بنام خدا

تست واحد چیست؟

تست واحد ابزاری است برای مشاهده چگونگی عملکرد یک متد که توسط خود برنامه نویس نوشته میشود. به این صورت که پارامتر های ورودی برای یک متد ساخته شده و ان متد فراخوانی میشود و خروجی متد بسته به حالت مطلوب بررسی میشود. چنانچه خروجی مورد نظر مطلوب باشد تست واحد با موفقیت انجام میشود.

اهمیت انجام تست واحد چیست؟

درستی یک متد مهمترین مسئله برای بررسی است و بارها مشاهده شده که استثناهایی رخ میدهد که توان تولید را به دلیل فرسایش تکراری رخداد میکاهد. نوشتن تست واحد منجر به این میوشد چناچه بعدها تغییری در بیزنس متد ایجاد شود و ورودی و خروجی ها تغییر نکند، صحت این تغییر بیزنس توسط تست بررسی مشود؛ حتی میتوان این تست ها را در build پروژه قرار داد و در ابتدای اجرای یک Solution تمامی تست ها اجرا و درستی بخش به بخش اعضا چک شود.

شروع تست واحد:

یک پروژه ساده داریم برای تعریف حساب های بانکی شامل نام مشتری و مبلغ سپرده و وضعیت و 3 متد واریز به حساب و برداشت از حساب و تغییر وضعیت حساب که به صورت زیر است.

/// <summary>

/// حساب بانکی

/// </summary>

public class Account

{

/// <summary>

/// مشتری

/// </summary>

public string Customer { get; set; }

/// <summary>

/// موجودی حساب

/// </summary>

public float Balance { get; set; }

/// <summary>

/// وضعیت

/// </summary>

public bool Active { get; set; }

public Account(string customer, float balance)

{

Customer = customer;

Balance = balance;

Active = true;

}

/// <summary>

/// افزایش موجودی / واریز به حساب

/// </summary>

/// <param name="amount">مبلغ واریز</param>

public void Credit(float amount)

{

if (!Active)

throw new Exception("این حساب مسدود است.");

if (amount < 0)

throw new ArgumentOutOfRangeException("amount");

Balance += amount;

}

/// <summary>

/// کاهش موجودی / برداشت از حساب

/// </summary>

/// <param name="amount">مبلغ برداشت</param>

public void Debit(float amount)

{

if (!Active)

throw new Exception("این حساب مسدود است.");

if (amount < 0)

throw new ArgumentOutOfRangeException("amount");

if (Balance < amount)

throw new ArgumentOutOfRangeException("amount");

Balance -= amount;

}

/// <summary>

/// انسداد / رفع انسداد

/// </summary>

public void ChangeStateAccount()

{

Active = !Active;

}

}

تابع اصلی نیز به صورت زیر است:

class Program

{

static void Main(string[] args)

{

var account = new Account("Ali",1000);

account.Credit(4000);

account.Debit(2000);

Console.WriteLine("Current balance is ${0}", account.Balance);

Console.ReadKey();

}

}

به Solution یک پروژه از نوع تست واحد اضافه میکنیم.

در این پروژه ابتدا Reference از پروژه ای که مورد تست هست میگیریم. سپس در کلاس تست مربوطه شروع به نوشتن متد برای انواع تست متدهای پروژه اصلی میکنیم.

توجه داشته باشید که Data Annotation های بالای کلاس تست و متدهای تست در تعیین نوع نگاه کامپایلر به این بلوک ها موثر است و باید این مسئله به درستی رعایت شود. همچنین در صورت نیاز میتوان از کلاس StartUp برای شورع تست استفاده کرد که عمدتا برای تعریف آن از نام ClassInit استفاده میشود و در بالای آن از [ClassInitialize] استفاده میشود.

در Library تست واحد میتوان به دو صورت چگونگی صحت عملکرد یک تست را بررسی کرد:1- با استفاده از Assert و با استفاده از ExpectedException که در زیر به هر دو صورت آن میپردازیم.

[TestClass]

public class UnitTest

{

/// <summary>

/// تعریف حساب جدید و بررسی تمامی فرآیند های معمول روی حساب

/// </summary>

[TestMethod]

public void Create\_New\_Account\_And\_Check\_The\_Process()

{

//Arrange

var account = new Account("Hassan", 4000);

var account2 = new Account("Ali", 10000);

//Act

account.Credit(5000);

account2.Debit(3000);

account.ChangeStateAccount();

account2.Active = false;

account2.ChangeStateAccount();

//Assert

Assert.AreEqual(account.Balance,9000);

Assert.AreEqual(account2.Balance,7000);

Assert.IsTrue(account2.Active);

Assert.AreEqual(account.Active,false);

}

همانطور که مشاهده میشود ابتدا در قسمت Arrange خوراک تست آماده میشود. سپس در قسمت Act فعالیت هایی که زیر ذره بین تست هستند صورت میپیرند و سپس در قسمت Assert درستی مقادیر با مقادیر مورد انتظار ما مطابقت داده میشود.

برای بررسی خطاهای تعیین شده هنگام نوشتن یک متد نیز میتوان به صورت زیر عمل کرد:

/// <summary>

/// زمانی که کاربر بخواهد به یک حساب مسدود واریز کند باید جلوی آن گرفته شود.

/// </summary>

[TestMethod]

[ExpectedException(typeof (Exception))]

public void When\_Deactive\_Account\_Wants\_To\_add\_Credit\_Should\_Throw\_Exception()

{

//Arrange

var account = new Account("Hassan", 4000) {Active = false};

//Act

account.Credit(4000);

//Assert

//Assert is handled with ExpectedException

}

[TestMethod]

[ExpectedException(typeof (ArgumentOutOfRangeException))]

public void When\_Customer\_Wants\_To\_Debit\_More\_Than\_Balance\_Should\_Throw\_ArgumentOutOfRangeException()

{

//Arrange

var account = new Account("Hassan", 4000);

//Act

account.Debit(5000);

//Assert

//Assert is handled with ArgumentOutOfRangeException

}

همانطور که مشخص است نام متد تست باید کامل و شفاف به صورتی انتخاب شود که بیانگر رخداد درون متد تست باشد. در این متد ها Assert مورد انتظار با DataAnnotation که پیش از این توضیح داده شد کنترل گردیده است. بدین صورت کارد میکند که وقتی Act انجام میشود متد تست میکند آن Assert رخ بدهد.

استفاده از Library Moq در تست واحد:

ابتدا باید به این توضیح بپردازیم که این کتابخانه چه کاری میکند و چه امکانی برای انجام تست واحد فراهم میکند.

در پروژه های بزرگ و زمانی که ارتباطات بین لایه ای زیادی موجود است و اصول SOLID رعایت میشود. شما در یک لایه برای ارایه فعالیت ها و خدمات متدهایتان با Interface های لایه های دیگر در ارتباط هستید و برای نوشتن تست واحد متدهایتان مشکلی بزرگ دارید که نمیتوانید به این لایه ها دسترسی داشته باشید و ماهیت تست واحد را زیر سوال میبرید. Library Moq این امکان را به شما میدهد که از این Interface ها یک تصویر مجازی بسازید همانند Snap Shot و با آن کار کنید بدون اینکه در لایه های دیگر بروید و ماهیت تست واحد را زیر سوال ببرید.

برای استفاده از متدهایی که در این Interface ها موجود است شما باید یک شی از نوع Mock<> از آنها بسازید و سپس با استفاده از متد Setup به صورت مجازی متد مورد نظر را فراخوانی کنید و مقدار بازگشتی مورد انتظار را با Return معرفی کنید، سپس از آن استفاده کنید.

همچنین برای دسترسی به خود شی از Property با نام Objet از موجودیت mock شده استفاده میکنیم.

برای شناسایی بهتر از اینکه از چه اینترفیس هایی باید Mock<> بسازید میتوانید به متد سازنده کلاسی که معرف لایه ایست که برای آن تست واحد مینویسید، مراجعه کنید.

نحوه اجرای یک تست واحد با استفاده از Moq با توجه به توضیحات بالا به صورت زیر است:

پروژه مورد بررسی لایه Service برای تعریف واحد های سازمانی است که با الگوریتم DDD و CQRS پیاده سازی شده است.

ابتدا به Constractor خود لایه سرویس نگاه میکنیم تا بتوانید شناسایی کنید از چه Interface هایی باید Mock<> کنیم.

public class OrganizationalService : ICommandHandler<CreateUnitTypeCommand>,

ICommandHandler<DeleteUnitTypeCommand>,

{

private readonly IUnitOfWork \_unitOfWork;

private readonly IUnitTypeRepository \_unitTypeRepository;

private readonly IOrganizationUnitRepository \_organizationUnitRepository;

private readonly IOrganizationUnitDomainService \_organizationUnitDomainService;

public OrganizationalService(IUnitOfWork unitOfWork, IUnitTypeRepository unitTypeRepository, IOrganizationUnitRepository organizationUnitRepository, IOrganizationUnitDomainService organizationUnitDomainService)

{

\_unitOfWork = unitOfWork;

\_unitTypeRepository = unitTypeRepository;

\_organizationUnitRepository = organizationUnitRepository;

\_organizationUnitDomainService = organizationUnitDomainService;

}

مشاهده میکنید که 4 Interface استفاده شده و در متد سازنده نیز مقدار دهی شده اند. پس 4 Mock نیاز داریم. در پروژه تست به صورت زیر و در ClassInitialize عمل میکنیم.

[TestClass]

public class OrganizationServiceTest

{

private static OrganizationalService \_organizationalService;

private static Mock<IUnitTypeRepository> \_mockUnitTypeRepository;

private static Mock<IUnitOfWork> \_mockUnitOfWork;

private static Mock<IOrganizationUnitRepository> \_mockOrganizationUnitRepository;

private static Mock<IOrganizationUnitDomainService> \_mockOrganizationUnitDomainService;

[ClassInitialize]

public static void ClassInit(TestContext context)

{

TestBootstrapper.ConfigureDependencies();

\_mockUnitOfWork = new Mock<IUnitOfWork>();

\_mockUnitTypeRepository = new Mock<IUnitTypeRepository>();

\_mockOrganizationUnitRepository = new Mock<IOrganizationUnitRepository>();

\_mockOrganizationUnitDomainService=new Mock<IOrganizationUnitDomainService>();

\_organizationalService = new OrganizationalService(\_mockUnitOfWork.Object, \_mockUnitTypeRepository.Object, \_mockOrganizationUnitRepository.Object,\_mockOrganizationUnitDomainService.Object);

}

از خود لایه سرویس با نام OrganizationService یک آبجک میگیریم و 4 واسط دیگر به صورت Mock شده تعریف میشوند. همچنین در کلاس بارگذار از همان نوع مقدار دهی میگردند تا در اجرای تمامی متدهای تست در دست کامپایلر باشند. همچنین برای new کردن خود سرویس از mock.obect ها که حاوی مقدار اصلی است استفاده مکنیم.

خود متد اصلی به صورت زیر است:

/// <summary>

/// یک نوع واحد سازمانی را حذف مینماید

/// </summary>

/// <param name="command"></param>

public void Handle(DeleteUnitTypeCommand command)

{

var unitType = \_unitTypeRepository.FindBy(command.UnitTypeId);

if (unitType == null)

throw new DeleteEntityNotFoundException();

ICanDeleteUnitTypeSpecification canDeleteUnitType = new CanDeleteUnitTypeSpecification(\_organizationUnitRepository);

if (canDeleteUnitType.IsSatisfiedBy(unitType))

throw new UnitTypeIsUnderUsingException(unitType.Title);

\_unitTypeRepository.Remove(unitType);

}

متد های تست این متد نیز به صورت زیر هستند:

/// <summary>

/// کامند حذف نوع واحد سازمانی باید به درستی حذف کند.

/// </summary>

[TestMethod]

public void DeleteUnitTypeCommand\_Should\_Delete\_UnitType()

{

//Arrange

var unitTypeId=new Guid();

var deleteUnitTypeCommand = new DeleteUnitTypeCommand { UnitTypeId = unitTypeId };

var unitType = new UnitType("خوشه");

var org = new List<OrganizationUnit>();

\_mockUnitTypeRepository.Setup(d => d.FindBy(deleteUnitTypeCommand.UnitTypeId)).Returns(unitType);

\_mockUnitTypeRepository.Setup(x => x.Remove(unitType));

\_mockOrganizationUnitRepository.Setup(z => z.FindBy(unitType)).Returns(org);

try

{

//Act

\_organizationalService.Handle(deleteUnitTypeCommand);

}

catch (Exception ex)

{

//Assert

Assert.Fail(ex.Message);

}

}

همانطور که مشاهده میشود ابتدا یک Guid به عنوان آی دی نوع واحد سازمانی گرفته میشود و همان آی دی برای تعریف کامند حذف به آن ارسال میشود. سپس یک نوع واحد سازمانی دلخواه تستی ساخته میشود و همچنین یک لیست خالی از واحد های سازمانی که برای چک شدن توسط خود متد Handle استفاده شده است ساخته میشود، در اینجا این متد خالی است تا شرط غلط شود و عمل حذف به درستی صورت پذیرد.

برای اعمالی که در Handle انجام میشود و متد هایی که از Interface ها صدا میشود Setup میکنیم و آنهایی را که Return دارند به object هایی که مورد انتظار خودمان هست نسبت میدهیم.

در Setup اول میگوییم که ان Guid مربوط به "خوشه" است. در Setup بعدی برای عمل Remove کدی مینویسیم که چون عمل حذف Return ندارد میتواند این خط به کل حذف شود! به طور کلی Setup هایی که Return ندارند میتوانند حذف شوند.

در Setup بعدی از Interface دیگر متد FindBy که قرار است چک کند این نوع واحد سازمانی برای تعریف واحد سازمانی استفاده شده است، در Return به آن یک لیست خالی اختصاص میدهیم که نشان دهیم لیست خالی برگشته است.

عملیات Act را وارد Try میکنیم تا اگر به هر دلیل انجام نشد، Assert ما باشد.

دو حالت رخداد استثنا که در متد اصلی تست شده است در دو متد تست به طور جداگانه تست گردیده است:

/// <summary>

/// کامند حذف یک نوع واحد سازمانی باید پیش از حذف بررسی کند که این شناسه داده شده برای حذف موجود باشد.

/// </summary>

[TestMethod]

[ExpectedException(typeof(DeleteEntityNotFoundException))]

public void DeleteUnitTypeCommand\_ShouldNot\_Delete\_When\_UnitTypeId\_NotExist()

{

//Arrange

var unitTypeId = new Guid();

var deleteUnitTypeCommand = new DeleteUnitTypeCommand();

var unitType = new UnitType("خوشه");

var org = new List<OrganizationUnit>();

\_mockUnitTypeRepository.Setup(d => d.FindBy(unitTypeId)).Returns(unitType);

\_mockUnitTypeRepository.Setup(x => x.Remove(unitType));

\_mockOrganizationUnitRepository.Setup(z => z.FindBy(unitType)).Returns(org);

//Act

\_organizationalService.Handle(deleteUnitTypeCommand);

}

/// <summary>

/// کامند حذف یک نوع واحد سازمانی نباید اجرا شود وقتی که نوع واحد برای تعریف واحد های سازمان استفاده شده است.

/// </summary>

[TestMethod]

[ExpectedException(typeof(UnitTypeIsUnderUsingException))]

public void DeleteUnitTypeCommand\_ShouldNot\_Delete\_When\_UnitType\_Exist\_but\_UsedForDefineOrganizationUnit()

{

//Arrange

var unitTypeId = new Guid();

var deleteUnitTypeCommand = new DeleteUnitTypeCommand { UnitTypeId = unitTypeId };

var unitType = new UnitType("خوشه");

var org = new List<OrganizationUnit>()

{

new OrganizationUnit("مدیریت یک", unitType, null),

new OrganizationUnit("مدیریت دو", unitType, null)

};

\_mockUnitTypeRepository.Setup(d => d.FindBy(deleteUnitTypeCommand.UnitTypeId)).Returns(unitType);

\_mockUnitTypeRepository.Setup(x => x.Remove(unitType));

\_mockOrganizationUnitRepository.Setup(z => z.FindBy(unitType)).Returns(org);

//Act

\_organizationalService.Handle(deleteUnitTypeCommand);

}

متد DeleteUnitTypeCommand\_ShouldNot\_Delete\_When\_UnitTypeId\_NotExist همانطور که از نامش معلوم است چک میکند که نوع واحد سازمانی که ID آن برای حذف ارسال میشود در DataBase وجود دارد و اگر نباشد Exception مطلوب ما باید داده شود.  
در متد DeleteUnitTypeCommand\_ShouldNot\_Delete\_When\_UnitType\_Exist\_but\_UsedForDefineOrganizationUnit بررسی میشود که از این نوع واحد سازمانی برای تعریف واحد سازمانی استفاده شده است یا نه و صحت این مورد با الگوی Specification صورت گرفته است. استثنای مطلوب ما Assert و شرط درستی این متد تست میباشد.

تهیه و تنظیم  
آرش کریمی