Лекция 14 (3. Януари, 2012)

(Лекцията беше представена от Васил Люнчев)

Unit Testing

В този текст ще бъдат разгледани последователно въпросите:

- Защо е хубаво да **тестваме**?
- Защо да тестваме на **Unit-и**, и кое всъщност е Unit?
- Кои са **най-простите** начини за тестване?
- Кое е **по-доброто** на Unit testing-а?
- **Как се пише** Unit test?
- Unit Testing **Framework**
- **Code Coverage**
- **Свързване с SVN**
- Unit tests като **жива документация**

Защо е хубаво да тестваме?

Защото иначе няма как да знаем дали нещо (проект, или част от проект) работи! Вариантът „работи, вярвайте ми“ не върви в комерсиалното програмиране. Програмата просто трябва да е била стартирана поне веднъж, за да можем да кажем, че тя работи. Същото се отнася и за по-малките парчета от програмата – всяка част от кода трябва да е била изпълнена поне веднъж, за да можем да кажем че тази част работи.

Защо да тестваме на Unit-и, и кое всъщност е Unit?

Unit е най-малката част от приложение, която е достатъчно голяма да има собствен смисъл и да може да бъде тествана. При процедурното програмиране Unit би могло да бъде цял модул от програмата, но най-често е самостоятелна процедура. При обектно-ориентираните езици най-често Unit е функция или метод на клас, като в някои по-рядки случаи Unit е цял клас или интерфейс. Тестваме кода на части (на отделни Unit-и), защото така е по-лесно да проверим дали **всяка** част от кода работи.

Ако тестовете ни се изпълняват върху цялия код, може да е много трудно да накараме програмата да изпълни дадена нейна част. За целта би се наложило да направим сложни (и в общия случай големи) тестове, които да са design-нати по такъв начин, че да trigger-ват точно определена част от кода. Доста по-лесно би било да имаме повече на брой, но много по-прости тестове, които да можем да пускаме само върху частта от кода, която бихме желали да тестваме.

Например, нека програмата ни е „за всеки човек с повече от 300 приятели намери броя friends of friends". За да проверим дали частта „намери броя

friends of friends“ работи, първо трябва да направим човек с над 300 приятели. Ако тестваме на части, можем да пуснем функцията “friendsOfFriends” на когото си поискаме и така ще видим дали тя работи.

Също така, ако пускаме тестове върху цялата програма, и тя даде грешен резултат, е много по-трудно да разберем в коя част от кода е проблемът. Нека в предишния пример сме пуснали програмата за човек с 400 приятели и тя не върне нищо. Къде е проблемът – в проверката дали човека има повече от 300 приятели, или във функцията friendsOfFriends, или някъде извън всичко това?

Улеснението, при тестването на части, идва от факта, че разглеждаме частите **независимо** една от друга. Няма нужда да се грижим за всички части, грижим се само за частта, която тестваме в момента.

Кои са най-простите начини за тестване?
Искаме да тестваме един клас. Най-просто (т.е. най-бързо и лесно за правене) е да изпечатаме на стандартния изход резултата от публичните му функции. Можем например да ги стартираме от функцията main( ). Ако тези функции връщат верен резултат, то можем да заключим, че те работят правилно.

Тук сигурно някой ще оспори веднага: „това че функциите връщат верен резултат в даден конкретен случай не значи че те работят вярно във всички случаи“. Да, така е. И нека стане ясно още сега – това че кодът връща верни резултати (респективно, това че Unit Test-овете минават без грешка) **не значи че програмата е вярна!** Значи само че нашите тестове не са намерили грешка.

Това е абсолютно аналогично с компилирането. Ако програмата ни се компилира, това не значи че тя работи вярно. Значи само че компилаторът не е успял да намери грешка.

Погледнато математически, за да е вярна една програма **необходимо** условие е да се компилира и тестовете да минат. Това обаче **не е достатъчно** условие.

Подходът с „печатане от main( )“ не е лош. Дори е препоръчителен за много малки програми (да кажем под 200 реда код). Предимствата му са, че се пише бързо и не ви е нужно нищо специално за него. Недостатъците му са, че имаме само един main( ) метод, който освен да тества, си има и друга работа. А и когато класовете станат много (повече от 2) кодът за тяхното тестване също става доста. Следователно става все по-трудно да тестваме **повторно** даден клас.

Други начини за тестване са „печатане от тялото на тестваната функция“ и „debugging”. И двата се ползват по-скоро за установяване къде е грешката, а не толкова за да се види дали кодът работи. Тези два начина са доста различни от Unit Testing.

Unit testing – кое му е по-доброто?
Предимствата на Unit testing пред простите методи за тестване са:

- Тестващият код стои далеч от кода на програмата (тествания код).
- Веднъж написан, тестът може много лесно да се пуска отново. Буквално „натискане на един бутон".
- Лесно е да се пуснат всички тестове наведнъж. Буквално „натискане на един бутон" -- така за секунди се тестват всички части на програмата една по една.
- Т.е., ако направим някаква промяна в кода, и искаме да видим дали не сме развалили нещо, можем просто да „натиснем един бутон" и да проверим дали всички тестове минават.

Недостатъци:

- За да се ползва Unit Testing, първо трябва да се инсталира Unit Testing Framework и да се свърже с IDE-то, на което се пише кода.
- Написването на теста отнема малко повече време отколкото просто изпечатване на резултат в main( ).

Как се пише Unit test?
Първо е хубаво да кажем, че всеки unit test за различен unit се пише отделно. Всъщност един Unit Test също представлява клас. Т.е. имаме клас с изпълним код който се казва Math. Искаме да го тестваме с Unit Test. Това значи че трябва да напишем клас MathTest, който вика методи от клас Photo и да проверява дали резултатите им са коректни.

В повечето от Unit Testing Framework-овете създаването на „скелета" на класа MathTest става с „натискането на един бутон". След това имаме клас и по един метод за всеки от методите във Photo (това бе показано на лекцията).

На практика, писането на Unit Test се свежда до попълването на методите на класа MathTest. В тези методи се пише код, който е много подобен на „изпечатване на конзолата от main( )" с 2 малки разлики:

- На конзолата от main( ) печатаме **само резултата** от тествания код, и го проверяваме дали е верен като го сравняваме с нещо което сме сметнали на ум (или на лист)
  В Unit test трябва да запишем и **резултата от тествания код**, и **очаквания резултат**
- В main( ) ползваме функции за печатане на конзолата (cout, printf, console.writeLine, system.out.println, echo), докато в Unit test ползваме **assert**. Пример:

```
assertEquals(Math.sum(5, 8), 13);
assertEquals(Math.minus(99, 100), -1);
assertTrue(Math.isGreaterThan(100, 2));
```

Примерите горе правят следното:

- Пусни функцията „сума“ с аргументи 5 и 8 (това е функцията която тестваме дали работи). Сега провери дали резултатът е 13.
- Пусни функцията „изваждане“ с аргументи 99 и 100 (тестваме функцията изваждане), след това провери дали резултатът е -1.
- Пусни функцията „isGreaterThan" с аргументи 100 и 2. Провери дали резултатът е „истина“.

Накратко – Unit Test се пише, като се прави по един TestClass за всеки Class от изпълнимия код. (Това обикновено става автоматично, само с „натискането на един бутон“). След това в този TestClass се пише код, които тества всички методи от основния Class. Кодът в TestClass е стандартен код за езика на който е написано приложението (C++, Java…), като самото тестване се осъществява с разновидност на функцията assert (assertEquals, assertTrue, assertNotNull, и още дост). И това е всичко.

Следва пускането на теста, което е още едно „натискане на един бутон“.

Unit Testing Framework

Може би забелязахте колко много неща стават с „натискане на един бутон“. Това е целта на Unit testing – да направи колкото се може повече от нещата при тестването лесни. За постигането на тази цел, се ползват Unit Testing Frameworks. Това са системи, които вършат доста неща вместо програмиста (нещата за които казах че са „натискане на един бутон“), и оставя на хората само това, което тя не може да свърши сама.

Отговорност на Unit Testing Framework са:

- Създаване на TestingClass за Class с изпълним код
- Осигуряване на assert функции
- Пускане на тест
- Пускане на всички тестове наведнъж
- Ясен начин за индикация дали всичко е минало успешно и ако не – къде са били намерени проблеми

Отговорност на програмиста:

- Писане на тестващия код (ползвайки assert функциите)

Най-известната такава система е фамилията xUnit. Тя има разновидности за почти всички езици: JUnit, PhpUnit, CppUnit, NUnit.
До колкото знам във Visual Studio има вграден друг Unit Testing Framework за C#.

Поради специфики на езика, на C++ не е толкова лесно да се направи Unit Testing Framework, затова и там е по-трудно да се ползва Unit Testing.

Code Coverage

Това е допълнително предимство ако се ползва Unit Testing. С него може да се визуализира кои редове са били изпълнени при пускането на Unit Test и кои редове са останали не изпълнени.

Code Coverage всъщност е метрика, която се записва в проценти.

Например ако за класа Math ви пише че имате 81% Code Coverage, това значи че 81% от редовете в кода са били изпълнени от Unit Test-а, а останалите 19% не са били изпълнени.

Money.cs

```
        /// the AddMoney helper.</summary>
        public IMoney Add(IMoney m)
        {
            return m.AddMoney(this);
        }

        public IMoney AddMoney(Money m)
        {
            if (m.Currency.Equals(Currency) )
                return new Money(Amount+m.Amount, Currency);
            return new MoneyBag(this, m);
        }

        public IMoney AddMoneyBag(MoneyBag s)
        {
            return s.AddMoney(this);
        }
```

1 Code Coverage. Изпълнените редове са зелени, неизпълнените са червени

Свързване с SVN

Възможно е (а също така е и често срещано) Unit Test-овете да се свържат с SVN системата. Така когато някой програмист commit-не нов код, веднага се пускат всички Unit Test-ове и на момента се проверява дали новият код не чупи нещо написано до сега.

Това е много полезно, защото колкото по-бързо се установи, че има проблем, толкова по-лесно се разбира кое точно прави проблема и съответно е много по-лесно той да бъде отстранен.

Unit tests като жива документация

Друго предимство на Unit Testing е, че сам по себе си представлява документация на кода. Защо? Защото в Unit Test-овете е описано в подробности как точно се очаква да се държи всяка една от функциите на всеки клас. Това си е точно документация.

И нещо повече, стандартните документации доста бързо стават „остарели“, защото кодът се развива по-бързо от тях и се стига до момент, когато в документацията пише едно, а кодът всъщност прави друго. Това не се случва с Unit Test-овете, защото те се развиват заедно с кода (иначе биха връщали грешен резултат или по-лошо – биха накарали програмата да crash-ва). Освен това успешното минаване на всички тестовете е достатъчно за да докаже че „кодът“ отговаря на „документацията“. Това не може да бъде постигнато при стандартната документация.