Understanding
Unit Testing

INTRODUCTION TO AUTOMATED TESTING AND UNIT TESTING

Introduction

- Ikhwan Hayat (a.k.a 1kHz)
- 9 years experience developing software
- Freelance Software Developer
- ikhwanhayat@gmail.com

Testing?

acceptance test
integration test
unit test

manual
vs
automated

white box
vs
black box

Unit?

Unit testing is a method by which individual units of source code are tested to determine if they are fit for use.

One can view a unit as the **smallest testable part of an application.**

Unit tests are **created by programmers** or occasionally by white box testers during the development process.

WIKIPEDIA
The Free Encyclopedia

Demo: Bank Account

- **Requirements**
  - Can create bank accounts
  - Can deposit money into account
  - Can withdraw money from account
    - Throw exception if balance is insufficient
  - Can transfer money from one account to another

TDD

- Test Driven Design
- It's not testing, but using tests to **DRIVE** the design
- As a *side-effect*, you got unit tests!
  With good level of coverage!

RED

Write a failing test. With empty class/method.

GREEN

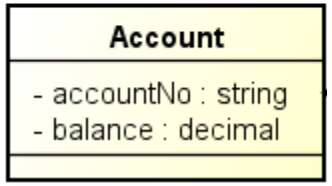
Fill in the class/method implementation. Make the tests pass.

REFACTOR

Make code better.

ARRANGE
ACT
ASSERT

Design



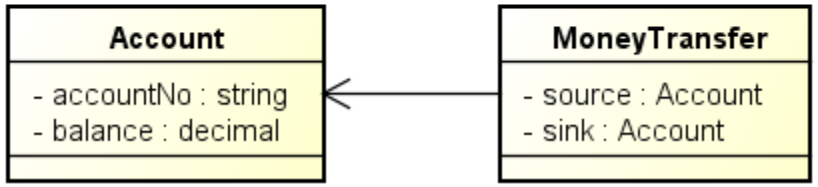
Unit

```
public class Account
{
    public string AccountNo { get; set; }
    public decimal Balance { get; set; }

    public Account(string accountNo, decimal initialBalance)
    {
        this.AccountNo = accountNo;
        this.Balance = initialBalance;
    }

    public void Deposit(decimal amount)
    {
        this.Balance += amount;
    }

    public void Withdraw(decimal amount)
    {
        if (this.Balance < amount)
            throw new InsufficientFundsException();

        this.Balance -= amount;
    }
}
```

Test

```
[TestFixture]
public class AccountTest
{
    [SetUp]
    public void SetUp()
    {
        account = new Account("001", 100.0m);
    }

    Account account = null;

    [Test]
    public void CreateAccount()
    {
        // Arrange, Act
        var account = new Account("001", 100.0m);

        // Assert
        Assert.That(account.AccountNo, Is.EqualTo("001"));
        Assert.That(account.Balance, Is.EqualTo(100.0m));
    }

    [Test]
    public void DepositMoney()
    {
        // Arrange in SetUp

        // Act
        account.Deposit(10.0m);

        // Assert
        Assert.That(account.Balance, Is.EqualTo(110.0m));
    }

    [Test]
    public void WithdrawMoney()
    {
        // Arrange in SetUp

        // Act
        account.Withdraw(10.0m);

        // Assert
        Assert.That(account.Balance, Is.EqualTo(90.0m));
    }

    [Test]
    public void WithdrawMoreThanBalance()
    {
        var ex = Assert.Throws<InsufficientFundsException>(() => account.Withdraw(110.0m));
    }
}
```

Design 2


Account
- accountNo : string
- balance : decimal
MoneyTransfer
- source : Account
- sink : Account

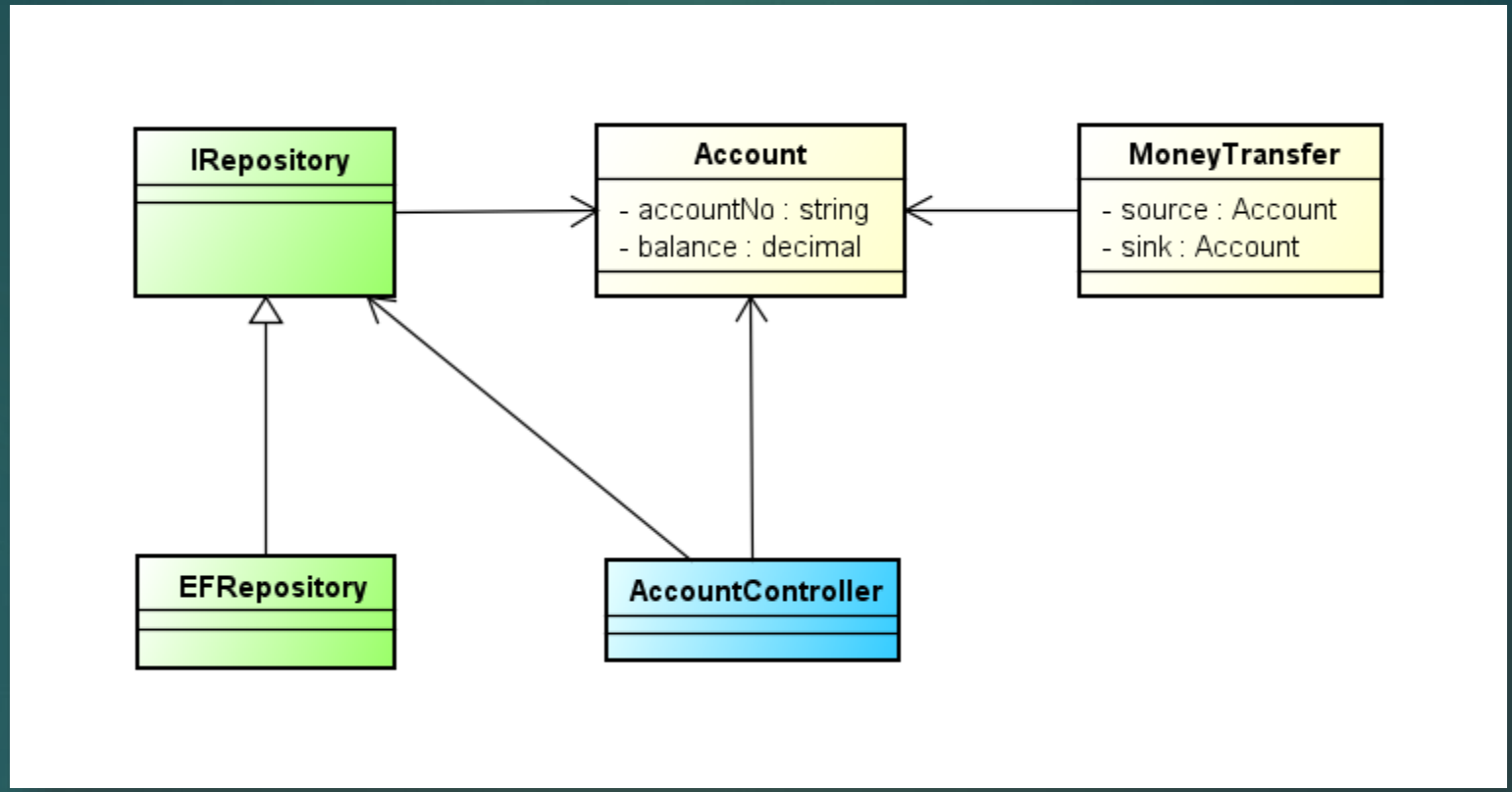
Unit

```
public class MoneyTransfer
{
    public MoneyTransfer(Account source, Account sink, decimal amount)
    {
        this.Source = source;
        this.Sink = sink;

        // Do transfer
        this.Source.Withdraw(amount);
        this.Sink.Deposit(amount);

        this.Amount = amount;
        this.Date = DateTime.Now;
    }

    public virtual Account Source { get; set; }
    public virtual Account Sink { get; set; }
    public virtual decimal Amount { get; set; }
    public virtual DateTime Date { get; set; }

}
```

Test

```
[TestFixture]
class MoneyTransferTest
{
    [Test]
    public void TransferMoney()
    {
        // Arrange
        var mockAcc1 = new Mock<Account>();
        mockAcc1.SetupGet(x => x.Balance).Returns(100.0m);
        var mockAcc2 = new Mock<Account>();

        // Act
        var moneyTransfer = new MoneyTransfer(mockAcc1.Object, mockAcc2.Object, 10.0m);

        // Assert
        mockAcc1.Verify(x => x.Withdraw(10.0m), Times.Once());
        mockAcc2.Verify(x => x.Deposit(10.0m), Times.Once());

        Assert.That(moneyTransfer.Amount, Is.EqualTo(10.0m));
        Assert.That(moneyTransfer.Date.ToShortDateString(),
                    Is.EqualTo(DateTime.Now.ToShortDateString()));
    }
}
```

Design 3


IRepository
Account
- accountNo : string
- balance : decimal
MoneyTransfer
- source : Account
- sink : Account
EFRepository
AccountController

Unit

```
public interface IRepository
{
    T Get<T>(object id) where T : class;
    IQueryable<T> Query<T>() where T : class;
    void Create(object acc);
    void Update(object acc);
}
```

```
public class EFRepository : IRepository, IDisposable
{
    BankDataContext db = null;

    public EFRepository(string connectionString)
    {
        db = new BankDataContext(connectionString);
    }

    public T Get<T>(object id) where T: class...

    public IQueryable<T> Query<T>() where T : class...

    public void Create(object entity)...

    public void Update(object entity)...

    public void Dispose()...
}
```

Test


```
[TestFixture]
class EFRepositoryTest
{
    EFRepository repo = null;

    [SetUp]
    public void SetUp()
    {
        var filePath = @"bank_test.sdf";
        if (File.Exists(filePath)) File.Delete(filePath);
        string connectionString = "Data Source = " + filePath;
        var connFac = new SqlCeConnectionFactory("System.Data.SqlServerCe.4.0");
        Database.DefaultConnectionFactory = connFac;
        repo = new EFRepository(connectionString);
    }

    [TearDown]
    public void TearDown()
    {
        repo.Dispose();
    }

    [Test]
    public void CreateAndRetrieveAccount()
    {
        var acc = new Account("001", 100.0m);
        acc.Id = Guid.NewGuid();

        repo.Create(acc);

        var results = repo.Query<Account>();

        Assert.That(results.Count(), Is.EqualTo(1));
        Assert.That(results.First().AccountNo, Is.EqualTo("001"));
    }
}
```

A good unit test is…

- **Isolated/independent**
  - Test one thing at a time.
  - Unit under test doesn't depend on the other to make test runs.
- **Repeatable**
  - Running multiple times yields the same result.
  - Doesn't rely on environment.
- **Fast**
  - You want to repeat it again and again.
  - You want it to be a pleasure to work with.
- **Self-Documenting**
  - Test code and code under test clear and concise.
  - Can be a reference for usage of your class/method/etc.

Dependency Injection

Single Responsibility Principal

Separation of Interface and Implementation

Web Controller

```
public class AccountsController : Controller
{
    IRepository repo;

    public AccountsController(IRepository repo)
    {
        this.repo = repo;
    }

    public ActionResult Index()
    {
        ViewBag.Accounts = repo.Query<Account>().ToList();
        return View();
    }

    [HttpPost]
    public ActionResult Transfer(Guid sourceId, Guid sinkId, decimal amount)
    {
        var source = repo.Get<Account>(sourceId);
        var sink = repo.Get<Account>(sinkId);

        var moneyTransfer = new MoneyTransfer(source, sink, amount);

        repo.Update(source);
        repo.Update(sink);

        TempData["Message"] = "Transfer successful.";
        return RedirectToAction("Index");
    }
}
```

DI/IoC Setup

```
public class MvcApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        AreaRegistration.RegisterAllAreas();

        WebApiConfig.Register(GlobalConfiguration.Configuration);
        FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
        RouteConfig.RegisterRoutes(RouteTable.Routes);
        BundleConfig.RegisterBundles(BundleTable.Bundles);


        //
        // IoC setup
        //

        var builder = new ContainerBuilder();

        var connStr = ConfigurationManager.ConnectionStrings["conn1"].ConnectionString;
        builder.RegisterType<EFRepository>()
            .As<IRepository>()
            .WithParameter(new NamedParameter("connectionString", connStr))
            .InstancePerHttpRequest();

        builder.RegisterControllers(typeof(MvcApplication).Assembly);

        var container = builder.Build();

        // Tell ASP.NET MVC to use this resolver
        DependencyResolver.SetResolver(new AutofacDependencyResolver(container));
    }
}
```

“Mocking” allow us to isolate dependent units.

Types of Test Doubles

- **Dummy objects** are passed around but never actually used. Usually they are just used to fill parameter lists.
- **Fake objects** actually have working implementations, but usually take some shortcut which makes them not suitable for production.
- **Stub objects** provide canned answers to calls made during the test, usually not responding at all to anything outside what's programmed in for the test.
- **Mock objects** are pre-programmed with expectations which form a specification of the calls they are expected to receive.

Unit

```
public class AccountsController : Controller
{
    IRepository repo;

    public AccountsController(IRepository repo)...

    public ActionResult Index()...

    public ActionResult Create()...

    [HttpPost]
    public ActionResult Create(Account account)...

    public ActionResult Transfer(Guid id)...

    [HttpPost]
    public ActionResult Transfer(Guid sourceId, Guid sinkId, decimal amount)
    {
        var source = repo.Get<Account>(sourceId);
        var sink = repo.Get<Account>(sinkId);

        var moneyTransfer = new MoneyTransfer(source, sink);
        moneyTransfer.Transfer(amount);

        repo.Update(source);
        repo.Update(sink);

        TempData["Message"] = "Transfer successful.";
        return RedirectToAction("Index");
    }
}
```

Test

```
[TestFixture]
class AccountsControllerTest
{
    [Test]
    public void TestTransfer()
    {
        // Arrange
        var acc1 = new Account("001", 100);
        acc1.Id = Guid.NewGuid();

        var acc2 = new Account("002", 100);
        acc2.Id = Guid.NewGuid();

        var mockRepo = new Mock<IRepository>();
        mockRepo.Setup(x => x.Get<Account>(acc1.Id)).Returns(acc1);
        mockRepo.Setup(x => x.Get<Account>(acc2.Id)).Returns(acc2);

        // Act
        var ctlr = new AccountsController(mockRepo.Object);
        ctlr.Transfer(acc1.Id, acc2.Id, 10);

        // Assert
        Assert.That(ctlr.TempData["Message"], Is.EqualTo("Transfer successful."));

        Assert.That(acc1.Balance, Is.EqualTo(90.0m));
        Assert.That(acc2.Balance, Is.EqualTo(110.0m));

        mockRepo.Verify(x => x.Update(acc1), Times.Once());
        mockRepo.Verify(x => x.Update(acc2), Times.Once());
    }
}
```

Benefits

- **Instant feedback**
  - Write test, write code, see instant result.
- **Promote modularity in your design**
  - DI, SRP, Interface vs Implementation.
- **Safety net**
  - Change/add code and check if business rules are still honored.
- **Free documentation**
  - Can be a reference for usage of your class/method/etc.

Done!

We have learned…

- What is unit testing.
- TDD.
- How to write good unit tests.
- How unit testing can benefit us.

Get the codes and slides at…

https://github.com/ikhwanhayat/jomweb_unittest

Next?

You can go on with…

- Learning BDD (Behavior Driven Design).
- Continuous Integration.
- Research on how to design testable systems.
- Use unit testing for your project!
  (I mean, seriously, USE IT!)

MOAR!

- Google+ **MyDev**
- http://www.mydev.my/automated-testing-dalam-pembangunan-perisian.html
- ikhwanhayat@gmail.com

THANK YOU FOR LISTENING!