Unit Test的好处

1. 加深对类和方法的功能需求理解
2. 提高代码质量，在写Unit Test过程中就能发现很多bug
3. 提高产品质量
4. 提高可维护性，为以后的重构打下坚实的基础

Unit test的工作单元是否应该尽可能小？

     -> 原文：如果你创建的工作单元更大，而且它的最终结果对这个接口的用户可见度更高，那么你的测试会更容易维护，如果你试图把工作单元缩到最小，最后会不得不伪造一堆东西，这些东西并不是使用公共API的真实结果，而是生成结果过程中的中间状态

     -> 我们的Engine的Unit Test是不是在犯同样的错误？ -> 是不是应该只测试public的方法？-> 那么不是就是Data Automation类似的东西

     -> 感觉还是要看产品的复杂程度 -> 过于复杂的东西很难只对接口测试就覆盖所有的case？-> 重构代码，让每个模块可以用TDD

优秀单元测试应该具备的特性

1. 能够使用自动化重复的运行
2. 容易实现（？）
3. 第二天还有意义
4. 任何人都能一键运行
5. 速度快
6. 结果稳定
7. 完全隔离的->有各种真实依赖物
8. 能完全控制被测试的单元
9. 失败消息清晰易懂，容易定位何处出错

集成测试和单元测试的区别，可以问自己下面问题

1. 两周前写的单元测试，今天还能运行并且得到结果么？一年前的呢？
2. 我一年前写的单元测试，现在团队内的任何人都能一键运行并且得到结果么？
3. 能一键运行所有单元测试，并且在几分钟（取决于单元测试有多少，最多不应该超过10分钟）内得到清晰的结果么？
4. 能在几分钟内写出一个基本的单元测试么？

优秀单元测试：Fast, Stable, Isolated, Readable, Maintainable

TDD需要的三个技能：知道如何编写优秀的测试，在编码前编写测试以及良好的测试设计 -> 没有这三个技能没法成功实践TDD

Unit Test的命名：[UnitOfWorkName]\_[ScenarioUnderTest]\_[ExpectBehavior]

Unit Test应该包含的三个行为: Arrange，Act，Assert (3A)

外部依赖项：被测试代码与这个对象发生了交互，但你**不能控制**这个对象

     -> 在编码时，可以使用存根来回避外部依赖项问题

     -> 存根：对系统中存在的一个依赖项的**可控制**的替代物 -> Mock Service之类的 Virtual方法

没有什么面向对象的问题是不能用添加一个间接层来解决的

     -> 单元测试的艺术的很大程度上就在于找到合适的地方添加或者使用一个间接层

     -> 无法测试的代码？

          -> 添加一个间接层封装这段代码的调用，然后在测试中mock掉这个间接层

          -> 或者使这段代码变得可以替换，这使得这段不能测试的代码变成了一个间接层

破除依赖的模式：

1. 找到被测试的工作单元使用的接口（这里的接口指与被测试代码一起工作的方法和类）
2. 如果这个接口直接和你的测试单元相连，在代码中添加一个间接层，隐藏这个接口
3. 把这个交互接口的底层实现替换成你可以控制的代码 -> 存根

代码重构：在不改变代码功能的前提下，对代码进行修改 -> 最好有一定的集成测试保护 automation testing

接缝：代码中可以插入不同功能的地方

1. 使用存根类
2. 添加一个构造函数参数
3. 添加一个可设置的公共属性
4. 把一个方法改成可以重写的虚方法
5. 把一个委托 外部化 成一个参数或者属性，可以从外部赋值

破除依赖的时候，可以使用的重构方法：

1. 把具体类抽象成接口
2. 重构代码，从而能够对其注入伪实现
   1. 在被测试类注入一个存根实现
   2. 在被测试类的构造函数层次注入一个伪对象
   3. 注入一个作为属性设置和读取的伪对象
   4. 在一个方法调用前注入一个伪对象 -> 不同之处是发起存根请求的对象是被测试代码，前面的几个都是测试开始前，由被测试代码之外的代码设置的

测试的几种类型

1. 基于值的测试 -> 测试被测试对象的返回值
2. 基于状态的测试 -> 测试改变被测试系统的状态，验证其可见的行为变化
3. 交互测试（最后才会考虑的类型，比较少用） -> 对一个对象如何向其他对象发送消息（ 调用方法 ）的测试 -> 测试一个对象采取的特定行为

模拟对象 -> 系统中的伪对象，验证被测试对象是否正确调用了这个伪对象

模拟对象和存根的最根本的区别是：存根不会导致测试失败，但是模拟对象会（换句话说，我们会对模拟对象进行断言，但是不会对存根断言）

如果一个测试只测试一件事情，测试中最多只有一个模拟对象，其他的伪对象都是存根。

过度指定：过多的指定被测试代码中应该发生的事情的行为，而这些事情实际上对测试无关紧要。例如，测试代码是否调用了存根

动态伪对象：在**运行时**创建的任何存根或者模拟对象，实际上在测试运行时，隔离框架会在**内存中**生成所需要的类的定义等。

受限框架：不能伪造static方法，非virtual方法和非public方法，它们在运行时生成和编译代码（通常是生成继承和重写接口或者基类的代码），因此受限于编译器和IL

不受限框架：基于探查器（profiler-based），它们使用一套叫做profiling API的非托管API。对于CLR的运行实例进行了封装。

     -> 对于CLR代码执行的任何事情，profiling API都提供了相应的事件，其中某些事件可以运行你修改和注入基于IL的代码。

测试层次和组织

在自动化构建过程中加入自动化测试

     构建过程是一个逻辑概念，包括：构建脚本，构建集成服务器，构建触发器以及团队对代码部署集成方式的了解和认同

一般会有好几个构建脚本，每个完成一个功能。

1. 持续集成（CI）脚本->构建test版本，运行所有单元测试，目的是为了最短的时间内得到最多的信息
2. 每日构建脚本->构建accp版本，运行所有时间长的测试，以及部署测试环境供第二天使用
3. 部署构建脚本->交付机制

项目结构要考虑到可以用命令行运行整个测试程序集-> 单元测试和集成测试放在不同的项目

将测试映射到被测试代码

* 将测试映射到项目 MyLibrary ->  MyLibrary.UnitTests
* 将测试映射到类    MyClass -> MyClass\_UnitTests

注入横切关注点

     如果需要处理类似于时间, 日志等横切关注点的时候，使用依赖注入并不是一个好主意

     使用一个定制类SystemTime，而不是DateTime -> 是的代码维护比较难，考虑的东西比较多

基于测试类继承的三种模式

* 抽象测试基础结构类
  + 创建一个抽象的测试类，包含必须的通用基础结构，比如在BaseTestClass中可以做一些通用mock方法，但是会降低可读性，需要在可读性和DRY之间做取舍
* 测试类模板
  + 如果有些类都属于某种特定的类型，想确保编写测试代码的人不要忘记包含某组测试。那么我们可以抽取出一个抽象测试类，对应的派生测试类必须实现抽象测试类中的接口
* 抽象测试驱动类

把你的API介绍给开发人员

1. 让团队的两个成员结对编写测试，至少偶尔一次
2. 准备一个小文档或者一个速查表，列出已有的API和存放位置 -> 可以用自动化的形式进行更新 -> 所有的辅助类都使用一套已知的前后缀
3. 团队会议上讨论API的变更
4. 新人入职培训的时候过一遍文档
5. 进行测试审查，确保在正确的地方使用了正确的API，而不是再写一个

单元测试失败的情况

1. 产品缺陷 -> 修改了代码，导致已有的测试失败 -> 修复产品代码，使单元测试成功
2. 测试缺陷 -> 测试中有缺陷 -> 修复测试 -> 确保测试在该成功的时候成功，该失败的时候失败, 千万不能是测试了错误的内容导致成功
3. 语义或者API变更 -> 被测试对象的功能虽然没有变化，但是使用方式发生了变化 -> 多了一个参数？
4. 冲突或者无效的测试 -> 删掉，不用注释

避免测试中的逻辑 -> 单元测试中不应该有if else之类的逻辑控制代码

可维护的测试：

1. 只测试公共契约
   * 测试私有方法：就算测试通过也不能说明其他地方正确的使用了这个方法
   * 如果一个私有方法值得测试，那么应该将其设为public或者放到另外一个类里面会比较干净
2. 删除重复测试
3. 实施测试隔离

过度指定的行为：

1. 对被测试对象的纯内部状态进行断言 -> 应该测试对象的公共契约和公共功能
2. 测试在需要存根的时候使用了模拟对象  -> 测试不应该关心什么内部方法调用或者有没有发生调用
3. 测试中指定顺序或者使用了精确匹配

可读性

1. 单元测试命名
   1. 被测试的方法名
   2. 测试场景
   3. 预期行为
2. 变量命名
3. 好的断言信息
4. 断言和操作分离

在现有组织中引入单元测试

* 逐步成为变革的倡导者
  + 准备好面对质疑 -> 做足功课
  + 说服组织内部成员
    - 找到支持者 -> 联系他们，并且征求意见，接着确保他们成为这个过程的**参与者**
    - 找出反对者 -> 给他们分配一些积极的角色，让他们成为参与者，请他们在过程中提供帮助，人们不喜欢别人告诉他们什么事情做得不好
  + 找到切入点
    - 选择较小的试点团队 -> 开发低风险项目 -> 容易说服人们进行尝试
    - 创建子团队 ->  创建一个子团队来进行黑洞组件的单元测试覆盖
    - 考虑项目可行性   ->  避免将目标定的太大
    - 使用代码和测试审查作为培训工具  ->  面对面的进行审查，初期审查每一行代码
* 成功之道
  + 游击
  + 说服高层
  + 引入外援 -> 言论自由，经验, 专有时间
  + 使进度可见
    - 在CI中集成
  + 应对障碍 -> 遇到挫折不要有失败感
* 失败原因
  + 缺少驱动力，要持续的推进变革很难，必须在日常工作之外抽出时间培训别人
  + 缺乏高层支持
  + 不好的实现和第一印象 -> 如果再开始阶段效果不佳，人们就会失去信心
  + 缺少团队支持，应该努力让团队成为新过程的参与者
* 质疑和回答
  + 单元测试会给现有流程增加多少时间？
    - 整体代码质量的提高可以提高生产效率，缩短项目时间
    - 单元测试会使实现特定功能的时间增加，但是产品整体的发布时间会提前
  + 单元测试是否会抢了QA的饭碗
    - 不会，QA可以从低级的错误中解放出来，来进行更有价值的逻辑性测试和构建自动化e2e测试
  + 证明单元测试行之有效的方法
    - 创建某种度量指标
  + 有大量的没有测试的代码，应该从哪里开始？
    - 通常20%的代码包含了90%的缺陷，问问团队哪个组件问题最多，那么就从这个组件开始
  + 我的代码已经调试通过了，还需要测试的原因
    - 代码在大部分时间都处于维护状态，需要单元测试来保证不会被破坏

从哪里开始增加单元测试

* 影响组件优先级的因素
  + 逻辑复杂度 -> 组件中逻辑的数量
  + 依赖数       -> 组件中依赖项的数目
  + 优先级       -> 在项目中的优先级
* 复杂度高 + 容易测试 VS  复杂度高 + 难以测试，选哪个
  + 对于新引入单元测试，大家熟练度都不高的，从容易的开始 -> 会增加后期交付的速度
  + 对于有经验的团队，才能用先难后易的策略，但是编写测试的速度会迅速提高

重构前编写集成测试

* 给现有系统添加一个或者多个集成测试