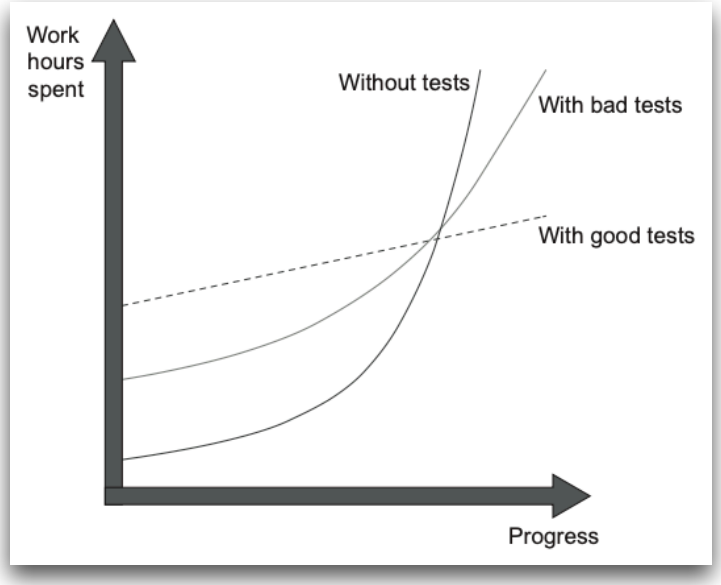
What is a good test?

	Test fails	Test passes
Code wrong	**True Positive** ↑	False Negative
Code works	False Positive ↓	**True Negative**

A good test is a test with **high test accuracy**

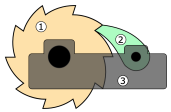
$$testAccuracy = \frac{Signal\ (number\ of\ bugs\ found)}{Noise\ (number\ of\ false\ alarms\ raised)}$$





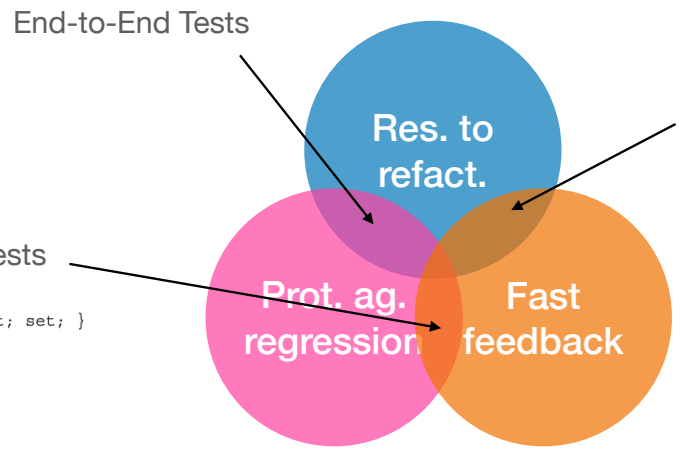
What is an ideal test in software development?

- **Protection against regression**
- **Resistance to refactoring**
- **Fast feedback**
- **Maintainability**



	Test fails	Test passes
Code wrong	**True Positive** ↑	False Negative ↓
Code works	False Positive ↓	**True Negative** ↑

```
public class UserRepository
{
    public User GetById(int id)
    {
        /* ... */
    }

    public string LastExecutedSqlStatement { get; set; }
}

[Fact]
public void GetById_executes_correct_SQL_code()
{
    var sut = new UserRepository();

    User user = sut.GetById(5);

    Assert.Equal(
        "SELECT * FROM dbo.[User] WHERE UserID = 5",
        sut.LastExecutedSqlStatement);
}
```



SELECT * FROM dbo.[User] WHERE UserID = 5
SELECT * FROM dbo.User WHERE UserID = 5
SELECT UserID, Name, Email FROM dbo.[User] WHERE UserID = 5
SELECT * FROM dbo.[User] WHERE UserID = @UserID

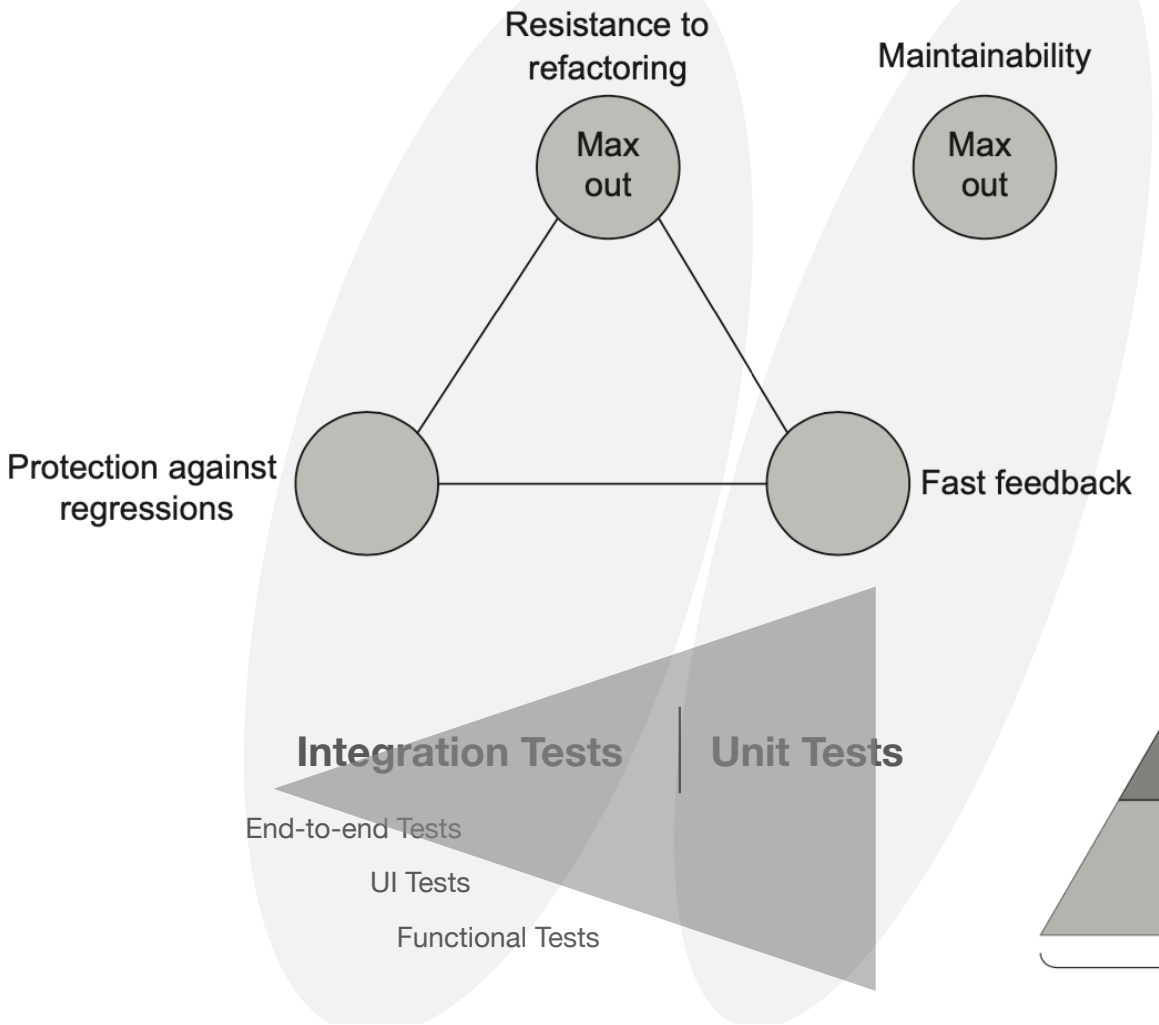
Trivial tests

```
public class User
{
    public string Name { get; set; }
}
```

One-liners like this are unlikely to contain bugs.

```
[Fact]
public void Test()
{
    var sut = new User();

    sut.Name = "John Smith";

    Assert.Equal("John Smith", sut.Name);
}
```

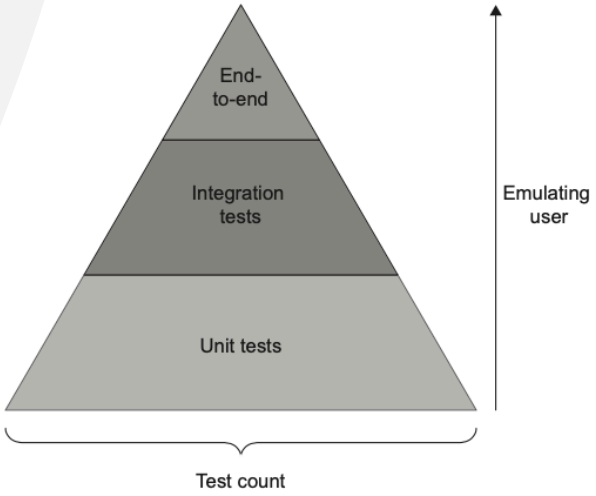
Unit Tests and Integration Tests



A **unit test** is a test that

* Verifies a single **unit of behaviour**
* Does it quickly
* And does it in isolation from **other tests**

An **integration test** is a test that doesn't meet one of these criteria.



Unit tests in different schools of testing

```
{ // CLASSICAL SCHOOL
  // This is a unit test from the view of the classical school
  // But an integration test for the the Mockist school
    public class CustomerTests
    {
        public void Purchase_succeeds_when_enough_inventory()
        {
            // Arrange
            var store = new Store();
            store.AddInventory(Product.Shampoo, 10);
            var customer = new Customer();

            // Act
            bool success = customer.Purchase(store, Product.Shampoo, 5);

            // Assert
            Assert.True(success);
            Assert.Equal(5, store.GetInventory(Product.Shampoo));
        }

    }
}
```

```
{   // MOCKIST (or LONDON) SCHOOL
    public class CustomerTests
    {

        public void Purchase_succeeds_when_enough_inventory()
        {
            // Arrange
            var storeMock = new Mock<IStore>();
            storeMock
                .Setup(x => x.HasEnoughInventory(Product.Shampoo, 5))
                .Returns(true);
            var customer = new Customer();

            // Act
            bool success = customer.Purchase(storeMock.Object, Product.Shampoo, 5);

            // Assert
            Assert.True(success);
            storeMock.Verify(x => x.RemoveInventory(Product.Shampoo, 5), Times.Once);
        }

    }
}
```

```
public enum Product
{
    Shampoo,
    Book
}
```

Contrasting two schools of testing

	Isolation of	A *unit* is	Uses test doubles for
Mockist school	Units	A class	All but immutable dependencies
Classical school	Unit tests	A unit of behaviour (a class or a set of classes)	Shared dependencies

Further reading:

Steve Freeman, Nat Pryce: Growing Object- Oriented Software, Guided by Tests (Addison-Wesley Professional, 2009)

Kent Beck: Test-Driven Development: By Example (Addison-Wesley Professional, 2002)

Advantages of the Mockist School:

* Better granularity with a clear rule: one class, one test
* If a test fails, you know for sure which functionality has failed

* Easier unit testing a larger graph of interconnected classes

=> Test shouldn't test units of code, but rather **units of behaviour**
=> Running your tests regularly, you know what caused the bug (look at the **last change**)
=> Focus on **not having such a graph of classes** in the first place! Good thing, that tests point out this problem

Dependencies

'Shared' = Shared between tests

= Not shared

	Shared	Private
Mutable	DB	Other classes in domain logic
Immutable	-	Value, value object

Classical school

Mockist School

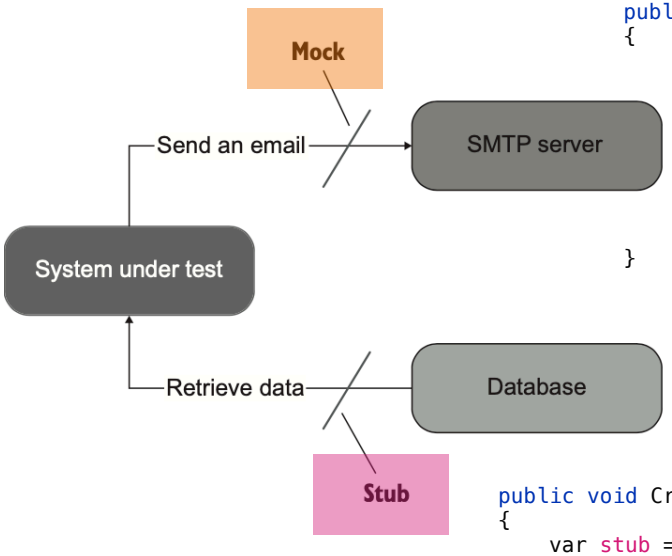
	shared	private
In-process	Static mutable field, singleton	**Other classes in domain logic**
out-of-process	**DB**	Read-only-API

```
public enum Product
{
    Shampoo,
    Book
}
```

```
{ // CLASSICAL SCHOOL

    public class CustomerTests
    {
        public void Purchase_succeeds_when_enough_inventory()
        {
            // Arrange
            var store = new Store();
            store.AddInventory(Product.Shampoo, 10);
            var customer = new Customer();

            // Act
            bool success = customer.Purchase(store,
                                             Product.Shampoo, 5);

            // Assert
            Assert.True(success);
            Assert.Equal(5, store.GetInventory(Product.Shampoo));
        }
}
```

```
// MOCKIST SCHOOL

...

// Arrange
var storeMock = new Mock<IStore>();
storeMock
    .Setup(x => x.HasEnoughInventory(Product.Shampoo, 5))
    .Returns(true);
var customer = new Customer();

...
```

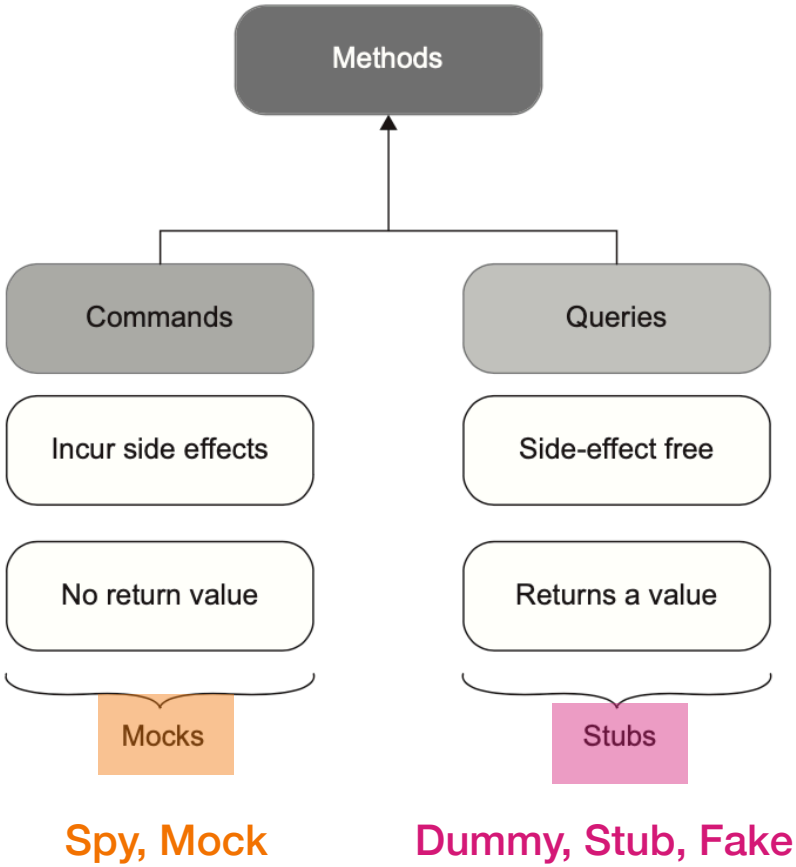
Test Doubles, Mocks, Stubs




```
public void Sending_a_greetings_email()
{
    var emailGatewayMock = new Mock<IEmailGateway>();
    var sut = new Controller(emailGatewayMock.Object);

    sut.GreetUser("user@email.com");

    emailGatewayMock.Verify(
        x => x.SendGreetingsEmail("user@email.com"),
        Times.Once);
}
```

```
public void Creating_a_report()
{
    var stub = new Mock<IDatabase>();
    stub.Setup(x => x.GetNumberOfUsers()).Returns(10);
    var sut = new Controller(stub.Object);

    Report report = sut.CreateReport();

    Assert.Equal(10, report.NumberOfUsers);
}
```

Command Querry Segregation Principle



Spy, Mock

Dummy, Stub, Fake

Mocks and test fragility (brittle tests)

	Observable behaviour	Implementation detail
Public	Good	Bad
Private	-	Good

For a piece of code to be part of the system's **observable behavior** it has to do one of the following things:

* Expose an **operation** that **helps the client achieve one of its goals**. An operation is a method that performs a calculation or incurs a side effect or both.
* Expose a **state** that **helps the client achieve one of its goals.** State is the current condition of the system.

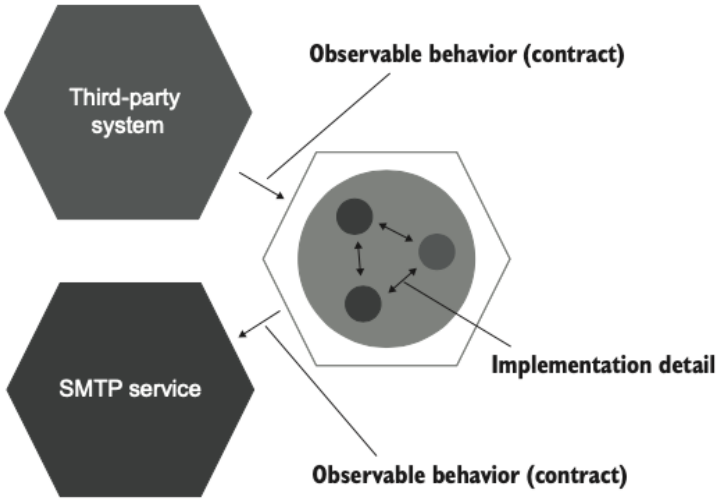
Everything that is not observable behaviour is **implementation detail.**

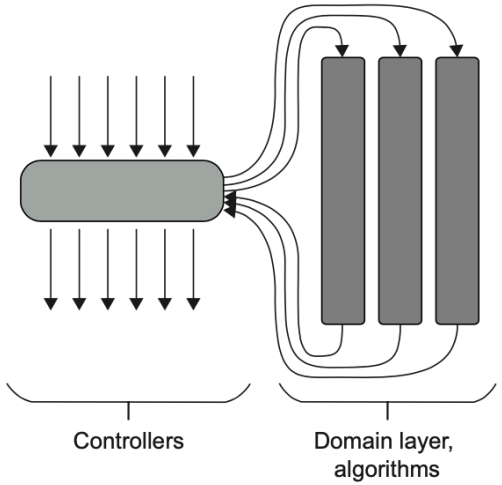
```
{    // BAD, BECAUSE OF LEAKING IMPLEMENTATION DETAILS (NormalizeName)
    public class User
    {
        public string Name { get; set; }

        public string NormalizeName(string name)
        {
            string result = (name ?? "").Trim();

            if (result.Length > 50)
                return result.Substring(0, 50);

            return result;
        }
    }

    public class UserController
    {
        public void RenameUser(int userId, string newName)
        {
            User user = GetUserFromDatabase(userId);

            string normalizedName = user.NormalizeName(newName);
            user.Name = normalizedName;

            SaveUserToDatabase(user);
        }

        private void SaveUserToDatabase(User user)
        { ...
        }

        private User GetUserFromDatabase(int userId)
        {
            return new User();
        }
    }
}
```

```
{    // GOOD
    public class User
    {
        private string _name;
        public string Name
        {
            get => _name;
            set => _name = NormalizeName(value);
        }

        private string NormalizeName(string name)
        {
            string result = (name ?? "").Trim();

            if (result.Length > 50)
                return result.Substring(0, 50);

            return result;
        }
    }

    public class UserController
    {
        public void RenameUser(int userId, string newName)
        {
            User user = GetUserFromDatabase(userId);
            user.Name = newName;
            SaveUserToDatabase(user);
        }

        private void SaveUserToDatabase(User user)
        { ...
        }

        private User GetUserFromDatabase(int userId)
        {
            return new User();
        }
    }
}
```

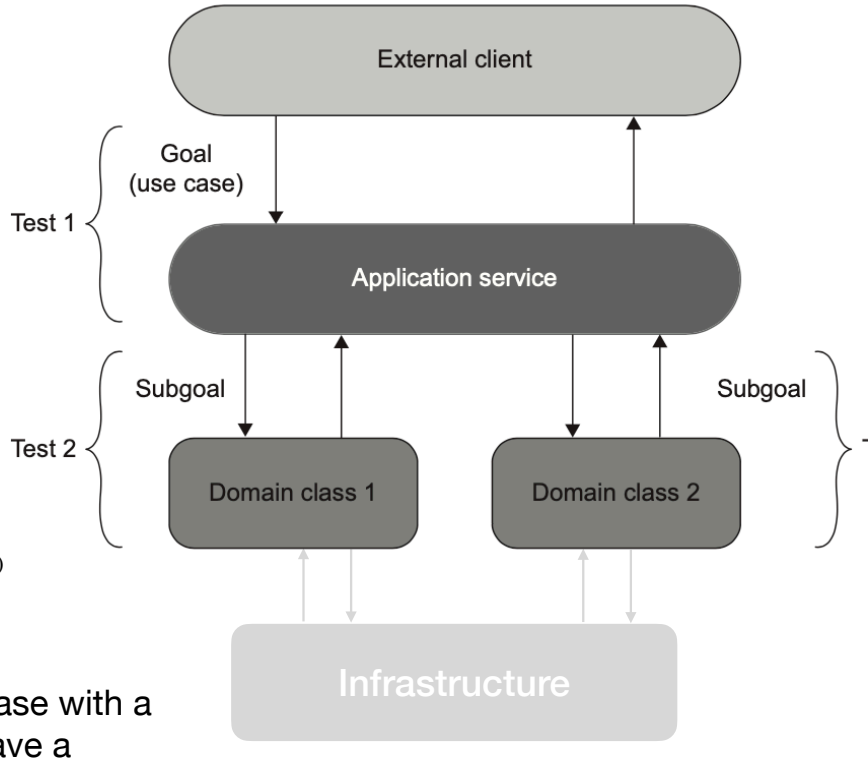
Making the API well-designed automatically improves unit tests (better resistance to refactoring)!

Mocks and test fragility (brittle tests)





See: Domain-Driven Design: Tackling Complexity in the Heart of Software by Eric Evans (Addison-Wesley, 2003)



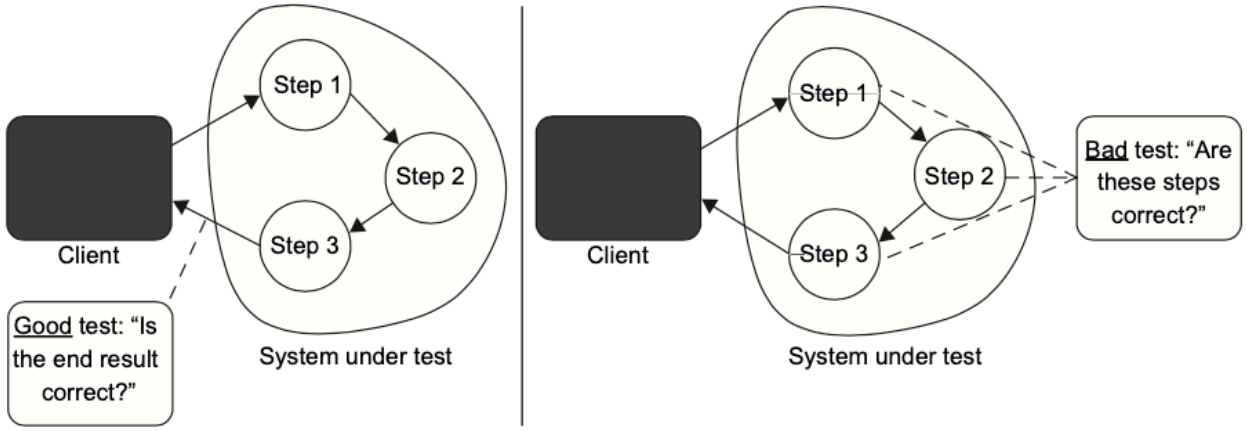
Tests that verify a code base with a well-designed API also have a connection to a business requirements because those tests tie to the observable behaviour only.

This guideline does not apply so much to the infrastructure code.

Anatomy of a unit test

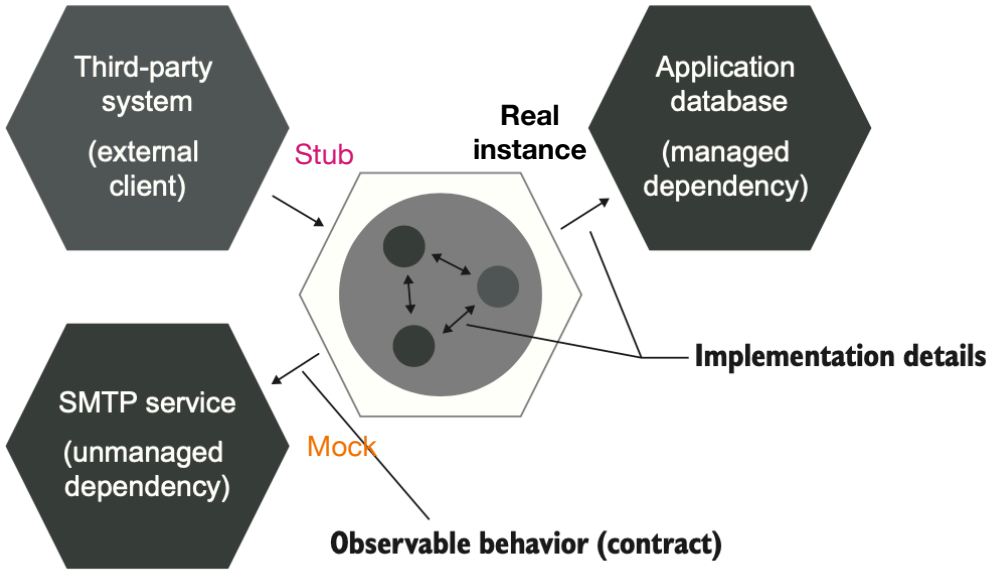
- AAA pattern
  - Arrange section
  - Act section
  - Assert section
- Given-When-Then Pattern

```
{
    public class CustomerTests
    {
        public void Purchase_succeeds_when_enough_inventory()
        {
            // Arrange
            var store = new Store();
            store.AddInventory(Product.Shampoo, 10);
            var customer = new Customer();

            // Act
            bool success = customer.Purchase(store, Product.Shampoo, 5);

            // Assert
            Assert.True(success);
            Assert.Equal(5, store.GetInventory(Product.Shampoo));
        }
    }
}
```



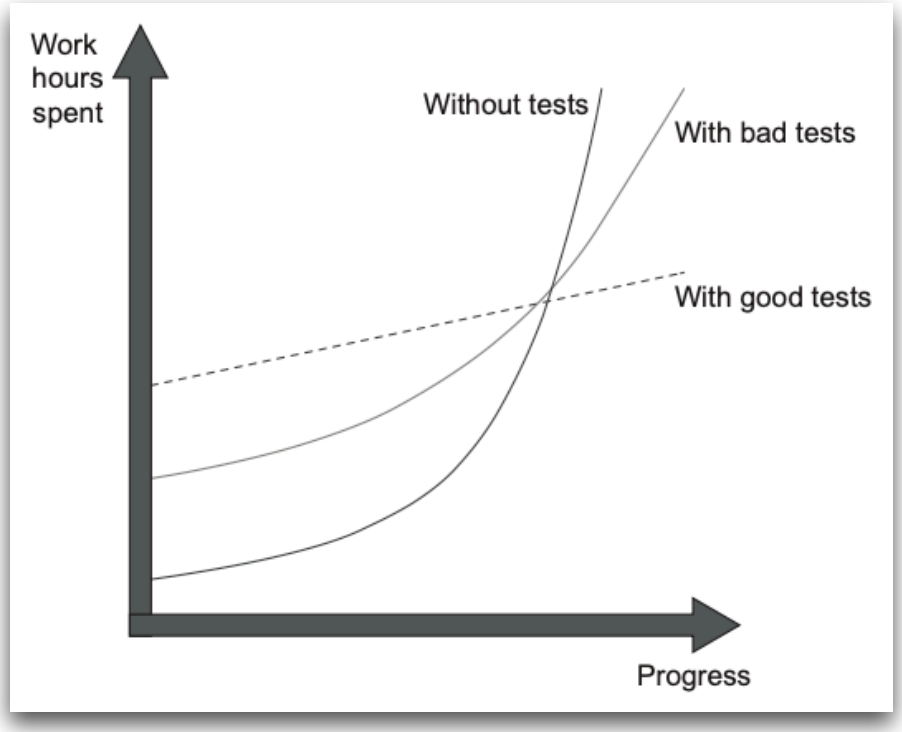
Integration tests

- **Out-of-process dependencies**
  - **Managed dependencies** (out-of-process dependencies you have full control over)
  - **Unmanaged dependencies** (out-of-process dependencies you don’t have full control over)
- Use **real instances of managed dependencies**; replace unmanaged dependencies with mocks.
- Assert that something did get called, as well as didn't get called.



The Value of tests

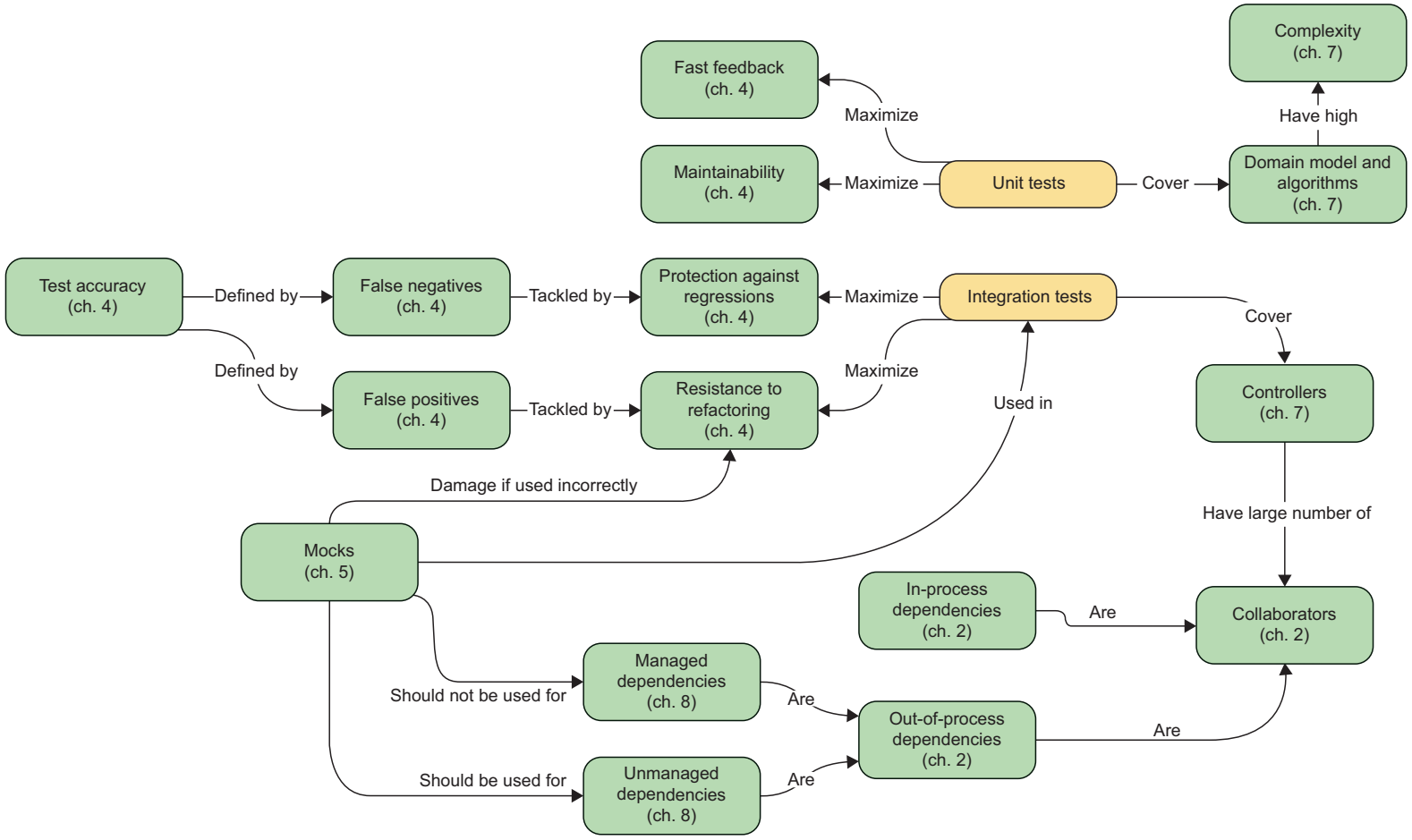
- Strong **negative indicator for bad code design**
- If not easily testable, refactor towards
  a) controllers (orchestrating) and
  b) domain model / business logic / algorithms
- Testable design is not only testable but also easy to maintain
- Sustainable long term Progress of the project
- Writing **good tests** is equally important to writing tests. Delete tests with neg net worth.



Conclusion

- For test doubles there are only two main categories: **mocks and stubs**
- The classical school advocates for **substituting only dependencies that are shared between tests**, which almost always translates into out-of-process dependencies
- Mockist (London) school encourages the use of mocks for all but immutable dependencies and **doesn't differentiate** between **intra-system and inner-system communications**.
  => often leads to tests that couple to implementation details and thus lack resistance to refactoring
- Always **test observable outcomes, not implementation details**
- **Mocks are a strong negative indicator for good code design.** If you need to use a lot of mocks, check if something is wrong with the architecture (e.g. hexagonal architecture, functional architecture) or with the APIs of your components
- **Input-output unit testing** has the highest resistance to refactoring and shortest testing code (readability)

Summary of this presentation



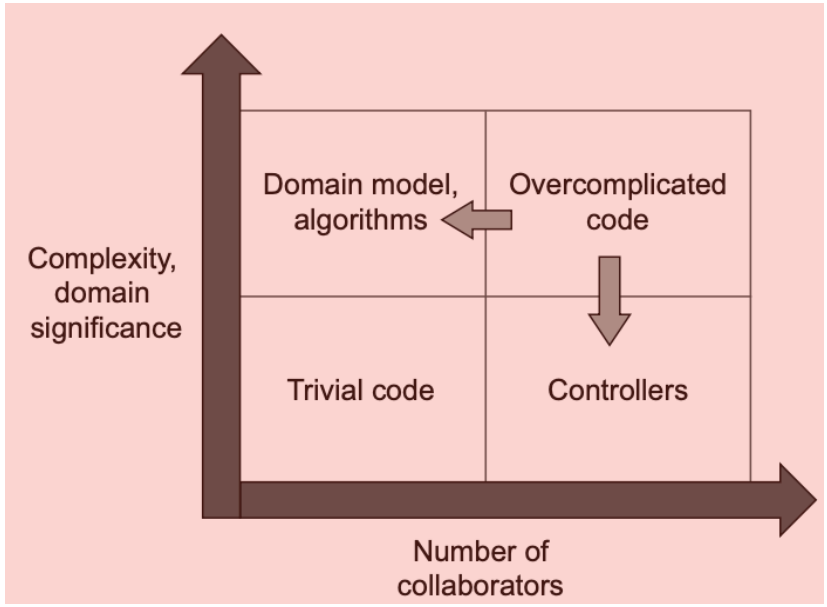
Thank you!
Questions?

II. Practical Part

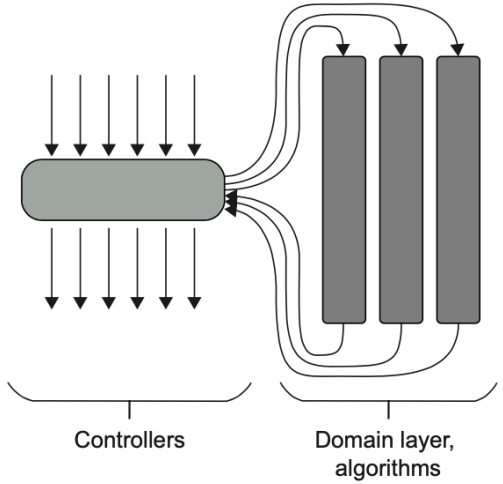
Take away Rules from theoretical part

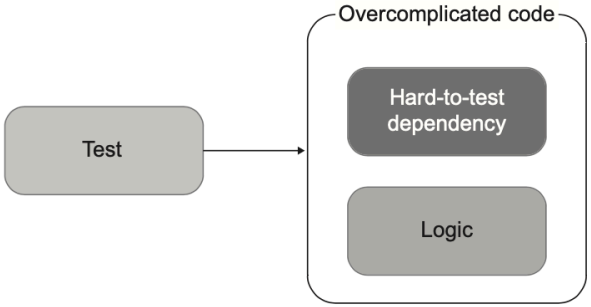
1. Test observable behaviour, never internals
2. Don't aim for 100% code coverage: only test the business logic (classes) via Unit tests, and controllers via integration tests (side effect: tests also business logic). Do not test complex code or trivial code.
3. Never use mocks in a unit test (mock only in integrations tests for controllers on the edge of an unmanaged dependency)
4. Input-Output unit testing style has shortest tests (high maintainability) and highest resistance to refactoring
5. Never assert a call to a stub
6. Integration test: introduce interfaces only if you need to mock an unmanned out-of-process dependency in your test. No "one class -> one interface" code.
   Genuine abstractions are discovered (ex post), not invented.

Refactoring towards good Unit Tests

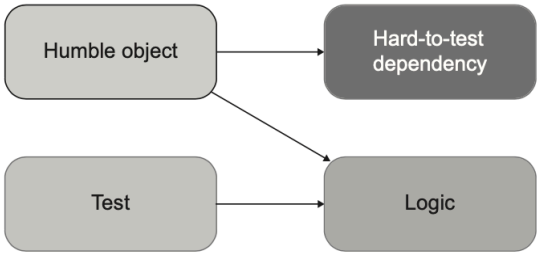


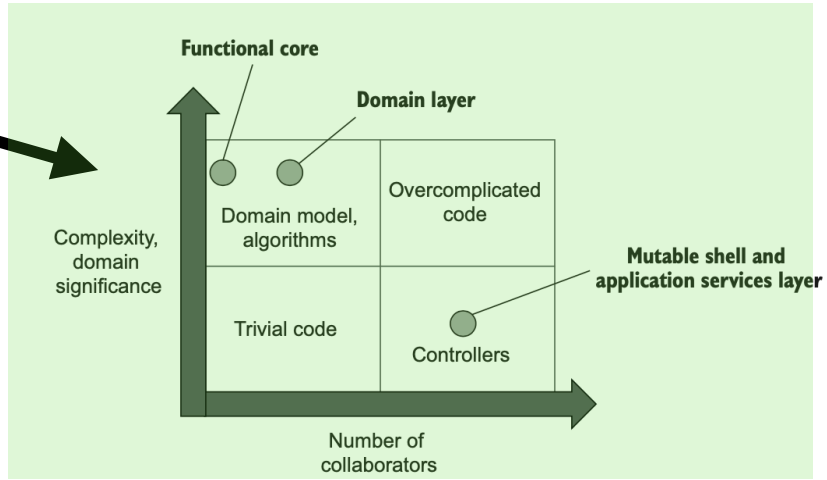
See: Domain-Driven Design: Tackling Complexity in the Heart of Software by Eric Evans (Addison-Wesley, 2003)





Using the humble object pattern





Refactoring towards good Unit Tests

The sample project is a **customer management system (CRM)** that handles user registrations. All users are stored in a database. The system currently supports only one use case: **changing a user's email**. There are three business rules involved in this operation:

- If the user's email belongs to the company's domain, that user is **marked as an employee**. Otherwise, they are treated as a customer.
- The system must **track the number of employees** in the company. If the user's type changes from employee to customer, or vice versa, this number must change, too.
- When the email changes, the system must **notify external systems** by sending a message to a message bus.

Refactoring Example

The sample project is a **customer management system (CRM)** that handles user registrations. All users are stored in a database. The system currently supports only one use case: **changing a user's email**. There are three business rules involved in this operation:

- If the user's email belongs to the company's domain, that user is **marked as an employee**. Otherwise, they are treated as a customer.
- The system must **track the number of employees** in the company. If the user's type changes from employee to customer, or vice versa, this number must change, too.
- When the email changes, the system must **notify external systems** by sending a message to a message bus.

```
public class User
{
    public int UserId { get; private set; }
    public string Email { get; private set; }
    public UserType Type { get; private set; }

    public void ChangeEmail(int userId, string newEmail)
    {
        object[] data = Database.GetUserById(userId);
        UserId = userId;
        Email = (string)data[1];
        Type = (UserType)data[2];

        if (Email == newEmail)
            return;

        object[] companyData = Database.GetCompany();
        string companyDomainName = (string)companyData[0];
        int numberOfEmployees = (int)companyData[1];

        string emailDomain = newEmail.Split('@')[1];
        bool isEmailCorporate = emailDomain == companyDomainName;
        UserType newType = isEmailCorporate
            ? UserType.Employee
            : UserType.Customer;

        if (Type != newType)
        {
            int delta = newType == UserType.Employee ? 1 : -1;
            int newNumber = numberOfEmployees + delta;
            Database.SaveCompany(newNumber);
        }

        Email = newEmail;
        Type = newType;

        Database.SaveUser(this);
        MessageBus.SendEmailChangedMessage(UserId, newEmail);
    }
}

public enum UserType
{
    Customer = 1,
    Employee = 2
}
```

Retrieves the user's current email and type from the database

Retrieves the organization's domain name and the number of employees from the database

Sets the user type depending on the new email's domain name

Updates the number of employees in the organization, if needed

Persists the user in the database

Sends a notification to the message bus

Own Examples

- Shop Monolith: Update.php
- Media Manager: Media Persister

How can we improve our testing suite?

Discussion and Brain Storming

My Propositions:

- we use these definitions here for **test doubles**: '**mock**' and '**stub**'
- We use these definitions here for **unit test** and **integration test**
- We use the **classic school** approach
- In cases where we have *clear, stable requirements* (and little exploration) we use **TDD**

- Ideas
  - Idea 1
  - Idea 2
  - ....

Test Driven Design

Reasons not to do it - misconceptions

- tests are slow to write;
- they “need” to test every little thing;
- you “need” to write tests for every piece of code;
- when we change the implementation we also need to change the tests;
- acceptance tests (which simulate a user clicking on a screen), are slow and too unstable because they need to change whenever the UI changes;
- they are slow to run;
- in sum, tests are annoying because they make us slow and none of us wants to be slow, we want to be the guy that quickly delivers reliable functionality!

Source: DevTernity 2017: Ian Cooper - TDD, Where Did It All Go Wrong
Cited from: https://herbertograca.com/2018/08/27/distillation-of-tdd-where-did-it-all-go-wrong/

Original source: Kent Beck: Test-Driven Development: By Example (Addison-Wesley Professional, 2002

Podcast on TDD:
https://changelog.com/gotime/185

Test Driven Design

How Kent Beck envisioned it

- Refactoring is the process of changing an implementation while maintaining the same behaviour;
- One of the promises of unit tests is that, when refactoring an implementation, we can use unit tests to make sure the implementation still yields the expected results;
- Focusing on testing methods or classes creates tests that are hard to maintain because we will need to update them when we refactor the implementation of the behaviour. Furthermore, they don't test the behaviour we want to preserve, otherwise, we wouldn't need to update them when we refactor the implementation;
- **Avoid testing implementation details, test behaviours;**
- The trigger to add a new test is not the creation of a new method, nor the creation of a new class. The **trigger is implementing a new requirement**;
- The Unit under test is not a class nor a method, it is a module. A class may be the entire module, or a class may be the module facade, but many classes are just implementation details of a module;
- Test the stable public API of a module;
- Write tests to cover the use cases or stories;
- Do not test using Acceptance TDD, use developer tests using the implementation language, they are faster and more stable;
- Use the "Given When Then" model;
- The unit of isolation is not the class under test, but the tests themselves. Tests need to run isolated from each other, but they can and should test several classes working together if that is what is needed to test the behaviour.
- **Avoid mocks at all costs,** use them only to isolate the tests on the module boundaries;
- We avoid the file system and the database, to help isolate tests and make tests faster. Those are, most of the time, module boundaries.

Source: DevTernity 2017: Ian Cooper - TDD, Where Did It All Go Wrong, Cited from: https://herbertograca.com/2018/08/27/distillation-of-tdd-where-did-it-all-go-wrong/

Thank you!