Testování Softwaru a Test-Driven Development

Literatura

- **Agile Principles, Patterns, and Practices in C#**
  MARTIN, Robert C.; MARTIN, Micah.
  Kapitola 4
- **The art of Unit Testing**
  OSHEROVE, Roy
  Kapitoly 1-5

Testování Softwaru

- Testování je nedílnou součástí vývoje každého softwaru a každý programátor minimálně jednou otestuje nějakým typem testu to co právě napsal.
- Nesystematické testování přináší řadu problémů, jako je neefektivita, neudržovatelnost kódu testů a špatné pokrytí testovaného SW.
- Existují metodiky jak správně integrovat testování do vývojového procesu – Test-Driven Development.
- Pro lepší pochopení smyslu testů při vývoji je třeba rozlišit jednotlivé typy testů:
  - Acceptance Testy
  - Unit Testy
  - Integrační Testy

Acceptance Testy

- Jsou testy na celém systému nebo aplikaci, které se provádějí za účelem zjištění, jestli systém (aplikace) dělá to, co zákazník požadoval.
- Měli by je psát samotní zákazníci, nebo systémový analytici jazykem, kterému dobře rozumí. Testy slouží jako specifikace a dokumentace k SW.
- Jsou typu **BlackBox**, netestují vnitřní strukturu SW ale jeho korektní chování na venek.

Unit Testy

- Unit test je část kódu, konkrétně metoda, která volá jinou část kódu a poté otestuje platnost nějakých předpokladů. Pokud se předpoklady ukážou neplatné, unit test selže.
- Unit testy jsou **WhiteBox** testy, znají strukturu testovaného SW a testují jeho dílčí funkcionalitu.
- Pokud unit test selže, mělo by to ukázat na konkrétní místo, kde je v SW chyba.
- Dobrý unit test má následující vlastnosti:
  - Měl by být automatický a opakovatelný
  - Měl by být jednoduše implementovatelný
  - Jakmile je jednou napsán, měl by být zachován pro budoucí používání
  - Každý by měl být schopen jej spustit
  - Měl by se dát spustit stiskem tlačítka
  - Měl by být rychlý

Jak Poznat Unit Test

V případě pochybností, je-li nějaký test unit test, je možné si položit následující otázky. Pokud alespoň na jednu z nich bude odpověď „ne“, pravděpodobně to unit test není.

- Mohu spustit a získat výsledky z testu, který jsem psal před měsícem?
- Mohou kolegové z týmu pustit testy psané před dvěma měsíci?
- Mohu spustit všechny unit testy, které jsem napsal v několika málo minutách?
- Mohu spustit všechny unit testy, které jsem napsal stisknutím jednoho tlačítka?
- Mohu napsat jednoduchý unit test během několika minut?

Výhody Unit Testů

Testy, které projdou předchozí sadou otázek by měli mít následující vlastnosti:

- Jsou persistentní, jakmile je otestována nějaká vlastnost, dá se kdykoli v budoucnu automaticky otestovat znovu. To zabraňuje vnášení nechtěných chyb do kódu při jeho úpravě.
- Jsou rychlé a automatické, lze je spouštět častěji při menších změnách kódu a tím zkrátit dobu vzniku a nalezení chyby.
- Dají se snadno psát, při přidání nové funkcionality lze snadno přidat i její testování.

Integrační Testy

- Integrační testování je testování dvou a více závislých softwarových modulů současně.
- „Špatné“ unit testy bývají integrační testy a přinášejí sebou celou řadu nevýhod:
- Při selhání integračního testu není jasné, která testovaná část selhala.
- Je složitější je psát a konfigurovat, protože je třeba vzít v úvahu závislosti mezi více moduly.
- Mají kratší dobu použitelnosti, protože pokud se změní jedna část testovaného celku, je třeba změnit i test.

Test-Driven Development

Test-Driven Development nebo také Test-First Development je přístup vyvíjení softwaru, kde jsou psány unit testy **před** produkčním kódem

- Opak zažitého přístupu: napiš kód, napiš test (pokud je čas), spusť test (pokud je čas), odlaď kód.
- V TDD, když je třeba naprogramovat nějakou funkcionalitu, se nejprve napíše unit test, který tuto funkcionalitu testuje. Poté se spustí všechny testy (nový test by neměl projít).
- Pak je naprogramován samotný produkční kód implementující funkcionalitu a opět se spustí všechny testy. Pokud je funkcionalita správná, nový test by měl projít. Pokud test neprojde je třeba odladit chyby.
- Pokud test projde, je produkční kód refaktorován nebo se pokračuje v psaní nového testu.

Vlastnosti TDD

Je-li TDD prováděno korektně, i přes vyšší nároky na čas kvůli psaní testů přináší celou řadu výhod, které se vyplatí:

- **Spolehlivost kódu** – zvyšuje spolehlivost kódu, protože jednotlivé části mají své testy, které jsou pravidelně spouštěny. Čím lépe napsané testy, tím větší pravděpodobnost že bude případná chyba rychle nalezena a opravena.
- **Dokumentace** – protože jsou testy jednoduché a snadno čitelné. Je z nich rychle vidět, jak se testovaná třída nebo metoda používá. Psaní dobrých testů nahrazuje psaní dokumentace.
- **Callability** – při psaní třídy (nebo metody) má programátor tendenci pohlížet na její fungování zevnitř. Při psaní testů je nucen pohlížet na třídu zvenčí a díky tomu je lépe schopen navrhnout třídu tak, aby se používala co nejjednodušeji.

Dopad TDD na Návrh

- Kromě motivace psát snadněji použitelné třídy má TDD mnohem zásadnější (pozitivní) dopad na návrh softwaru.
- Příkladem může být třída, která používá přístup do databáze, ze které načte výplaty zaměstnanců a vytiskne je.
- Pokud by vývojář implementoval tuto funkcionalitu tak, že třída má reference přímo na třídu poskytující připojení k databázi a třídu pro komunikaci s tiskárnou (čímž poruší OCP a DIP), v TDD narazí na problém ještě než napíše první řádek.
- Podle TDD musí napsat test na logiku třídy: třída načte data z DB a ty pak pošle na tiskárnu. Zde však přichází celá řada problémů. Jakou databázi a tiskárnu použít? Potřebujeme mít třídy implementované abychom mohli testovat? Neděláme integrační test?
- Řešení je odstínit třídy pomocí vhodné abstrakce a použít Mock Object pattern při testování. Všimněte si, že jako vedlejší produkt dodržování TDD jsme dostali návrh, který neporušuje OCP ani DIP.

Testovací Frameworky

- Testovací frameworky nabízí celou řadu nástrojů pro jednoduchou a snadno udržovatelnou práci s testy.
- Poskytují třídy pro psaní testů, verifikaci předpokladů, *stub* a *mock* objektů atd.
- Poskytují nástroje pro spouštění testů, které umí identifikovat test v kódu a automaticky jej spustit. Podporují spouštění testů podle kategorií apod.
- Zobrazování výsledků testů: které testy byly vykonány, které prošly, které selhaly a proč.
- Zobrazování statistik a **Code Coverage** (otestované a neotestované řádky v kódu).

Testovací Frameworky – Příklady

- **NUnit** – asi nejpoužívanější testovací framework pro .NET. Patří do rodiny xUnit frameworků (Junit, CppUnit …)
  http://www.nunit.org/
- **MS Visual Studio Team Test** – součást Visual Studia 2010 Ultimate, umožňuje generování testovacích tříd a metod, jejich spouštění, načítání testovacích dat z databáze a zobrazování Code Coverage.
- **Rhino Mock** – Isolation framework, obsahuje generické *mock* a *stub* objekty, pomocí nichž lze izolovaně testovat funkcionalitu tříd zavislých na jiných třídách.
  http://ayende.com/Blog/archive/2009/09/01/rhino-mocks-3.6.aspx

Struktura Unit Testů

- Unit Testy se většinou nacházejí v odděleném projektu (tedy .dll knihovně) od produkčního kódu.
- Každé testované třídě odpovídá jedna nebo více testovacích tříd s testy. Tyto třídy jsou označeny atributem `[TestClass]`.
- Jednotlivé unit testy jsou potom metody z testovací třídy označené atributem `[TestMethod]`.
- Označování testovacích tříd a metod atributy umožňuje testovacím frameworkům automaticky spouštět testy z .dll knihoven.
- Při pojmenovávání testů je dobré dodržovat následující konvence:
  - Jméno projektu: [JmenoTestovanehoProjektu].Tests
  - Jméno třídy: [JmenoTestovaneTridy]Tests
  - Jméno metody: [JmenoTestovaneMetody]_[StavPriTestovani]_[OcekavanyVysledek]

Další Atributy a Metody

- Kromě samotných testovacích metod mohou být v testovací třídě přítomny inicializační a čistící metody.
- Je užitečné je používat ze dvou důvodů: při spuštění více testů najednou může docházet k jejich vzájemnému ovlivnění. Inicializační a čistící metoda může zajistit, že se pro každý test vytvoří nové zdroje, které nejsou ovlivněny předchozími testy. Navíc je inicializace na jednom místě a lze ji jednoduše změnit.
- Metody jsou označeny těmito atributy:
  - `[TestInitialize]` Inicializační metoda pro test, spustí se před každým testem.
  - `[TestCleanup]` Čistící metoda pro test, spustí se po každém testu.
  - `[ClassInitialize]` Inicializační metoda pro testovací třídu, spustí se pouze jednou před všemi testy
  - `[ClassCleanup]` Čistící metoda pro třídu, spustí se pouze jednou po všech testech
- Pro filtrovaní testů nebo jejich dočasné vypnutí lze testovací metody označit atributy `[TestCategory]` resp. `[Ignore]`.

Struktura Testu – AAA

- Unit test je jedna jednoduchá metoda, která by měla testovat vždy jen jednu věc a která má strukturu *Arrange*, *Act* a *Assert* (AAA).
- Část **Arrange** inicializuje testovanou třídu (pokud se tak nestalo v inicializační metodě) a nachystá vše potřebné pro vykonání testovaného kódu.
- V části **Act** se vykoná testovaný kód.
- V části **Assert** se zkontrolují předpoklady, které by měli platit po vykonání testovaného kódu. To se děje pomocí statických metod třídy `Assert` z testovacího frameworku, které v případě neplatných předpokladů způsobí selhání testu.
- V případě testování vyhození výjimky je místo **Assert** části použit atribut `[ExpectedException]`.

Testování Návratové Hodnoty

```
[TestMethod]
public void Multiply_PassTwoNumbers_ReturnsItsProduct()
{
    // arrange
    var calc = new Calculator();

    // act
    int result = calc.Multiply(3, 4);

    // assert
    Assert.AreEqual(12, result, “incorrect multiplication”);
}
```

Testování Vyhození Výjimky

```
[TestMethod]
[ExpectedException(typeof(ArgumentNullException))]
public void Open_PassNullConStr_ThrowsArgumentNullExc()
{
    // arrange
    var connection = new DbConnection();

    // act
    connection.Open(null);
}
```

Izolace Tříd

- V některých případech nestačí pouze volat testované metody a kontrolovat předpoklady. Testovaná třída může používat další třídy, které je třeba izolovat, aby neovlivňovali výsledky testů (unit test vs. integrační test).
- V takových případech je třeba refaktorovat třídu, jak bylo popsáno v příkladu tisknutí dat z databáze a podstrčit (injektovat) testované třídě speciální objekty, které zaručí korektní průběh testu
- Přes tyto objekty lze předávat testované třídě testovací data nebo monitorovat interakce s testovací třídou. Testy lze rozdělit do dvou skupin:
  - *State-based testing* – je vykonán testovaný kód a kontroluje se, jestli se třída dostala do očekávaného stavu (předchozí dva příklady)
  - *Interaction testing* – testuje se, jestli při vykonávání testovaného kódu třída správně interaguje s dalšími objekty (např. zavolá metodu pro čtení z databáze a metodu pro tisk výsledků)

Stub a Mock objekty

Při testování se používají dva typy objektů:

- **Stuby** – slouží pouze k tomu, aby nahradili objekty s kterými třída interaguje, aby nenarušovaly průběh testování. Mohou být přes ně předávána testovací data. Stuby nikdy nemohou způsobit selhání testu.
- **Mocky** – na rozdíl od stubů si pamatují, jak byly při testu volány jejich metody, a na konci testu je vyhodnoceno, jestli toto volání odpovídá předpokladům. Slouží pro *interaction testing* a mohou způsobit selhání testu. Lze je dále rozdělit na
  - **Strict mocky** – způsobí selhání testu pokud nebyla zavolána metoda, která zavolána být měla, nebo byla zavolána metoda, která nebyla očekávána.
  - **Dynamic mocky** – způsobí selhání testu pokud nebyla zavolána metoda, která zavolána být měla. Pokud byla navíc zavolána metoda, která nebyla očekávaná, na výsledek testu to nemá vliv.
- Testované třídě může být injektováno více těchto objektů, ale podle pravidla, že test testuje pouze jednu věc by mezi nimi měl být pouze jeden mock objekt.

Injektování Objektů

Testovanou třídu je někdy třeba refaktorovat tak, aby do ní bylo možné injektovat stub nebo mock. Nejčastější možnosti jsou:

- Předání objektu přes konstruktor.
- Předání objektu přes property.
- Podstrčení objektu přes *Factory.*
- Podstrčení objektu přes lokánlí *Factory* – přístup k objektu je ve třídě izolován do *virtual protected* metody. Z testované třídy je poděděn nový objekt, který přepíše metodu tak aby vracela stub nebo mock. Testy jsou poté prováděny na odvozeném objektu.

Práce se Stuby a Mocky

- Izolační frameworky (jako Rhino Mock) nabízejí generické nástroje pro práci s mocky a stuby.
- Lze jim nastavit chování způsobem *record–reply*: nejprve je v testu definováno, jak budou volány a jak na to mají reagovat. Při vykonávání testovaného kódu pak toto chování zopakují.
- Mocky si navíc uchovávají informace o provedených voláních, které jsou na konci testu porovnány s předpoklady.
- Framework nabízí celou řadu užitečné funkcionality, jako:
  - Vyhazování výjimek
  - Vyvolávání událostí
  - Složitější kontrola parametrů
  - Atd.

Použití Stubu

```
[TestMethod]
public void Send_SendingFails_ReturnFalse()
{
    MockRepository mocks = new MockRepository();
    ICommunicationChannel channel = mocks.Stub<ICommunicationChannel>();

    // record
    using(mocks.Record())
    {
        channel.Send("message");
        LastCall.Throw(new Exception("sending failed"));
    }

    // reply
    adapter.Channel = channel;
    bool result = adapter.Send("message");
    Assert.IsFalse(result, "Failed sending has returned true");
}
```