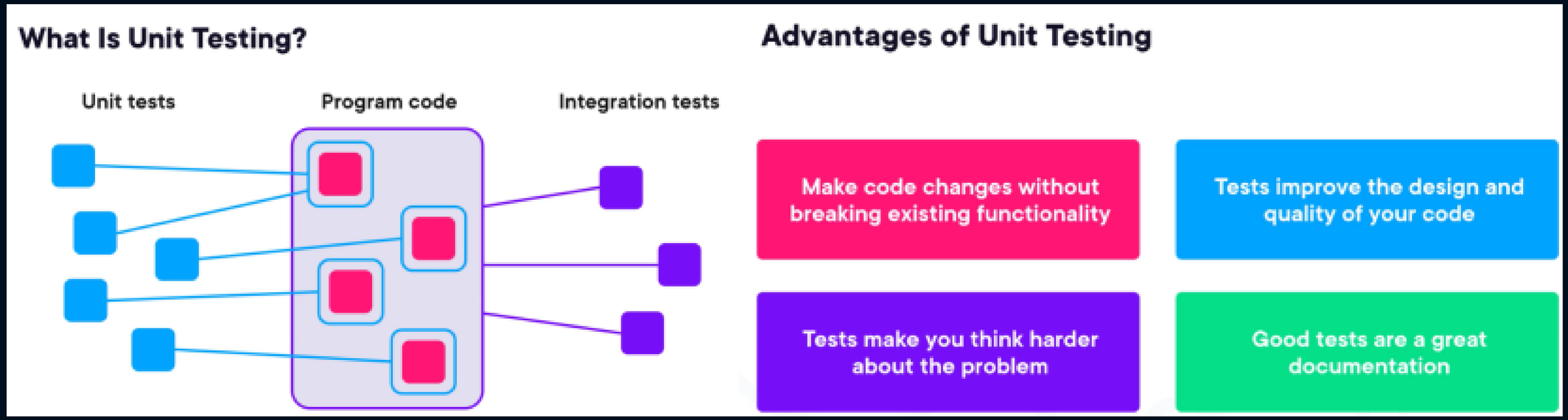
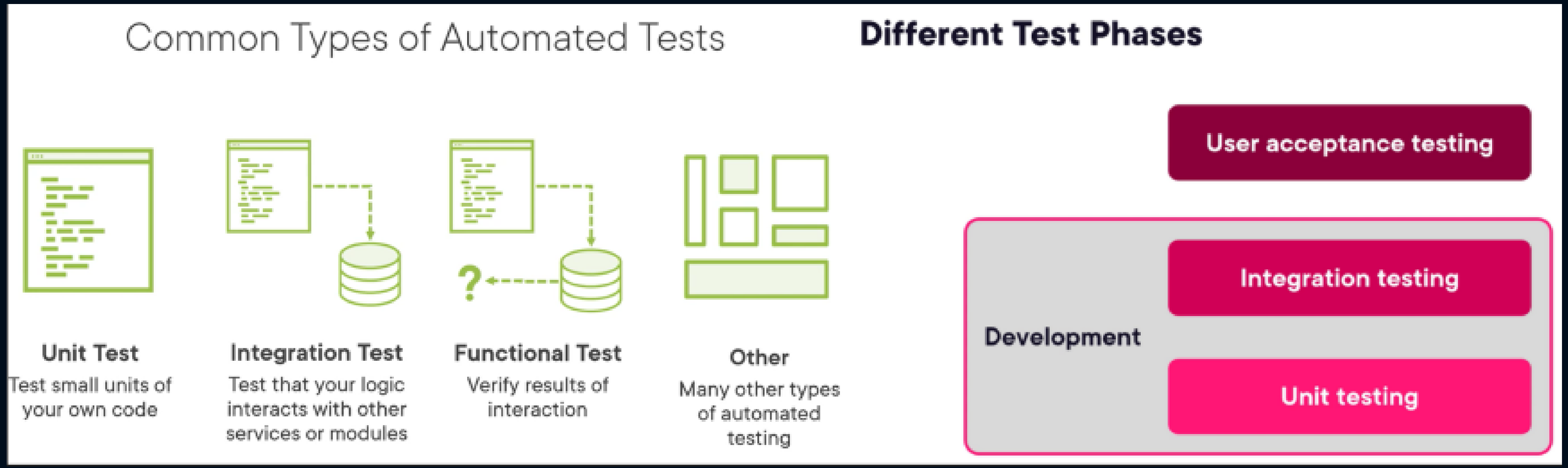
What is Unit Testing



A Unit test is code that will, in an automated way, invoke code to be tested. It will check an assumption about the behavior of the code under test.

Types of tests

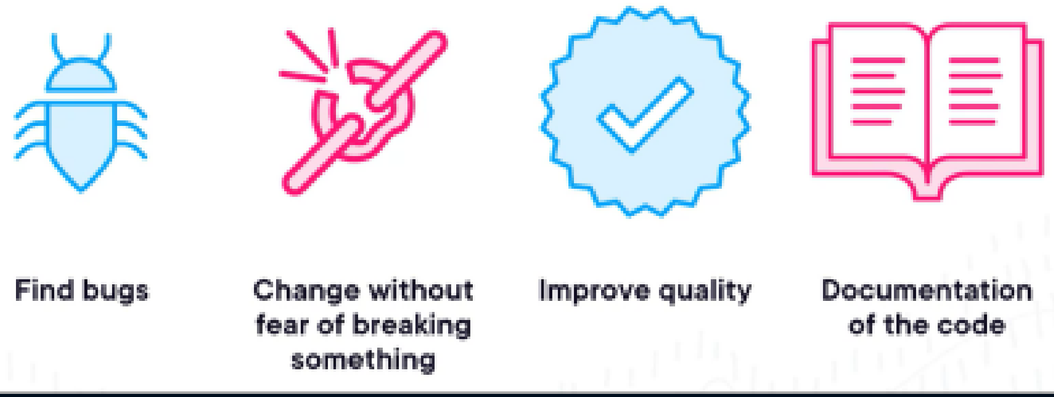


Wy do we need unit tests?

Unit Tests

- Test public API
- Run in isolation
- Consistent results
- Fast
- Often automated

Why Do We Need Unit Tests?



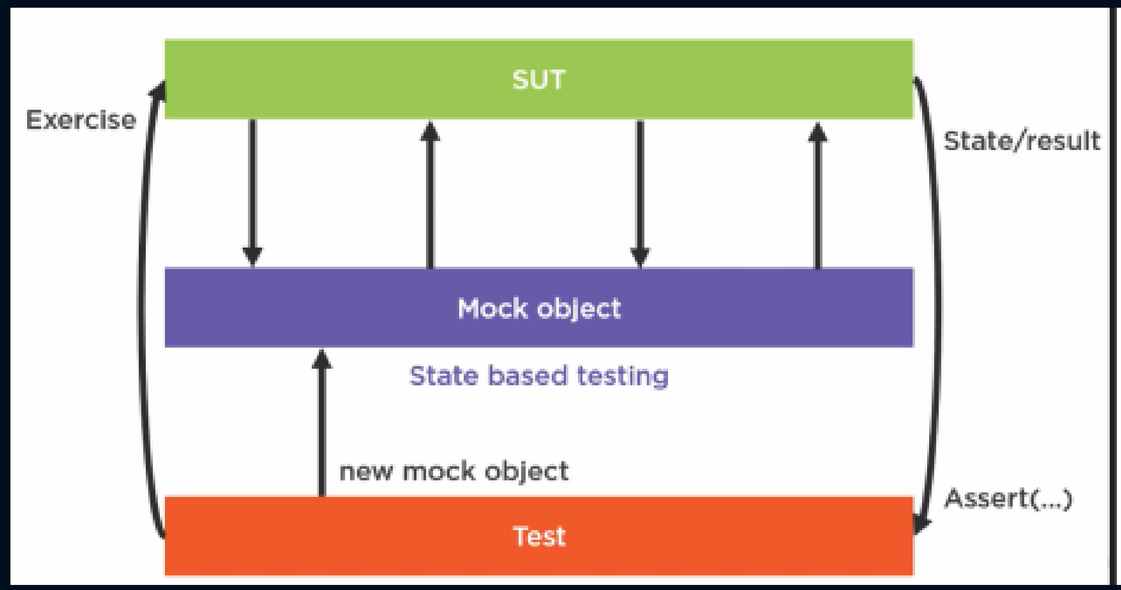
- Find bugs
- Change without fear of breaking something
- Improve quality
- Documentation of the code

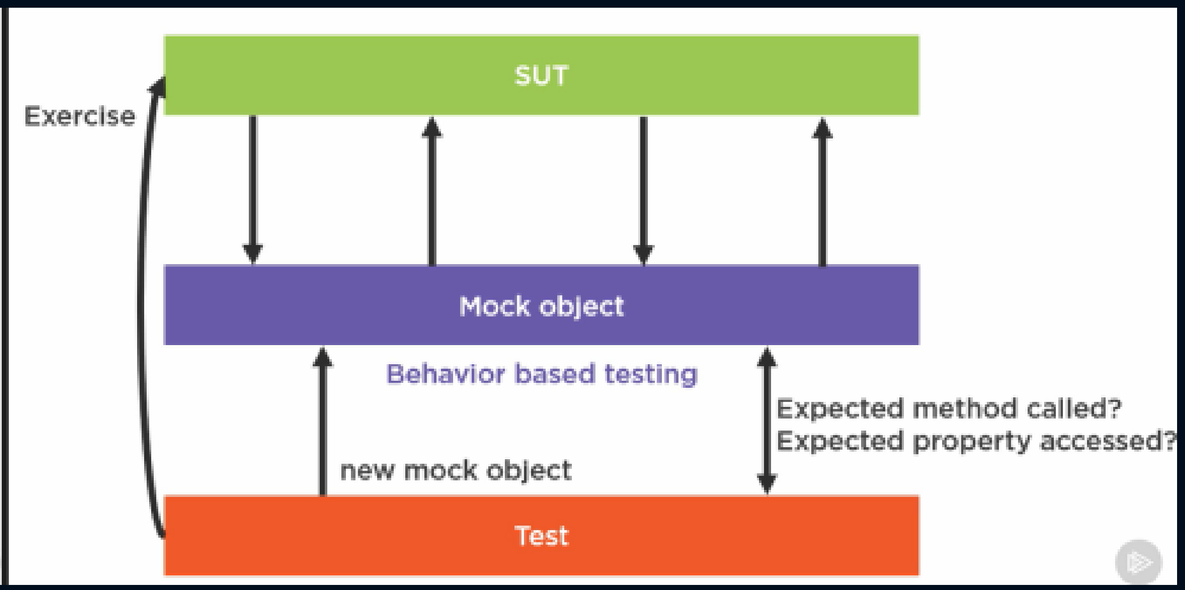
Data driven tests and exceptions

```
private static MachineDataItem Parse(string csvLine)
{
    var lineItems = csvLine.Split(';');

    if (lineItems.Length != 2)
    {
        throw new Exception();
    }

    return new MachineDataItem(lineItems[0], DateTime.Parse(lineItems[1]));
}

[Fact]
public void ShouldThrowExceptionForInvalidLine()
{
    // Arrange
    string[] csvLines = new[] { "Cappuccino" };

    // Act
    Assert.Throws<Exception>(()=> CsvLineParser.Parse(csvLines));
}
```

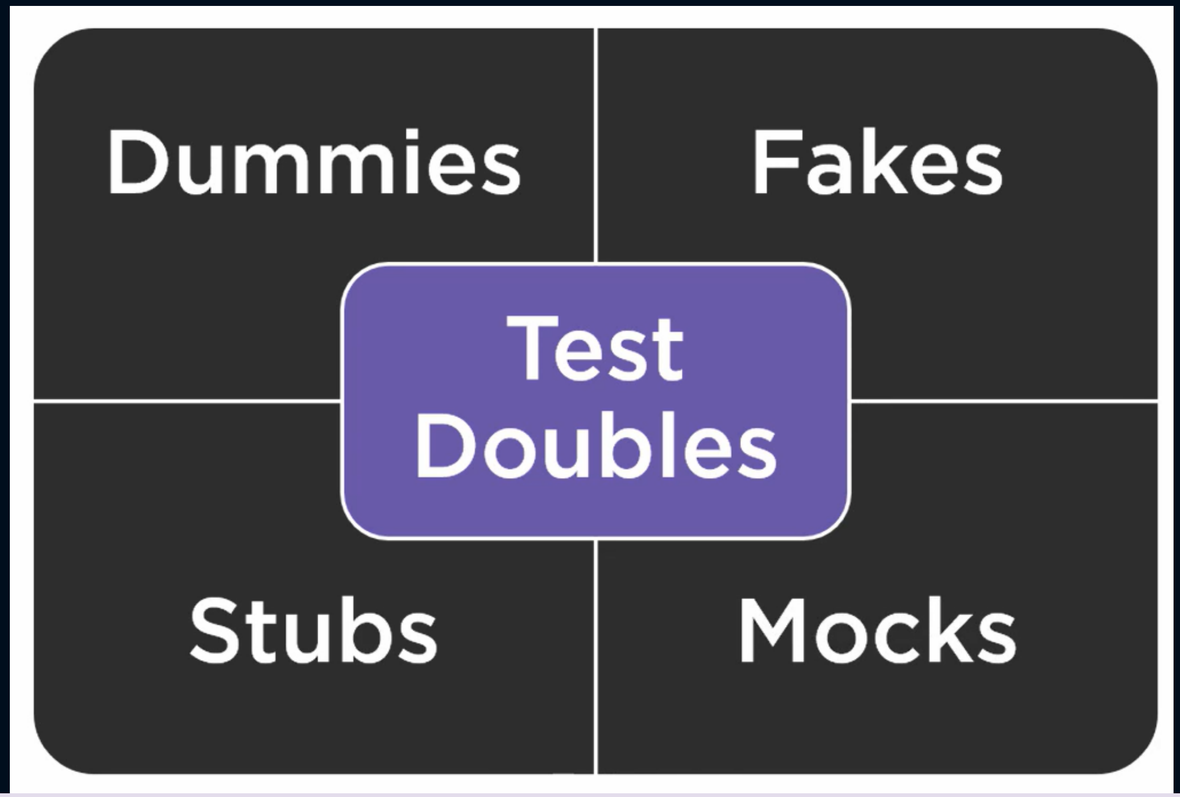
```
[InlineData("Cappuccino", "Invalid csv line")]
[InlineData("Cappuccino;InvalidDateTime", "Invalid datetime in csv line")]
[Theory]
public void ShouldThrowExceptionForInvalidLine(string csvLine,
    string expectedMessagePrefix)
{
    // Arrange
    var csvLines = new[] { csvLine };

    // Act and Assert
    var exception = Assert.Throws<Exception>(() => CsvLineParser.Parse(csvLines));

    Assert.Equal($"{expectedMessagePrefix}: {csvLine}", exception.Message);
}
```

Behavior testing and state based testing





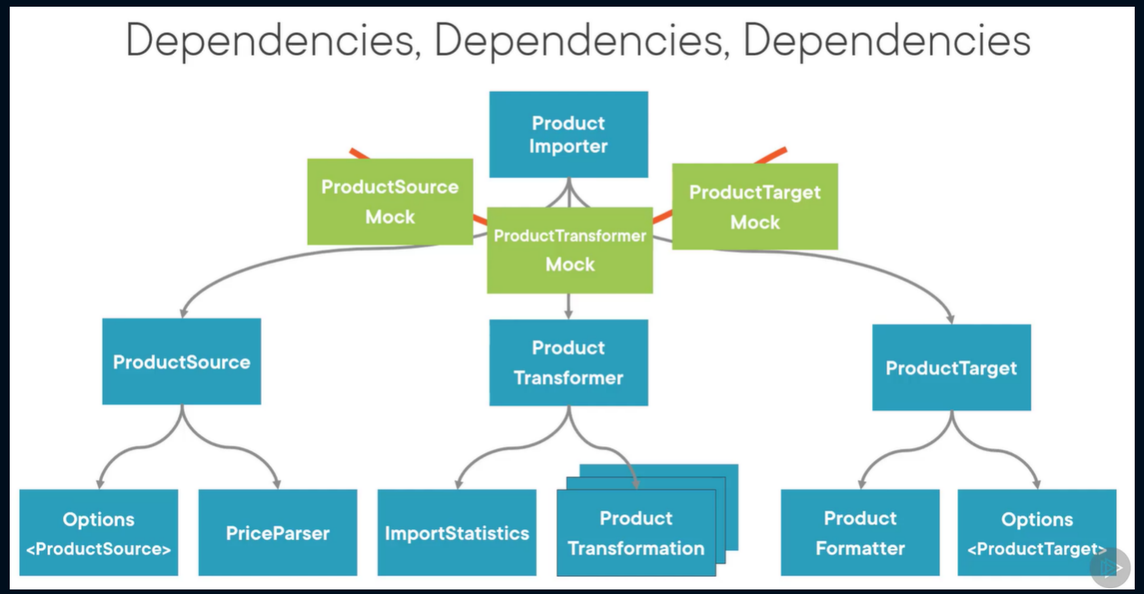
Test doubles



Test Double is a generic term for any case where you replace a production object for testing purposes.
Fakes: Working implementation; not suitable for production. e.g. EF Core In-memory provider.
Dummies: Passed around, never used/ accessed. Satisfy parameters.
Stubs: Provide answers to calls. Property gets and Method return values.
Mocks: Expect/verify calls. Properties and Methods

Dependency Injection and Unit Testing

Dependency injection plays a crucial role in unit testing, as it allows for better testability and isolation of dependencies. Dependency injection works hand in hand with mocking frameworks, which provide convenient ways to create and manage test doubles. These frameworks often integrate well with dependency injection containers and allow for seamless integration of test doubles into the codebase.



Mocking dependencies

```
[Theory]
[InlineData(0)]
0 references
public async Task WhenItReadsNProductsFromSource_ThenItWritesNProductsToTarget(int numberOfProducts)
{
    var productSource = new Mock<IProductSource>();
    var productTransformer = new Mock<IProductTransformer>();
    var productTarget = new Mock<IProductTarget>();
    var importStatistics = new Mock<IImportStatistics>();

    var subjectUnderTest = new ProductImporter(productSource.Object, productTransformer.Object,
        productTarget.Object, importStatistics.Object);

    var productcounter = 0;

    productSource
        .Setup(x => x.hasMoreProducts())
        .Callback(() => productcounter++)
        .Returns(() => productcounter <= numberOfProducts);
```

Example using MOQ

Mock objects

```
[Fact]
| 0 references
public void ValidateFrequentFlyerNumberForLowIncomeApplications()
{
    var mockValidator = new Mock<IFrequentFlyerNumberValidator>();

    mockValidator.Setup(x => x.ServiceInformation.License.LicenseKey).Returns("OK");

    var sut = new CreditCardApplicationEvaluator(mockValidator.Object);

    var application = new CreditCardApplication
    {
        FrequentFlyerNumber = "q"
    };

    sut.Evaluate(application);

    mockValidator.Verify(x => x.IsValid(It.IsAny<string>()));
}
```

```
public class GetCategoriesListQueryHandlerTests
{
    private readonly IMapper _mapper;
    private readonly Mock<IAsyncRepository<Category>> _mockCategoryRepository;

    0 references
    public GetCategoriesListQueryHandlerTests()
    {
        _mockCategoryRepository = RepositoryMocks.GetCategoryRepository();
        var configurationProvider = new MapperConfiguration(cfg =>
        {
            cfg.AddProfile<MappingProfile>();
        });

        _mapper = configurationProvider.CreateMapper();
    }

    [Fact]
    | 0 references
    public async Task GetCategoriesListTest()
    {
        var handler = new GetCategoriesListQueryHandler(_mapper,
          _mockCategoryRepository.Object);

        var result = await handler.Handle(new GetCategoriesListQuery(),
          CancellationToken.None);

        result.ShouldBeOfType<List<CategoryListVm>>();
```

- Improved test execution speed (slow algorithms, external resources)
- Support parallel development streams (real object not your developed, external contractor)
- Improve test reliability
- Reduce development/testing costs (External company bills per usage, Interfacing with mainframe)
- Test when non-deterministic dependency

Fakes

```
    [Fact]
    public async Task GetCategoriesListTest()
    {
        var handler = new GetCategoriesListQueryHandler(_mapper,
          _mockCategoryRepository.Object);

        var result = await handler.Handle(new GetCategoriesListQuery(),
          CancellationToken.None);

        result.ShouldBeOfType<List<CategoryListVm>>();

        result.Count.ShouldBe(4);
    }
```

```
[Fact]
public async void Save_SetCreatedByProperty()
{
    var ev = new Event() {EventId = Guid.NewGuid(), Name = "Test event" };

    _globoTicketDbContext.Events.Add(ev);
    await _globoTicketDbContext.SaveChangesAsync();

    ev.CreatedBy.ShouldBe(_loggedInUserId);
}
```

```
namespace GloboTicket.TicketManagement.Persistence.IntegrationTests
{
    public class GloboTicketDbContextTests
    {
        private readonly GloboTicketDbContext _globoTicketDbContext;
        private readonly Mock<ILoggedInUserService> _loggedInUserServiceMock;
        private readonly string _loggedInUserId;

        public GloboTicketDbContextTests()
        {
            var dbContextOptions = new DbContextOptionsBuilder<GloboTicketDbContext>
              ().UseInMemoryDatabase(Guid.NewGuid().ToString()).Options;

            _loggedInUserId = "00000000-0000-0000-0000-000000000000";
            _loggedInUserServiceMock = new Mock<ILoggedInUserService>();
            _loggedInUserServiceMock.Setup(m => m.UserId).Returns(_loggedInUserId);

            _globoTicketDbContext = new GloboTicketDbContext(dbContextOptions,
              _loggedInUserServiceMock.Object);
        }
```

Task

Using the Library API:

- Add Unit testing to each layer of your Library API
  - Unit tests for Controllers
  - Unit tests for Services and validations