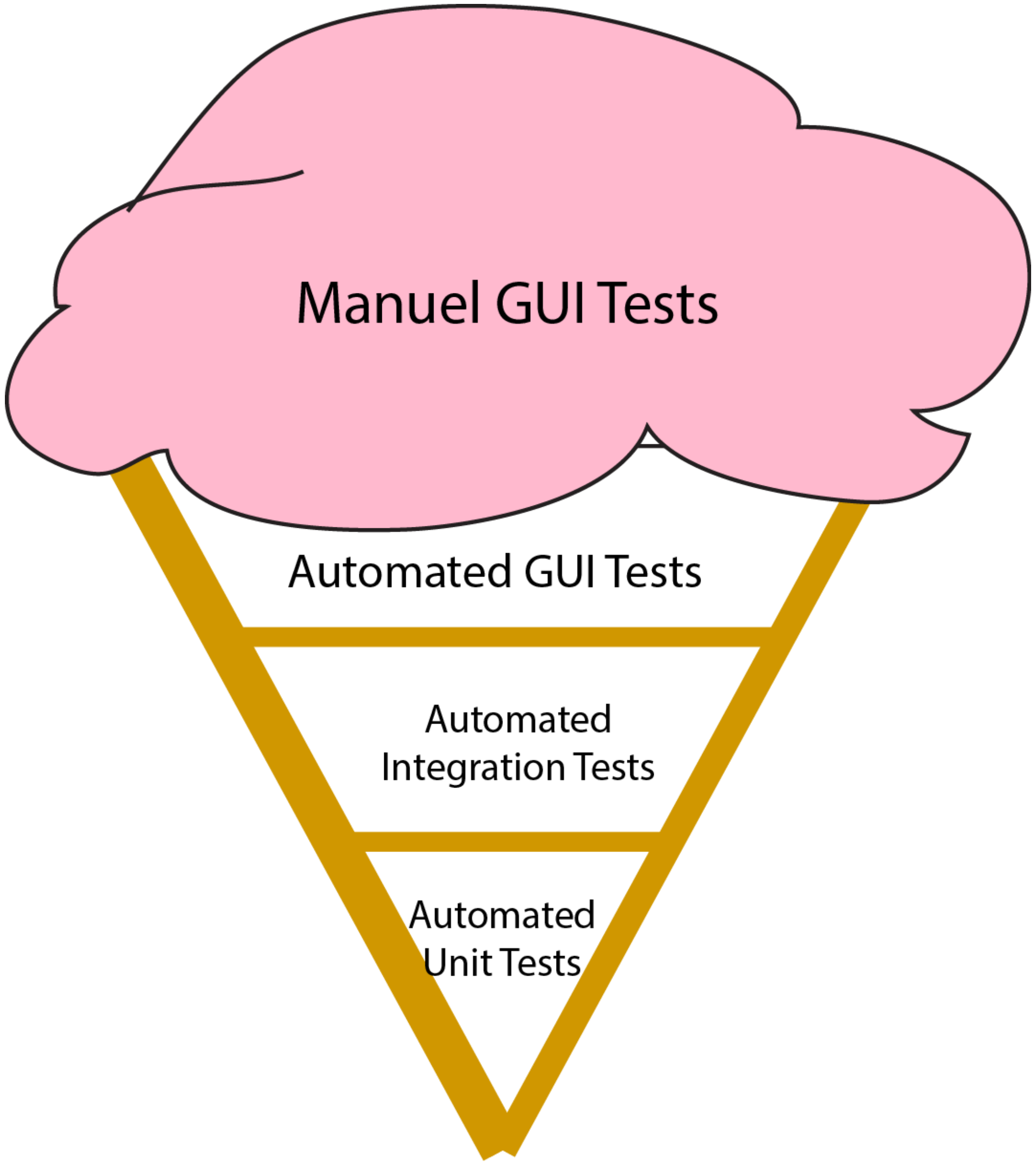
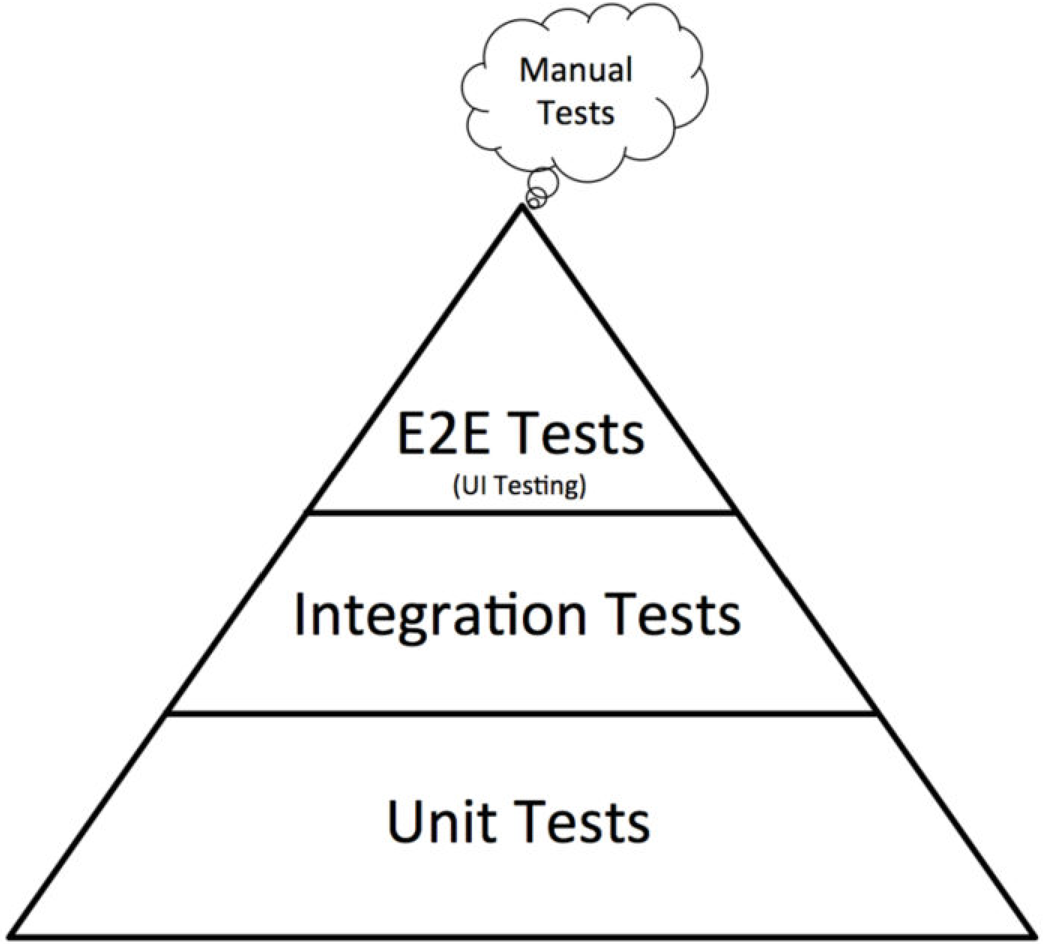
Testing


Manuel GUI Tests
Automated GUI Tests
Automated
Integration Tests
Automated
Unit Tests


Manual
Tests
E2E Tests
(UI Testing)
Integration Tests
Unit Tests

Testing

- unit tests - very small, independent code
- integration tests - some components interconnected (DB)
- system tests (end-to-end) - from the highest level (REST API, GUI)
- manual tests - well...

Unit testing

- small **unit** of code
- simple
  - well-known structure
  - DAMP - Descriptive and Meaningful Phrases
  - cyclomatic complexity = 1
- fast - live testing & CI
- independent - from environment, from each other, **idempotent**
- ideal test:
  - *easy* to write
  - *simple* to read
  - *trivial* to maintain

Unit testing - when it is Not

- talks to a database
- communicates over the network
- uses a file system
- calls `DateTime.Now` or `Random`
- depends on other non-deterministic behaviour
- can't run in parallel with another
- must run before/after another
- requires something from the environment

Unit testing

- structure:
    - Arrange, Act, Assert
    - Given, When, Then (BDD)
- State preparation (arrange):
    - Object Mother
    - Assembler (*Builder*-ish)
- Sprawdzenie
    - Assert Object (*Specification*-ish)

Object Mother

Set of helper methods to create data for various scenarios:

```
var basket = CreateWithoutDiscount();
var basket = CreateWithSmallDiscount();
var basket = CreateWithLargeDiscount();
var basket = CreateWithoutDiscountButSpecialOffer();
var basket = CreateWithSmallDiscountAndSpecialOffer();
...
// expotential explosion
```

Fluent Builder (Assembler)

Fluent API to build test data:

```
var basket = new BasketBuilder()
        .Build();
```

```
var basket = new BasketBuilder()
         .WithSmallDiscount()
         .Build();
```

```
var basket = new BasketBuilder()
         .WithLargeDiscount()
         .Build();
```

```
var basket = new BasketBuilder().
         .WithSmallDiscount()
         .WithSpecialOffer(type: Product.Book)
         .Build();
```

So we are building our DSL for test data.

Assembler - Bogus

(aka old Faker.NET)

```
var testUsers = new Faker<User>("pl")
    .WithRecord()
    .StrictMode(true)
    .RuleFor(u => u.FirstName, f => f.Name.FirstName())
    .RuleFor(u => u.LastName, f => f.Name.LastName())
    .RuleFor(u => u.DateOfBirth, f => f.Person.DateOfBirth)
    .RuleFor(u => u.Email, (f, u) => f.Internet.Email(u.FirstName, u.LastName))
    .RuleFor(u => u.Subscription, f => f.PickRandom<Subscription>())
    .Generate(10);
```

```
User { FirstName = Tamara, LastName = Krzeminski, Email = Tamara72@gmail.com, ... }
User { FirstName = Arseniusz, LastName = Fratczak, Email = Arseniusz_Fratczak83@hotmail.com, ... }
User { FirstName = Stefan, LastName = Kowalczuk, Email = Stefan.Kowalczuk@gmail.com, ... }
User { FirstName = Patrycja, LastName = Chmielewski, Email = Patrycja.Chmielewski@yahoo.com, ... }
User { FirstName = Damian, LastName = Kopczynski, Email = Damian_Kopczynski@yahoo.com, ... }
```

Good patterns

- Exceptions verification
- Mocking environment
  - Mocks for Commands, Stubs for Queries (in CQS sense)
  - Mock-free approach

Mocking environment - exceptions verification

Mixed Act and Assert:

```
[Test]
public void InsertTestNameHere() {
    var input = "a string";
    Assert.Throws<FormatException>(() => int.Parse(input));
}
```

Mocking environment - exceptions verification

Mixed Act and Assert:

```
[Test]
public void InsertTestNameHere() {
    var input = "a string";
    Assert.Throws<FormatException>(() => int.Parse(input));
}
```

Better but still an assert (fails immediately if no exception is thrown)

```
[Test]
public void InsertTestNameHere() {
    var input = "a string";
    var exception = Assert.Catch(() => int.Parse(input));
    Assert.IsInstanceOf<FormatException>(exception);
}
```

Mocking environment - exceptions verification

Mixed Act and Assert:

```
[Test]
public void InsertTestNameHere() {
    var input = "a string";
    Assert.Throws<FormatException>(() => int.Parse(input));
}
```

Better but still an assert (fails immediately if no exception is thrown)

```
[Test]
public void InsertTestNameHere() {
    var input = "a string";
    var exception = Assert.Catch(() => int.Parse(input));
    Assert.IsInstanceOf<FormatException>(exception);
}
```

Much better - clean AAA in xUnit:

```
[Fact]
public void InsertTestNameHere() {
    var input = "a string";
    var exception = Record.Exception(() => int.Parse(input));
    Assert.NotNull(exception);
    Assert.IsType<FormatException>(exception);
}
```

Mocking environment - file system

- create file system abstraction and inject it

```
interface IFileSystem
{
bool FileExists(string fileName);
DateTime GetCreationDate(string fileName);
...
}
```

- heavy-weight - stub **`System.IO.File`** with TypeMock Isolator
- reuse others work - System.IO.Abstractions

Mocking environment - `HttpClient`

- endless discussion about mocking **`HttpClient`** which has no interface, virtual methods and abstract base class…
- create *http client* abstraction and inject it - tiresome…
- **`HttpClient`** logic depends on injectable **`HttpMessageHandler`** and its **`SendAsync`** (protected…)
  - derive from **`HttpMessageHandler`** to create your custom mock
  - **`Moq`** it with protected mocks
  - reuse others work - MockHttp with **`MockHttpMessageHandler`**
- **`IHttpClientFactory`**? Mock its **`CreateClient`** to return HttpMock:

```
httpClientFactoryMock.CreateClient().Returns(fakeHttpClient);
```

Test doubles

- Fake - has some kind of working implementation, only partial/limited - not suitable for production (e.g. in-memory database)
- Stub - coded responses to calls made during the test
- Mock - as above but with verification of what was called (aka **`Verify`**)

Mocks - "loose mocks"

- **Loose mocks** (default in *Moq*) - *"Will never throw exceptions, returning default values when necassary (`null` for reference types, zero for value types or empty for enumerables and arrays)"*
- used when you want to verify that an expected method has been called with the proper parameters
- yes, it exposes our dependency behaviour (implementation detail)

Mocks - "loose mocks"

- **Loose mocks** (default in *Moq*) - *"Will never throw exceptions, returning default values when necassary (`null` for reference types, zero for value types or empty for enumerables and arrays)"*
- used when you want to verify that an expected method has been called with the proper parameters
- yes, it exposes our dependency behaviour (implementation detail)

```
var mockService = new Mock<ISomeService>();
mockService.Setup(x => x.DoWork()).Returns(200);
var handler = new Handler(..., mockService.Object);

handler.Handle(...);

mockService.Verify(x => x.DoWork()); // Fails if DoWork method was not called (but other may be called to)
```

Mocks - "loose mocks"

- **Loose mocks** (default in *Moq*) - *"Will never throw exceptions, returning default values when necassary (`null` for reference types, zero for value types or empty for enumerables and arrays)"*
- used when you want to verify that an expected method has been called with the proper parameters
- yes, it exposes our dependency behaviour (implementation detail)

```
var mockService = new Mock<ISomeService>();
mockService.Setup(x => x.DoWork()).Returns(200);
var handler = new Handler(..., mockService.Object);

handler.Handle(...);

mockService.Verify(x => x.DoWork()); // Fails if DoWork method was not called (but other may be called to)
```

```
var mockService = new Mock<ISomeService>();
var handler = new Handler(..., mockService.Object);

handler.Handle(...); // Won't fail, will just use DoWork returning null/0/... 😇
```

Mocks - "strict mocks"

- **Strict mocks** - *"Causes the mock to throw exceptions for invocations that don't have a corresponding setup"*
- used to verify that only expected methods have been called and no other
- yes, they expose implementation details so much more than loose mocks 🙄

Mocks - "strict mocks"

- **Strict mocks** - *"Causes the mock to throw exceptions for invocations that don't have a corresponding setup"*
- used to verify that only expected methods have been called and no other
- yes, they expose implementation details so much more than loose mocks 🙄

```
var mockService = new Mock<ISomeService>(MockBehavior.Strict);
mockService.Setup(x => x.DoWork()).Returns(200);
var handler = new Handler(..., mockService.Object);

handler.Handle(...);

mockService.Verify(x => x.DoWork());
```

Mocks - "strict mocks"

- **Strict mocks** - *"Causes the mock to throw exceptions for invocations that don't have a corresponding setup"*
- used to verify that only expected methods have been called and no other
- yes, they expose implementation details so much more than loose mocks 🙄

```
var mockService = new Mock<ISomeService>(MockBehavior.Strict);
mockService.Setup(x => x.DoWork()).Returns(200);
var handler = new Handler(..., mockService.Object);

handler.Handle(...);

mockService.Verify(x => x.DoWork());
```

```
var mockService = new Mock<ISomeService>(MockBehavior.Strict);
var handler = new Handler(..., mockService.Object);

handler.Handle(...); // Fails because no Setup for DoWork
```

Mocks - "strict mocks"

- **Strict mocks** - *"Causes the mock to throw exceptions for invocations that don't have a corresponding setup"*
- used to verify that only expected methods have been called and no other
- yes, they expose implementation details so much more than loose mocks 🙄

```
var mockService = new Mock<ISomeService>(MockBehavior.Strict);
mockService.Setup(x => x.DoWork()).Returns(200);
var handler = new Handler(..., mockService.Object);

handler.Handle(...);

mockService.Verify(x => x.DoWork());
```

```
var mockService = new Mock<ISomeService>(MockBehavior.Strict);
var handler = new Handler(..., mockService.Object);

handler.Handle(...); // Fails because no Setup for DoWork
```

```
var mockService = new Mock<ISomeService>();
mockService.Setup(x => x.DoWork()).Returns(...);
var handler = new Handler(..., mockService.Object);
handler.Handle(...);
mockService.Verify(x => x.DoWork());
mockService.VerifyAll(); // Fails if DoWork method was not the only one called
```

Stubs

Stub - coded responses to calls made during the test, also we use **`Moq`** (😇)

```
// Arrange
var monitoringSystem = new Mock<IEngineMonitoringSystem>();
monitoringSystem.Setup(x => x.CurrentRPM).Returns(1200);

// Act

// Assert
// do not Verify anything on monitoringSystem stub
```

Stubs

Stub - coded responses to calls made during the test, also we use **Moq** (😇)

```
// Arrange
var monitoringSystem = new Mock<IEngineMonitoringSystem>();
monitoringSystem.Setup(x => x.CurrentRPM).Returns(1200);

// Act

// Assert
// do not Verify anything on monitoringSystem stub
```

As it's simple, we can even ignore using **Moq** and go manually:

```
class CustomerRepositoryStub : ICustomerRepository
{
  public List<Customer> GetCustomers()
  {
    return new List<Customer>()
    {
      new Customer("Konrad", "Kokosa", ...),
      new Customer("John", "Doe", ...)
    }
  }
}
```

Mocks for Commands, Stubs for Queries

Mocks vs Stubs

```
public interface IUserRepository
{
    User Read(int userId);
    void Create(int userId);
}

[Fact]
public void GetUserReturnsCorrectResult()
{
    var expected = new User();
    var td = new Mock<IUserRepository>();
    td.Setup(r => r.Read(It.IsAny<int>())).Returns(expected);
    var sut = new SomeController(td.Object);
    var actual = sut.GetUser(1234);
    Assert.Equal(expected, actual);
}
```

Mocks vs Stubs

```
public interface IUserRepository
{
    User Read(int userId);
    void Create(int userId);
}

[Fact]
public void GetUserReturnsCorrectResult()
{
    var expected = new User();
    var td = new Mock<IUserRepository>();
    td.Setup(r => r.Read(It.IsAny<int>())).Returns(expected);
    var sut = new SomeController(td.Object);
    var actual = sut.GetUser(1234);
    Assert.Equal(expected, actual);
}
```

Passing!!!

```
public User GetUser(int userId)
{
    return this.userRepository.Read(0);
}
```

Mocks vs Stubs

```
[Theory]
[InlineData(1234)]
[InlineData(9876)]
public void GetUserCallsRepositoryWithCorrectValue(int userId)
{
  var td = new Mock<IUserRepository>();
  var sut = new SomeController(td.Object);
  sut.GetUser(userId);
  td.Verify(r => r.Read(userId), Times.Once());
}
```

Mocks vs Stubs

```
[Theory]
[InlineData(1234)]
[InlineData(9876)]
public void GetUserCallsRepositoryWithCorrectValue(int userId)
{
  var td = new Mock<IUserRepository>();
  var sut = new SomeController(td.Object);
  sut.GetUser(userId);
  td.Verify(r => r.Read(userId), Times.Once());
}
```

Not passing - good

```
public User GetUser(int userId)
{
  return this.userRepository.Read(0);
}
```

Mocks vs Stubs

```
[Theory]
[InlineData(1234)]
[InlineData(9876)]
public void GetUserCallsRepositoryWithCorrectValue(int userId)
{
  var td = new Mock<IUserRepository>();
  var sut = new SomeController(td.Object);
  sut.GetUser(userId);
  td.Verify(r => r.Read(userId), Times.Once());
}
```

Not passing - good

```
public User GetUser(int userId)
{
  return this.userRepository.Read(0);
}
```

Passing!!!

```
public User GetUser(int userId)
{
  this.userRepository.Read(0);
  return this.userRepository.Read(userId);
}
```

Mocks vs Stubs

Zamiast weryfikować zachowanie (Mocks), sprawdźmy wynik (na podstawie danych ze Stub)

```
[Theory]
[InlineData(1234)]
[InlineData(9876)]
public void GetUserReturnsCorrectValue(int userId)
{
    var expected = new User();
    var td = new Mock<IUserRepository>();
    td.Setup(r => r.Read(userId)).Returns(expected);
    var sut = new SomeController(td.Object);

    var actual = sut.GetUser(userId);

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

Mocks vs Stubs

```
// Production code to describe phase of moon (JavaScript)
import * as moon from "astronomy";
import { format } from "date_formatter";

export function describeMoonPhase(date) {
  const visibility = moon.getPercentOccluded(date);
  const phase = moon.describePhase(visibility);
  const formattedDate = format(date);
  return `The moon is ${phase} on ${formattedDate}.`;
}
```

Mocks vs Stubs

```
// Production code to describe phase of moon (JavaScript)
import * as moon from "astronomy";
import { format } from "date_formatter";

export function describeMoonPhase(date) {
  const visibility = moon.getPercentOccluded(date);
  const phase = moon.describePhase(visibility);
  const formattedDate = format(date);
  return `The moon is ${phase} on ${formattedDate}.`;
}
```

```
// Interaction-based test of describeMoonPhase()
const moon = mocker.mockImport("astronomy");
const { format } = mocker.mockImport("date_formatter");
const { describeMoonPhase } = mocker.importWithMocks("describe_phase");

it("describes phase of moon", () => {
  const date = new Date();     // specific date doesn't matter

  mocker.expect(moon.getPercentOccluded).toBeCalledWith(date).thenReturn(999);
  mocker.expect(moon.describePhase).toBeCalledWith(999).thenReturn("PHASE");
  mocker.expect(format).toBeCalledWith(date).thenReturn("DATE");

  const description = describeMoonPhase(date);
  mocker.verify();
  assert.equal(description, "The moon is PHASE on DATE");
};
```

Mocks vs Stubs

```
// Production code to describe phase of moon (JavaScript)
import * as moon from "astronomy";
import { format } from "date_formatter";

export function describeMoonPhase(date) {
  const visibility = moon.getPercentOccluded(date);
  const phase = moon.describePhase(visibility);
  const formattedDate = format(date);
  return `The moon is ${phase} on ${formattedDate}.`;
}
```

```
// State-based test of describeMoonPhase() (JavaScript)
import { describeMoonPhase } from "describe_phase";

it("describes phase of moon", () => {
  const dateOfFullMoon = new Date("8 Dec 2022");    // a date when the moon was actually full
  const description = describeMoonPhase(dateOfFullMoon);
  assert.equal(description, "The moon is full on December 8th, 2022.";
});
```

- *don't* test all phases of the moon in your **`describeMoonPhase()`** tests, but *do* test them in your **`Moon`** tests
- *don't* check the intricacies of date formatting in your **`describeMoonPhase`** tests, but *do* test them in your **`format(date)`** tests.

`AutomaticGearBox` story

FluentAssertions

```
string actual = "ABCDEFGHI";
actual.Should()
       .StartWith("AB").And
       .EndWith("HI").And
       .Contain("EF").And
       .HaveLength(9);
```

```
IEnumerable numbers = new[] { 1, 2, 3 };
numbers.Should().HaveCount(4)
```

FluentAssertions

```
var recipe = new RecipeBuilder()
                    .With(new IngredientBuilder().For("Milk").WithQuantity(200, Unit.Milliliters))
                    .Build();

var exception = Record.Exception(() => recipe.AddIngredient("Milk", 100, Unit.Spoon));

exception.Should()
         .BeOfType<RuleViolationException>()
         .Which.Message.Should().Be("No milk.");
```

FluentAssertions tips

```
actual.Contains(expected).Should().BeTrue();
```

```
actual.Should().Contain(expected);
```

```
actual.Should().HaveCount(1);
// Expected collection to contain 1 item(s),
// but found 3.
```

```
actual.Should().ContainSingle();
// Expected collection to contain a single item,
// but found {"a", "b", "c"}.
```

```
actual[k].Should().Be(expected);
// Expected string to be "Expected" with a length of
// 8, but "Unexpected" has a length of 10.
```

```
actual.Should().HaveElementAt(k, expected);
// Expected "Expected" at index 0, but found
// "Unexpected".
```

Assembler + fluent assertions = ♥

```
[Fact]
public void GivenSampleGame_WhenPlayedEmojiCard_ThenPointsChangeAccordingly()
{
    GivenGameEngineRepository()
        .ForGameId(It.IsAny<Guid>().ToString())
        .WithState(GameState.PlayerA_Move)
        .WithPlayerA(player =>
            player.WithCardsInHand(CardDefinitionId.Feature_Social_Emoji)
                  .WithFeaturePoints(0)
                  .WithMemory(0)
                  .WithTicks(0))
        .IsPrepared();
    GivenGameEngineLoadedGame();

    GameEngine().ProcessCommand(
        new CardPlayedCommand()
        {
            GameId = Game.Id,
            PlayerSide = PlayerSide.A,
            CardId = CardDefinitionId.Feature_Social_Emoji
        });

    Game.GetPlayerState(PlayerSide.A).Should()
        .HaveEmptyHand().And
        .HaveOnTable(CardDefinitionId.Feature_Social_Emoji).And
        .HaveFeaturePoints(1).And
        .HaveMemory(2).And
        .HaveTicks(3);
}
```

Property-based testing

- Property-Based Testing with C# | Codit