Софтуерно тестване

ИТ Кариера

Учителски екип
Обучение за ИТ кариера
https://it-kariera.mon.bg/e-learning/

Съдържание

- Unit тестване
- nUnit
- Регресивно тестване
- Mocking (подпъхване на функционалност)
- Покритие на кода
- Интеграционно тестване
- Непрекъсната интеграция
- Travis CI

Софтуерно тестване

- Процеса по проверяване и валидиране, че дадено приложение няма бъгове, не прави грешки, отговаря на техническите изисквания, отговаря на потребителските изисквания, като се справя с всички по-специални сценарии
- Процеса по тестване не цели само да открие грешките, но и да предприеме мерки за подобряване функционалността и прозиводителността на проекта

Софтуерно тестване

- Отговаря на два въпроса:
  - Правилно ли изграждаме продукта?
  - Правилния продукт ли изграждаме?

Софтуерно тестване – ръчно тестване

- Тестване на софтуер без използването на скрипт или инструмент за автоматизация
- Разработчика приема ролята на краен потребител
- Изпозлват се планове за тестване и тестови сценарии

Софтуерно тестване – автоматизирано тестване

- Разработчика изпозлва скрипт или инструмент за автоматизация на тестването
- Използва се, за да се проведат многократно и бързо тестовете, които са проведени ръчно
- Подобрява покритието на тестовете, точността и спестява време и пари

Софтуерно тестване – Test Driven Development

- Концпеция, според която, преди да се напише код, се пишат тестове, които да играят ролята на спецификация какво точно трябва да се случва при изпълнението на дадения код
- Задължава разработчика да изгради класовете си коректно
- Задължава разработчика да спазва KISS (keep it stupid simple) принцип на работа

Софтуерно тестване – покритие на кода

- Измерва колко реда или блокове код се изпълняват повреме на провеждането на тестовете
- Използва се като маркер, за да се подобрят тестовете с цел да се изпълнява възможно най-голяма част от кода

Unit тестване

Unit тестване

- Тип софтуерно тестване, при който те тестват отделни компоненти (разбирай отделни класове, файлове)
- Изолира секция от кода и валидира, че извършва правилно задачата си
- Обикновено се пише от софтуерен разработчик
- Хваща грешките в кода на много ранен етап в софтуерната разработка

Unit тестване

- Добрите unit тестове могат да служат и като документация на проекта

Unit тестване – логически фази на тестване

- Arrange, Act, Assert
- Arrange – фаза за подготвяне на тествания обект, инициализация на тестови данни и т.н.
- Act – фаза за извикване на тествания метод, свойство и т.н.
- Assert – фаза за сравнение на получения резултат, на крайно състояние с очакваните такива

Unit тестване – предимства

- Разработчици, които искат да научат каква функционалност предлага дадена единица могат да проверят теста, за да получат добра представа
- Позволява даден код да бъде рефакториран на по-късен етап и подсигурява, че модула все още работи правилно
  - При писане на тест за всеки метод (функция)
- Позволява тестване на проекта, без да се чака завършването на другите части

Unit тестване – недостатъци

- Не може да се разчита на unit тестове да открият абсолютно всяка грешка в проекта
- По същество unit тестовете се фокусират върху дадена единица – т.е. не могат да хванат интеграционните грешки в проекта

Unit тестване – практики

- Пишете тестове възможно най-често – колкото повече код без тестове имате, от толкова пътища може да възникне грешка
- Грешките, открити при тестването, да се оправят преди да се продължи с разработката на нова функционалност
- Преди промени в даден код, бъдете сигурни, че за него има съответен тест и теста минава
- Следвайте чиста и ясна конвенция за именоване на тестовете

Unit тестване – практики

- Тествайте само един код едновременно
- Тестовете трябва да са независими

Unit тестване – инструменти

- nUnit - .NET
- Emma - Java
- Junit - Java
- PHPUnit - PHP

nUnit

nUnit

- Фреймуърк за unit тестване на всички .NET програмни езици
- С отворен код
- Тестовете могат да се провеждат паралелно
- Силна поддръжка за data driven тестване
- Всеки тест може да бъде добавен към една или повече категории
  - Позволява селективно провеждане на тестове

nUnit + Visual Studio

1. Създава се нов проект – библиотека
2. Добавя се референция към тествания проект
3. Нужни пакети: NUnit, NUnit3TestAdapter, Microsoft.NET.Test.Sdk
4. Build-ва се solution-а, за да се подсигурим, че не възникват никакви грешки

nUnit + Visual Studio

- Използват се класове, за групиране на тестовете
- Създават се методи, като всеки метод представлява един тест
  - Методите, трябва да са анотирани с атрибута [Test]
- За да проверим резултатите от теста използваме т.нар. Assertions

nUnit – провеждане на тестовете

- Във Visual Studio:
  - Чрез Visual Studio Test Explorer
- От конзолата/терминала:
  - В директорията на проекта, съдържащ тестовете, използваме командата dotnet test
  - С командата dotnet test –list-tests можем да изкараме списък на всички тестове

nUnit – атрибути

- [TestFixture] – маркира клас, съдържащ тестове (атрибутът се поставя по избор след nUnit 3)
- [Test] – маркира метод като тест
- [Category] – организира тестове в категории
- [TestCase] – използва се за провеждане на един и същи тест с неколкократно, но с различни данни
- [SetUp] – кода се изпълнява преди всеки тест
- [OneTimeSetUp] – кода се изпълнява веднъж преди първия тест в класа

nUnit – Assertions

- Оценяват или валидират резултата от тест, основавайки се на върнат резултат, финално състояние на обект или провеждането на събития, наблюдавани повреме на изпълнение
- Един Assert или минава или се проваля
- Ако всички Assert-и преминат и теста минава
- Ако някой Assert се провали, целият тест се проваля
- Синтаксис – Assert.That(test result, constraint instance)
  - Assert.That(sut.Years, Is.EqualTo(1))

nUnit – Assertions примери

- Is.Null
- Is.Not.Null
- Is.Empty
- Is.Not.Empty
- Is.EqualTo(string).IgnoreCase
- Does.StartWith
- Does.EndWith
- Doest.Contain
- …

nUnit – пример за nUnit тест

- Класът и метода му, който ще тестваме:

```
0 references
public class ExtendedCalculator
{
    0 references
    public long Factorial(int n)
    {
        long result = 1;
        for (int i = 1; i <= n; i++)
        {
            result = result * i;
        }

        return result;
    }
}
```

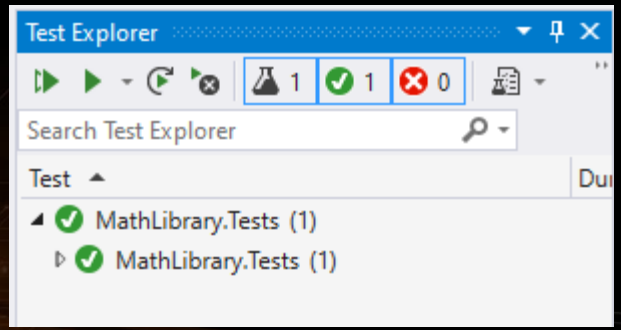
nUnit – пример за nUnit тест

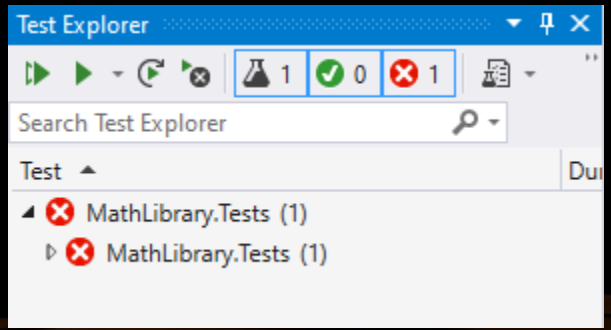
- Тестът ни:

```
0 references
public class ExtendedCalculatorShould
{
    [Test]
    0 references
    public void CalculateCorrectFactorial()
    {
        //Arrange
        var calculator = new ExtendedCalculator();

        //Act
        var result = calculator.Factorial(n: 3);

        //Assert
        Assert.That(result, expression: Is.EqualTo(expected: 6));
    }
}
```

nUnit – пример за nUnit тест

- Резултатът:
  - След Build на проекта би трябвало да виждаме теста си в Test Explorer-а на Visual Studio
  - Ако пуснем теста, бихме видели, че той преминава успешно, ако зададем стойност различна от очакваната, бихме видели как теста се проваля





Регресивно тестване

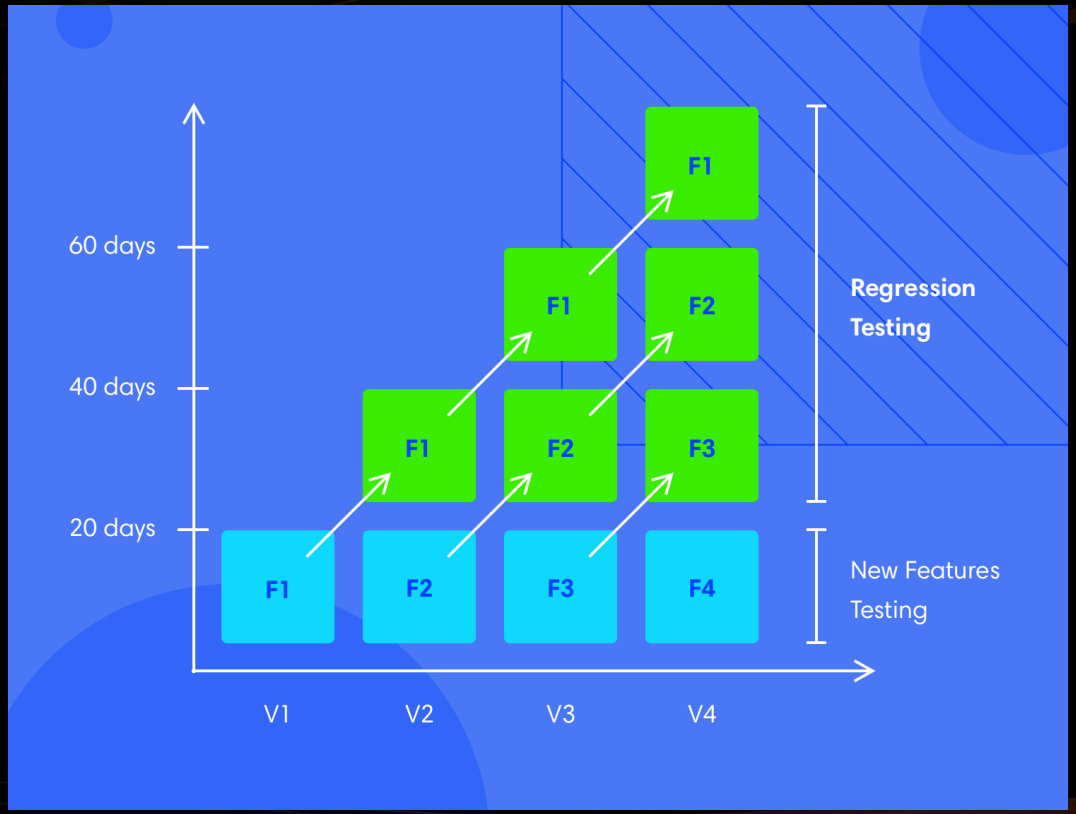
Регресивно тестване

- Тип софтуерно тестване, при който уверява, че при въвеждането на промени не са настъпили дефекти в други компоненти на софтуерния продукт
- Промените в приложението могат да предизивкат непреднамерени странични ефекти
- Не е различно от пълното или частичното селектиране на вече проведени тестове, за да се уверим, че съществуващата функционалност продължава да работи

Регресивно тестване - техники

- Повторно провеждане на всички тестове
- Селективно регресивно тестване – избират се набор от тестове, чиито резултати могат да дадат представа за резултатите от всички тестове; Спестява се време
- Приоритизиране на тестовете – според тяхната критичност, честота на използваната функционалност, която се тества

Регресивно тестване



Mocking

Mocking

- Метод, използван за изолиране поведението на даден обект
- Зависимостите се заменят с обекти, които имитират поведението на реалните такива
- Основно се използва при unit тестване

Mocking: Защо да го използваме?

- Намалява сложността на тестовете
- Подобрява времето за изпълнение на тестовете
  - Бавни алгоритми и външни ресурси
- Полезен при недетерминирани зависимости
  - DateTime.Now
- Поддържа паралална разработка
  - Реалната зависимост все още не е достъпна
  - Зависимостта се изгражда от друг екип, контрактор и т.н.

Mocking: Защо да го използваме?

- Подобрява предсказуемостта на тестовете, както и надежността им

Забележка: Освен като отделни частици (unit-и), е добре софтуера да се тества и като цялостна единица с истинските си зависимости (интеграционно тестване).

Mocking + nUnit

- Инсталация на нужните пакети от NuGet Package Manager
  - Moq
- Създаване на mock обект (използвайки интерфейс):

```
var sampleMockObject = new Mock<ISampleMock>();
```

- Самия обект можем да изпозлваме през свойството Object на новосъздадената Mock инстанция

Mocking + nUnit

- Конфигуриране метод на Mock обект да връща стойност
  - Използва се метода Setup() върху обекта, за да се даде спецификация за метода, след което метода Returns(), за да се зададе очакваната за резултат стойност:

```
var mock = new Mock<IExample>();
mock.Setup(x => x.SomeMethod()).Returns(1);
```

Mocking + nUnit

- Конфигуриране метод на Mock обект да връща стойност при определени параметри

```
var mock = new Mock<IExample>();
mock.Setup(x => x.SomeMethod("Value1")).Returns(1);
```

Mocking + nUnit

- Конфигуриране Mock метод да връщат null като резултат
  - Трябва да се обозначи типа данни, който метода връща

```
var mock = new Mock<IExample>();

mock.Setup(x => x
.MyMethod("Value2", 350))
.Returns<string>(null);
```

Mocking + nUnit

- Конфигуриране свойства на Mock обект
  - Подсказва на Mock обекта да следи промените по дадено свойство:

```
var mock = new Mock<IExample>();
mock.Setup(x => x.Count);
```

  - Настройка за всички свойства:

```
mock.SetupAllProperties();
```

Mocking + nUnit

- Конфигуриране свойство на Mock обект да връща определена стойност:

```
var mock = new Mock<IExample>();
mock.Setup(x => x.Count).Returns(315);
```

Mocking + nUnit

- Конфигуриране свойство на Mock обект при йерархия:

```
var mock = new Mock<IExample>();
mock.Setup(x => x.Result.Value.Count).Returns(681);
```

Moq автоматично ще създаде Mock обекти, за да се справи със зависимостите

Mocking + nUnit

- Проверка за извикване на метод без параметри:

```
var mock = new Mock<IExample>();
mock.Verify(x => x.SomeMethod());
```

За проверка дали метод се извиква определен брой пъти може да се подаде 2-ри аргумент от тип Times -> Times.AtLeast(3)

Mocking + nUnit

- Проверка за извикване на метод с дадени параметри:

```
var mock = new Mock<IExample>();
mock.Verify(x =>
x.SomeMethod(
It.IsAny<string>(),
It.IsAny<int>()));
```

Статичният клас It, който идва от Moq може да бъде използван, за определяне шаблон, на който подадени аргументи трябва да отговарят, вместо да ги подава експлицитно

Mocking + nUnit

- Проверка дали единствено конфигурираните методи са извикани:

```
var mock = new Mock<IExample>();
mock.VerifyNoOtherCalls();
```

Mocking + nUnit

- Проверка за извикване на getter:

```
var mock = new Mock<IExample>();
mock.VerifyGet(x => x.SomeProperty);
```

- Проверка за извикване на setter с дадена стойност:

```
var mock = new Mock<IExample>();
mock.VerifySet(x => x.SomeProperty = It.IsAny<int>());
```

Mocking + nUnit

- Проверка за хвърлена грешка при подаване на определени параметри:

```
var mock = new Mock<IExample>();

mock.Setup(x =>
x.SomeMethod(
It.IsAny<int>()))
.Throws<ArgumentOutOfRangeException>();
```

Интеграционно тестване

Интеграционно тестване

- Процесът по тестване на взаимодействието между две единици или два модула
- Индивидуалните единици се комбинират в по-големи частици – модули
- Цел – да се разкрият грешки, възникващи при интеракцията между интегрираните модули
- Обикновено се извършва след unit тестването

Интеграционно тестване: техники

- Big Bang интеграционно тестване
  - Интегрира всички модули в един
  - Трудно е да се открие кой модул предизвиква грешката, ако такава бъде открита
  - Добър подход за малки системи
  - Отнема повече време, за да се интегрират всички модули

Интеграционно тестване: техники

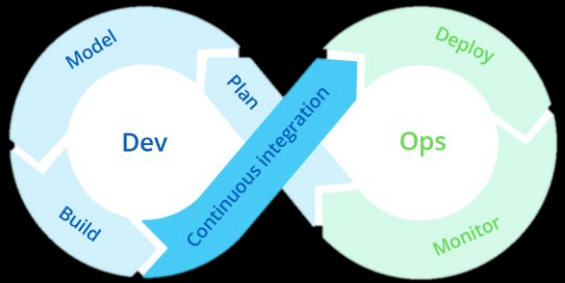
- Bottom-Up интеграционно тестване
  - Всеки модул на по-ниско ниво се тества с модул на по-високо ниво, докато всички модули не бъдат тествани
  - Няколко подсистеми могат да бъдат тествани паралелно
  - Недостатък е сложността, която възниква при система, изградена от много подсистеми

Интеграционно тестване: техники

- Top-Down интеграционно тестване
  - Първо се тестват модулите от високо ниво, последвани от модулите от по-ниско ниво
  - Всеки модул се дебъгва самостоятелно
  - Модулите на по-ниско ниво се тестват недостатъчно

Интеграционно тестване: техники

- Смесено интеграционно тестване
  - Комбинация от Top-Down и Bottom-Up интеграционно тестване
  - Още се нарича сандвично интеграционно тестване
  - Полезно при много големи проекти, изградени от няколко подпроекта
  - Превъзмогва недостатъците на Top-Down и Bottom-Up подходите
  - Не може да се използва за малки системи с голяма зависимост между различните модули



Непрекъсната интеграция

Непрекъсната интеграция

- Практика в софтуерната разработка, при която разработчиците интегрират кода в споделено репозитори често (няколко пъти на ден)
- Всяка интеграция се проверява с автоматизирани build и тестове
- Позволява бързото откриване на грешки и лесното откриване на източника им
- Позволява евнтуален deploy на системата по всяко време

Непрекъсната интеграция

- Подобрява качеството и способността да се тества кода
- Помага да се избегне „интеграционен ад“
- За да се постигне непрекъсната интегация са нужни:
  - Система за управление на версиите: GitHub, GitLab…
  - Инструмент за тестване: JUnit, nUnit, Pytest…
  - CI (Continuous Integration) сървър: Travis CI, Circle CI, Jenkins

Travis CI + GitHub

Travis CI + GitHub

- Услуга за непрекъсната интеграция, използвана да тества и build-ва проекти хостнати в GitHub
- Безплатен за проекти с отворен код
- Лесен за конфигурация
  - .travis.yml файл, който се добавя в директорията на репозиторито
  - Оказват се програмният език, желаните среди за build и тестване и множество други параметри

Travis CI + GitHub: .travis.yml

- Дава специфика за всичко свързано с конфигурацията на Build
- Можем да манипулираме неговите етапи, за да контролираме механизма на действие
- Трябва да се добави в репозиторито предварително

Travis CI + GitHub: .travis.yml

- Примерен файл:

```
language: csharp
solution: solution-name.sln
install:
  - nuget restore solution-name.sln
  - nuget install NUnit.Console -Version 3.9.0 -OutputDirectory
testrunner
script:
  - msbuild /p:Configuration=Release solution-name.sln
  - mono ./testrunner/NUnit.ConsoleRunner.3.9.0/tools/nunit3-
console.exe ./MyProject.Tests/bin/Release/MyProject.Tests.dll
```

Travis CI + GitHub: .travis.yml

- Може да конфигурира различни стъпки
  - before_install / install / after_install
  - before_script / script/ after_script
  - before_success / after_success
  - before_deploy / deploy / after_deploy
- Определя
  - Branch
  - Променливи на средата
  - Известия
  - Docker

Обобщение

- Unit тестване
- nUnit
- Регресивно тестване
- Mocking (подпъхване на функционалност)
- Покритие на кода
- Интеграционно тестване
- Непрекъсната интеграция
- Travis CI

Софтуерно тестване

Въпроси?

Лиценз

- Настоящият курс (слайдове, примери, видео, задачи и др.) се разпространяват под свободен лиценз "Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International"