TECHNICKÁ UNIVERZITA V LIBERCI

Fakulta mechatroniky, informatiky a mezioborových studií

Studijní program: B2646 – Informační technologie

Studijní obor: 1802R007 – Informační technologie

Testování stability a zvyšování spolehlivosti knihoven pro vývoj programu ENVIS

Testing and Stability Improvements of Libraries for ENVIS Software Development

Bakalářská práce

Autor: Vítězslav Kačmár

Vedoucí práce: Ing. Tomáš Tobiška

Konzultant: Ing. Jan Kraus

V Liberci 17. 5. 2011

Zadání

- Seznamte se s technikami pro psaní testování kódu v jazyce C# pro platformu .NET
- Doplňte knihovny pro manipulaci s daty aplikace ENVIS o vhodné unit testy a tam, kde je to vhodné, i o automatickou kontrolu konzistence dat
- Aktualizujte dokumentaci doplněných tříd pomocí nástrojů pro automatickou tvorbu dokumentace
- Shrňte výhody a nevýhody jednotlivých zkoumaných metod a uveďte jejich praktický dopad na Vámi testované knihovny

Prohlášení

Byl(a) jsem seznámen(a) s tím, že na mou bakalářskou práci se plně vztahuje zákon č. 121/2000 o právu autorském, zejména § 60 (školní dílo).

Beru na vědomí, že TUL má právo na uzavření licenční smlouvy o užití mé bakalářské práce a prohlašuji, že **s o u h l a s í m** s případným užitím mé bakalářské práce (prodej, zapůjčení apod.).

Jsem si vědom(a) toho, že užít své bakalářské práce či poskytnout licenci k jejímu využití mohu jen se souhlasem TUL, která má právo ode mne požadovat přiměřený příspěvek na úhradu nákladů, vynaložených univerzitou na vytvoření díla (až do jejich skutečné výše).

Bakalářskou práci jsem vypracoval(a) samostatně s použitím uvedené literatury a na základě konzultací s vedoucím bakalářské práce a konzultantem.

Datum

Podpis

Poděkování

Poděkovat bych chtěl všem, kteří mě při práci podporovali a pomáhali mi, ať už fyzicky nebo psychicky. Hlavně bych chtěl poděkovat konzultantovi a vedoucímu práce za trpělivost a pochopení. Určitě se mnou měli veliké starosti a doufám, že jejich čas strávený se mnou nebyl zbytečný.

Abstrakt

Práce se v úvodu zabývá obecným pojmem testování a důvody jeho používání. Představuje různé typy testů a rozděluje je do různých skupin. Zaměřuje se na konkrétní typ, detailněji ho rozebírá, vyjmenovává jeho výhody a nevýhody, představuje ukázky toho, jak vypadá, vytváří se a spouští. Práce dále vysvětluje strategii vývoje kódu a jeho různé postupy, pár jich představí. Seznámí se s pojmem a teorií automatické dokumentace a vytváří ukázku její tvorby ve zvoleném nástroji. Představí zvolenou testovanou knihovnu, kterou popíše. Pokusí se vysvětlit její slabá místa a ukáže, jakým směrem se budou testy vyvíjet. Na konci práce odhaluje výsledky testů.

Klíčová slova:

Testování, Unit testy, strategie vývoje kódu, vodopád, vývoj řízený testy (TDD), agilní metodiky, automatická dokumentace, komentáře.

Abstract

Introduction of the thesis deals with testing in general and reasons why using them. It presents different types of tests and divides them into various groups. It focuses on specific type,analyzes it in detail, enumarates its pros and cons, shows samples, how it looks like, is created and runs. The thesis then explanes strategy of source development and its different approches. It familiarizes with term automatic documentation and shows sample of its creation in chosen tool. It introduces chosen tested library and then it describes it. It tries to explain weaknesses of the library and shows the direction where the test development will head. At the end of thesis reveals the results of tests.

Keywords:

Testing, Unit testing, Strategy Development Code, Waterfall, Test-driven development (TDD), Agile methodology, Automatic documentation, Comment

Obsah

Seznam obrázků

Úvod

Testování je velmi užitečnou technologií pro vývojáře, umožňuje jim velmi rychle získat zpětnou vazbu o stavu vyvíjeného produktu a tím dříve představit zákazníkovi výsledný produkt. V době norem, kdy existuje prakticky na cokoliv nějaký doklad nebo certifikát dokazující jeho nezávadnost, se objevuje tato technologie, která podporuje rychlost a nízkou chybovost vývoje.

V úvodní části se práce zaobírá teorií testování a jejími výhodami. Protože samotný pojem pokrývá velkou oblast a pro každou existuje konkrétní technologie, tak se je v práci pokusím vyjmenovat a vysvětli. Oblasti lze rozdělit do několika skupin, podle různých kritérií, kterými můžou být rozsah, náplň nebo přistup. Práce popisuje většinu z nich, kde nakonec jednu konkrétní více přiblíží. Zaměřuje se především na testování jednotek kódu, které se nazývá Unit testing. Popisuje, o jakou problematiku se stará nebo řeší, vyjmenovává jeho výhody a nevýhody. Obsahuje pravidla, která jsou potřeba při používání dodržovat a hlavně, čeho lze používáním docílit. Nezbytnou součástí jsou zásady pro psaní testů, kde se popisuje, jak mají obecně jednotlivé testy vypadat, jakým směrem se ubírat a co vlastně testovat. Aby bylo vůbec možné tuto činnost provádět, tak v kapitole nástroje jsou vyjmenované prostředky, které lze použít. Na trhu existují dva typy frameworků a rozdíly mezi nimi a hlavně jejich výhody a nevýhody jsou popsány v kapitole nástrojů. Pro jednoho ze jmenovaných nástrojů je pak připravena ukázka toho, jak se test vytváří, spouští a vyhodnocuje.

V další části práce jsou popisovány různé způsoby vývoje kódu, mezi jehož body patří i právě zmiňované testování. Snahou je snížit celkový počet chyb ve výsledném produktu a to za různých okolností. Některé slouží jen pro snižování chyb, jiné zase řeší ochranu proti pádu aplikace nebo umějí vyvíjet kód za krátký čas s měnícím se zadáním. A protože je potřeba k produktu i dokumentace už z jakéhokoliv důvodu je součástí práce vytvoření automatické dokumentace spolu s možným nástrojem na použití.

Nakonec práce aplikuje výše zmiňované body na otestování knihovny, kterou podrobí různým testům. Vytvoří dokumentaci ke svému dílu a ve finále představí výsledky, které byly zjištěny, případně problémy, s nimiž se bylo nutné potýkat.

1 Testování

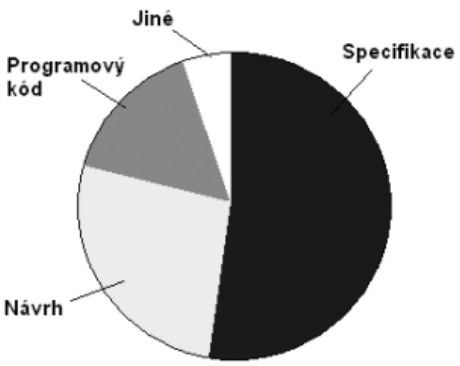
Testování je zjišťování určité informace o konkrétním produktu nebo technologii. Dále s tím spojené hledání chyb v dané problematice v různém odvětví průmyslu [6]. Mezi obecně nejznámější patří například testy: výkonnosti (hardwarová náročnost), bezpečnosti (zranitelnost), spolehlivosti (stabilita), funkčnosti (správnost implementace), použitelnosti (rozsah pokrytí), atd.

Implementováním se snažíme snížit celkový počet chyb v konečném produktu. Ověřit správnost specifikace a všech bodů zadání [6]. Zabránit pádu aplikaci při neočekávané chybě. Testováním se snažíme pokrýt všechny možné a neočekávané požadavky klienta na aplikaci nebo produkt. Snažíme se docílit nejen správné funkčnosti, ale i správné nefunkčnosti [5].

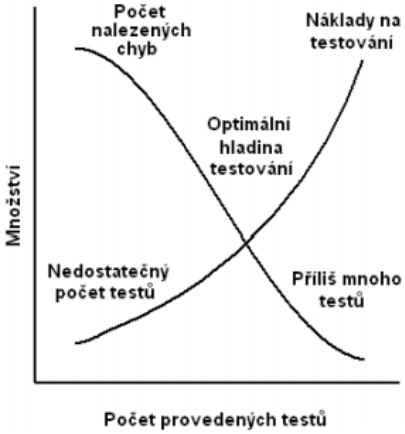
Vyžaduje nezávislý pohled na věc a trochu odlišné myšlení, proto tuto činnost mají nastarosti testeři z vývojářského týmu. U malých společností si však testy píšou sami programátoři. Činnost testerů ovlivňuje i zbytek týmu. Analytik návrhem specifikace, která se může neustále měnit. Programátoři vlastní implementací a opravami nalezených chyb.

Testování je nezbytnou součástí každé větší, ale i menší společnosti, které jde o vytvoření kvalitního produktu. Chyba se sama nenajde a pro nalezení jehly v kupce sena, v rozumném čase, je zapotřebí efektivních prostředků. Při programování vznikají různé chyby, které je obtížné zjistit běžnými postupy, a proto se je snažíme odhalit pomocí testů [4]. Ale ani testování není jednoduchou záležitostí. Má svá úskalí, problémy a omezení, kterými se musí řídit.

Jeden z možných problémů nebo omezení je ten, že ne každou situaci, která může nastat, lze pokrýt testem. Pokud test nenalezl chybu, neznamená to ještě, že tam není, může se také jednat o špatně napsaný test. A nakonec je potřeba testy neustále přizpůsobovat měnícím se okolnostem [6].



Obrázek 1 Výskyt chyb [1]



Obrázek 2 Náklady testů [1]

Obrázek 1 popisuje nejčastější výskyt a množství chyb a obrázek 2 ukazuje charakteristikou náklady na testování, tj. objevení a opravení chyb.

1.1 Rozdělení testů

1.1.1 Podle přístupu

Black box

Tester nemá přístup k testovanému kódu. Na vstup se zadávají různá data a sleduje se reakce objektu, respektivě výstup, ten se pak porovnává. Takovéto testy bývají často psány před napsáním samotného kódu a typicky oproti specifikaci. Obyčejným příkladem může být kalkulačka. Neznáme vnitřní mechanismus, zadáváme vstup, sledujeme výstup [18].

White box

Tester má přístup k testovanému kódu. Testy vznikají až po jeho napsání a typicky jsou psány, orientovány podle jeho struktury. Snažíme se, o co největší pokrytí kódu. A tímto způsobem lze psát testy, které opravdu najdou chyby [18].

1.1.2 Podle rozsahu

Rozsahem myslíme, jak velkou částí kódu se testy zaobírají. Unit testy testují jednotky kódu programu (proto unit), kterými jsou typicky třídy a jejich metody. O velké kusy se starají testy komponent. Vzájemnou funkčnost, například jednotlivých modulů, zjišťují integrační testy. A nakonec systémové testy testují funkčnost jako celek.

1.1.3 Podle náplně

Do této kategorie spadají regresní testy, to jsou testy, které jsou spouštěny automaticky, tzn. každý automatický test, se stává regresním testem. Dále pak testy zabývající se určitou náplní jako výkonnost, použitelnost, atd., viz začátek.

1.2 Unit testy

Patří asi mezi nejznámější a nejčastěji zmiňované testy, když se mluví o pojmu testování. Jsou používány pro otestování jednotky kódu (typicky třídy a jejich metody). Jejich úkolem je automatizovaným způsobem zjistit, jestli konkrétní jednotka dělá to, co dělat má a naopak, tj. nedělá, co dělat nemá.

Výhody

Unit testy nám mohou posloužit jako ochranná vrstva, která může při změně kódu odhalit nově zanesenou chybu. Výsledkem toho je podpora změn a častější refaktorizace kódu. Zjednodušují integraci a slouží také jako dokumentace. Pomáhají oddělit rozhraní od implementace. Nutí k neustálému zamyšlení nad designem. To, že testy reprezentují uživatele, vede k vytvoření lepšího rozhraní. Zaručují 100% pokrytí kódu. A díky jasnému výsledku testu (PASS/FAIL) nabízí rychlou zpětnou vazbu [18].

Nevýhody

Svým psaním a dodržováním podmínek vyžadují disciplínu. Zpomalují psaní a úpravu kódu, jelikož se musí provádět i úprava testů. Můžou omezit experimentování a to z toho důvodu, že k cíli vede víc cest, ale dovolit si je všechny vyzkoušet stojí moc času. Vybere se nejlepší a pod ní se pak provádí testování [18].

1.2.1 Pravidla

Abychom mohli daný test vůbec nazvat unit testem, musí splňovat několik základních požadavků. Mezi nejzákladnější podmínky patří, že nesmí pracovat s externími daty, tzn.: nesmí pracovat s databází, nesmí komunikovat po síti, nesmí sahat na souborový systém. Musí být nezávislí, tzn.: musí být schopen běžet současně s ostatními testy a také běžet, bez potřeby nastavování běhového prostředí [18].

Samozřejmě, že potřebujeme testy, které tyto věci dělají, ale nemůžeme je v žádném případě nazývat unit testy. I když je lze napsat za pomocí unit-testing frameworku. Aby vůbec měly unit testy nějaký smysl a vyplatilo se do nich investovat, snažíme se o maximální možné splnění jejich hlavních kritérií a cílů, kvůli kterým je používáme.

Kritéria

Při psaní unit testů, se snažíme o co nejširší pokrytí testovaného kódu (tzv. code coverage). Jak již bylo zmíněno test je zástupcem uživatele a naší snahou je otestovat, co nejvíce možných případů, které můžou nastat.

Cíl

Hlavním cílem je dosažení toho, aby nám test vracel správné hodnoty na správné vstupní údaje a správné hodnoty na mezní údaje. Dále, aby program správně zareagoval na nesprávné vstupní údaje, tj. výpisové hlášky, odchytávání výjimek, atd. A v neposlední řadě, aby legitimně zhavarovat, když zhavarovat má a není už jiná možnost.

Převod do testu

Na unit testy se snažíme převést všechno, co jsme si kdy zkoušeli otestovat ručně, dále pak všechny možné testovací a ladící funkce. A samozřejmě pak testy, které zjišťují výskyt chyb.

1.2.2 Zásady pro psaní testů

Pro začátek bychom měli dodržovat, že pro každou testovací třídu si vytvoříme jednu třídu testů. V té pak máme pro jednu testovanou metodu, více těch testovacích. A to z toho důvodu, že každá se zaobírá právě jedním konkrétním problémem, který řeší. Je samozřejmé, že na jednu metodu bude spadat více problémů. V samotných testech se pak snažíme projít všechny možné cesty v testovaném kódu. Tzn., máme-li například v kódu podmínky, procházíme všechny její větve. Zaměřujeme se na okrajové případy, je-li např. vstupem číslo, tak zkoušíme reakci na hodnoty: 0, MAX_VALUE, MIN_VALUE. Dále pak zadáváme chybná vstupní data, protože jak už bylo zmiňováno, naší snahou je docílit i kladné reakce na špatný vstup. Výjimky, které probublávají ven, neodchytáváme, necháváme je, ať shodí test. Velkou pozornost

věnujeme složitým pasážím. Hlavně nesmíme zapomínat na to, že kód testů, je taky kód a tudíž pro ně platí stejná pravidla. Měl by být čistý a přehledný, tj. měla by se udržovat nějaká zvolená struktura. Měl by obsahovat vhodné komentáře a nemělo by se zapomínat i na ty dokumentační, protože jak bylo řečeno, testy slouží i jako dokumentace. Minimalizovat nebo redukovat duplicitu, atd.

Vývojový cyklus

Vývojový cyklus začíná psaním testu, aniž bychom měli napsaný kód. Napíšeme tedy test, následně ho spustíme a ověříme jeho nesplnění („FAILNUTÍ“). Po té napíšeme kód, který už se bude testovat, a vložíme do něj minimum pro splnění testu. Spustíme test a ověříme jeho splnění. Poté můžeme refaktorizovat kód a opakovat cyklus. Tímto způsobem nám test ukáže, kdy přejde z funkční fáze do té nefunkční.

1.2.3 Nástroje

Aby bylo vůbec možné provádět testy, musela vzniknout technologie, která obsahuje příslušné třídy, metody, atributy, které umožňují vytváření, spouštění a vyhodnocování testů. Jejich celá řada a jejich názvy většinou začínají písmenem programovacího jazyka, který podporují.

Pro malou ukázku můžu zmínit JUnit, který slouží jazyku Java, PyUnit podporuje Python, Test::Unit zase Ruby. Pro platformu Microsoftu, tedy .NET, je jich hned několik. Mezi asi nejznámější patří NUnit a mezi relativně nové Gallio, který je zajímavý v tom, že podporuje hned několik frameworků najednou. Jako vývojový nástroj, který obsahuje integrovaný framework, bych rád zmínil Microsoftu Visual Studio 2008.

Za zmínku stojí Gallio Icarus, který podporuje většinu zmiňovaných frameworků a spousty dalších, které zde uvedeny nejsou. Ale stejně jako ostatní, ani on, neumožňuje psaní kódu uvnitř sebe sama. Tudíž nejdříve potřebujeme nějaké integrované vývojové prostředí (IDE), kde nejdříve napíšeme testy a další, kde budou spouštěny. Navíc v sobě neobsahuje i jiné důležité funkce, jako je code coverage, který podporuje např. NUnit. Naštěstí existuje nástroj, který tohle všechno umí a tím je Microsoft Visual Studio 2008.

Externí versus integrované frameworky

Externí frameworky jsou většinou zaměřeny na přehlednější a jednodušší zobrazení i spouštění testů. Testy jsou zobrazovány podle jejich stromové struktury, tzn., v jakém jmenném prostoru jsou a v jaké třídě. Mívají více statistických údajů například o délce trvání testů, atd. Také podporují export výsledků do více typů formátu. Neumí však nebo spíš nedá se v nich vyvíjet kód, slouží pouze pro práci s již hotovými testy. To pak vyžaduje mít více nástrojů pro práci, přepínat mezi nimi a výsledkem toho je jen ztráta času.

Velkou výhodou integrovaných frameworků spočívá v tom, že vše potřebné mají v sobě v jednom vývojovém prostředí. To má za následek menší nepřehlednost, ale za to se v nich dá vyvíjet kód, psát testy, spouštět je, vyhodnocovat, zobrazit menší statistiku, nastavit v jakém pořadí se budou testy spouštět a nakonec exportovat výsledek do nějakého podporovaného formátu.

1.2.4 MS Visual studio 2008

Visual Studio je integrované vývojové prostředí (IDE) pro vývoj aplikací pro platformy .NET. Obsahuje velké množství nástrojů, které usnadňují práci programátorů. Pro nás je důležité, že v něm můžeme vyvíjet kód. A díky unit-testing frameworku, testovat produkt.

Obsah frameworku

Nás bude zajímat namespace Microsoft.VisualStudio.TestTools.UnitTesting, kde se nachází pro nás nejdůležitější statická třída Assert. Obsahuje metody (asserty) pro vyhodnocování testů podle kritérií, jež nás v dané konkrétní situaci zajímají. V jednom testu můžeme mít samozřejmě více assertů.

Asserty

Metody jsou přetížené a mají tedy několik různých vstupních parametrů. Pro vyhodnocování dvou objektů stejného typu složí Assert.AreEqual. Pro porovnávání shody dvou referencí slouží Assert.ReferenceEquals. Ke zjištění, jestli je daná instance daného typu, obstarává Assert.IsInstanceOfType. Kdybychom potřebovali zjistit, zdali mají dva dané objekty stejnou referenci, použili bychom Assert.AreSame. Ke zjištění odpovědi na otázku *ano* nebo *ne* můžeme zase použít Assert.IsTrue nebo Assert.IsFalse.

Pokud z nějakého důvodu potřebujeme zařídit, aby test neuspěl, použijeme k tomu Assert.Fail. Může se stát, že chceme vědět, jestli je nějaký objekt nastaven na NULL. K tomu je určen Assert.IsNull. Speciální případem je Assert.Inconclusive, který se používá v případě, když nechceme, aby test uspěl, ale ani selhal, tzv. nevyhodnotitelný vztah. Všechny zmiňované asserty mají i své negativní protějšky.

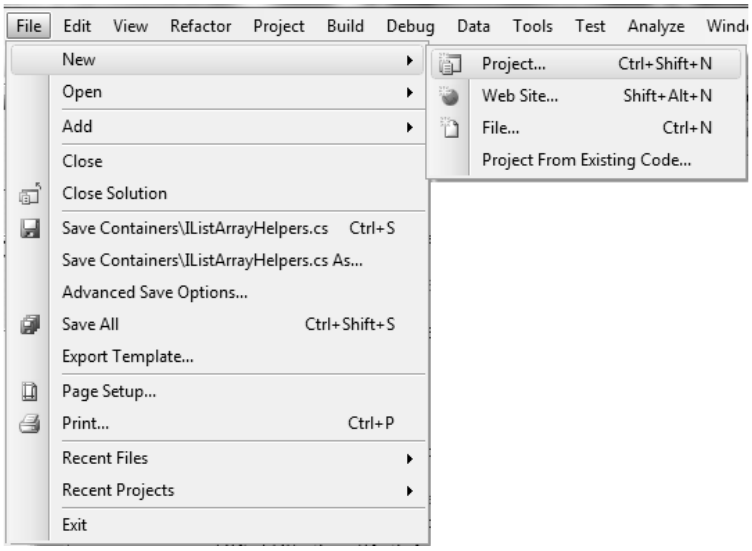
Dále nám framework nabízí atributy, které se vkládají před testovací kód a popisují jeho charakter a tím i určují, kdy a jak se budou spouštět. Píší se do hranatých závorek a beze slova atribut.

Atributy

Pro popisování testovací třídy slouží *TestClassAttribute* a její metody dostávají značku *TestMethodAttribute*. Pokud potřebujeme nastavit nějakou inicializační část před spuštěním všech testů, používá se k tomu *ClassInitializeAttribute*. Po skončení posledního testu uklidí *ClassCleanupAttribute*. Před každým a po každém testu se provádí *TestInitializeAttribute* a *TestCleanupAttribute*. A je-li potřeba z jakéhokoliv důvodu odchytit nějaký typ výjimky, poslouží k tomu *ExpectedExceptionAttribute*.

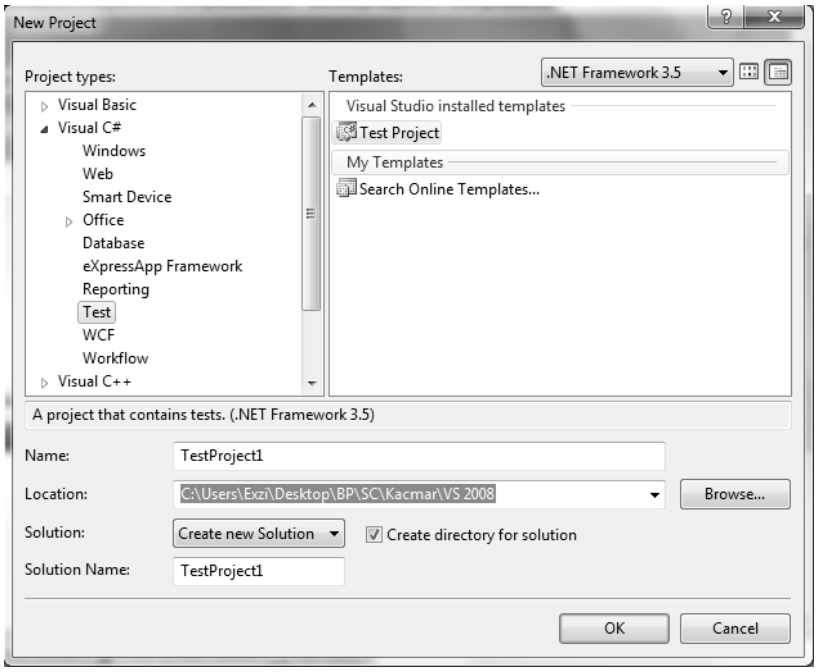
Vytvoření unit testu ve VS 2008

Vytvoření unit testu ve visual studiu je stejné, jako u jakéhokoliv jiného projektu. V nabídce menu zvolíme položku File, ze seznamu vybereme New a jako novou položku zvolíme Projekt.



Obrázek 3 Vytvoření projektu

Objeví se nabídka, kde v levé části je panel obsahující programovací jazyky a k nim podporované typy projektů. My si vybereme námi požadovaný jazyk, v tomhle případě C# a typ projektu Test. V nabídce se pak už jen objeví Test Project, kterému přidělíme jméno a místo uložení.



Obrázek 4 Výběr projektu

Napsání jednoduchého unit testu ve VS 2008

Příklad jednoduchého unit testu může být cokoliv, zvolil jsem ukázku testování shody dvou čísel. Jako vyhodnocovací metoda je použita Assert.AreEqual, která, jak název napovídá, testuje shodu dvou zadaných objektů stejného typu. V ukázce je také vidět dodržování psaní komentářů a přehlednost kódu.

```
/// <summary>
/// Test vyhodnocujici rovnost dvou čísel
/// </summary>
[TestMethod]
public void TestRovnosti()
{
    //
    // TODO: test shody dvou cisel
    //

    int a = 2;
    int b = 3;
    Assert.AreEqual(a, b);
}
```

Obrázek 5 Jednoduchý unit test

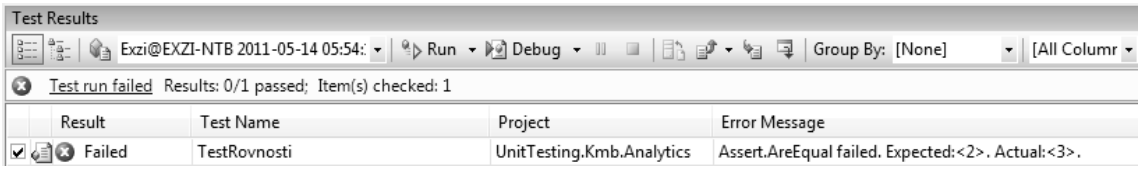
Spouštění unit testu ve VS 2008

Z panelu vývojového prostředí nás bude nejvíce zajímat toolbar zobrazený na obrázku, který se stará o obsluhu unit testů. Spouštění testů se provádí druhý až pátým tlačítkem, kde každý, až na malé odlišnosti, dělá to samé. Obsahuje, kromě i jiných funkcí, také editor testů, kde lze zvolit, jaké testy mají být spuštěny a v jakém pořadí.

Obrázek 6 Toolbar obsluhující unit testy

Výsledky unit testu ve VS 2008

Výsledky testů jsou pak zobrazeny v panelu Test Results, který obsahuje seznam spuštěných testů a v případě jejich neúspěchu i jeho důvod. Zobrazuje menší statistické údaje, jako například kolik jich bylo spuštěno a jaký počet uspěl a neuspěl. Při dvojkliku na daný test, zobrazí detailnější zprávu s více informacemi o jeho průběhu.



Obrázek 7 Výsledky unit testů

2 Strategie vývoje kódu

Zvolením správné strategie se snažíme docílit, snížení celkového počtu chyb a eliminaci různých dalších nežádoucích akcí, jako je pád aplikace na neočekávaný vstup od uživatele. Dále nám definuje cyklus psaní a testování kódu během vývoje.

2.1 Defenzivní a ofenzivní programování

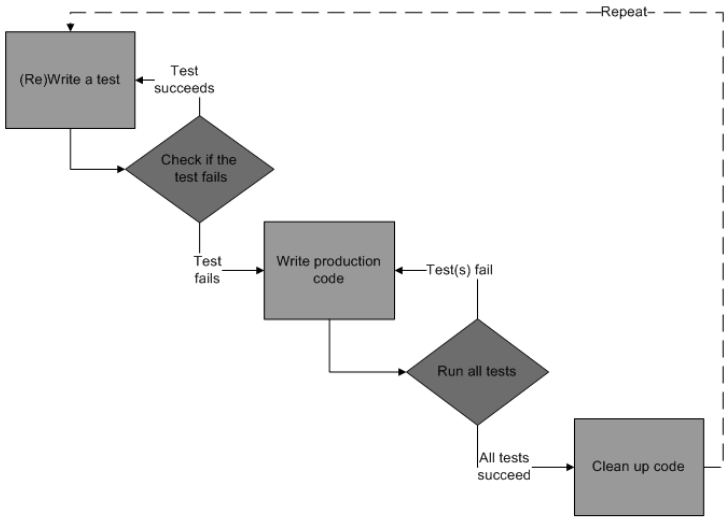
Defenzivním programováním se snažíme předejít chybám ze vstupu nebo jiných externích zdrojů dat. Cílem je nevpustit špatná data do implementace a nezpůsobit si pád aplikaci nebo vznik jiné nežádoucí chyby. Veškerá data, která budeme dostávat je potřeba si nejdříve zkontrolovat a provádět s nimi operace, až když budou splňovat podmínky, které požadujeme. Příkladem může být test úspěšného načtení dat ze souboru, test přístupných vstupních hodnot a zvýšená opatrnost se doporučuje při práci s řetězci.

Ofenzivním programováním chceme, aby se chyby projevily co nejdříve a tím mohly být opraveny ještě během samotného vývoje produktu. A aby neošetřená chyba nezpůsobila pád aplikace a nezavinila tím např. ztrátu informací.

Rozdíl mezi těmito dvěma styly je v tom, že defenzivní programování, jak název napovídá, se snaží předejít vzniku chyby. Kdežto ofenzivní programování výskyt chyby očekává a snaží se ji najít a opravit nebo zabránit chybě způsobit škodu, tj. např. způsobit pád aplikace.

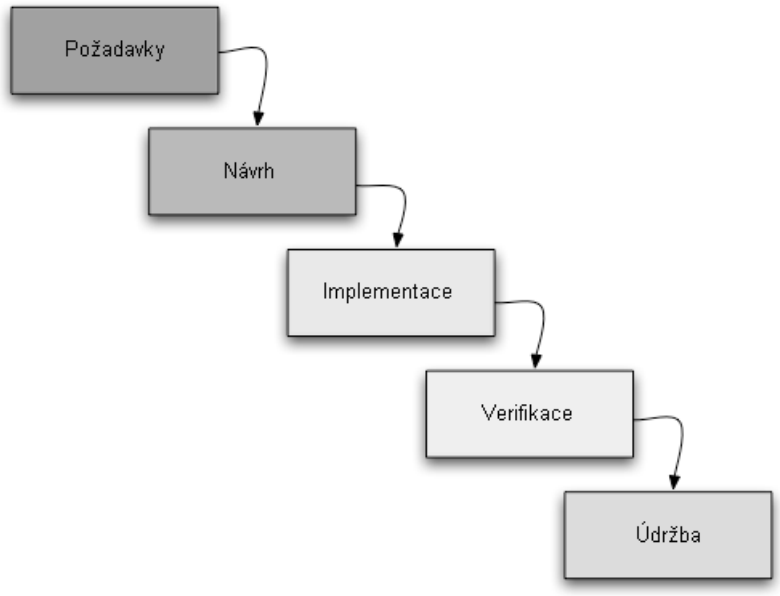
2.2 Test-driven development

Vývoj řízený testy, zobrazený na obrázku 3, je základní model testování používaný při programování s automatizovanými testy. Popisuje klasický cyklus programování s testováním, který díky unit testům zaručuje pokrytí každého kusu (jednotky) kódu. Kvůli jednoznačnosti výstupu testu (prošel nebo neprošel) a následné opravě kódu, který v testu neuspěl, je tento model intuitivně používán. Nenáročnost a absence potřeby znát spousty informací si tento způsob nese velké obliby.



Obrázek 8 Vývoj řízený testy [2]

2.3 Waterfall



Obrázek 9 Vodopádový model [3]

Model vodopád popisuje celkový postup při vývoji produktu, od samotného začátku až do jeho vydání zákazníkovi. Existuje jeho několik různých modifikací, původní model je zobrazen na obrázku 4. U tohoto modelu se přistupuje od jedné fáze, k druhé (sám model to vyžaduje), až když je ta předním kompletní a řádně připravená. Pečlivostí a věnováním dostatku času v počáteční fázi vývoje softwaru, se snaží docílit snížení nákladů (peněz, času, úsilí) v pozdějších etapách vývoje. Model má však i kritiky, kteří namítají jeho nepoužitelnost v praxi. Neschopnost demonstrují například na návrh specifikace, kdy ještě ani sám zákazník vlastně neví, co chce a svoje představy

si ujasní až poté co uvidí nějaký funkční prototyp. A v tu chvílí přijde nazmar spousty práce. Vyplývá z toho, že model je vhodné použít tam, kde se neustále nemění požadavky na jednotlivé jeho větve.

2.4 Agilní vývoj

Agilní metodiky jsou postupy, navržené pro konkrétní řešení problému, např. pro maximální rychlost vývoje, měnící se zadání, atd. S tímto vývojem je spjat termín extrémní programování. A to protože se zde všechno dobré pro programování, dotahuje do extrémů, jako otázka „Je dobré průběžně testovat?“, tady se bude testovat pořád.

Agilní metodika bývá velmi často zaměňována za agilní techniku, ale ve skutečnosti se jedná o oblasti s rozdílnými významy. Agilní metodika se zaměřuje na organizaci práce, bez ohledu na to, jestli je vyvíjen software nebo ne. Agilní metodika umožňuje změny, kdežto agilní technika ne. Agilní technika je konkrétní postup, který se týká vývojářů.

Tato metodika umožňuje kvalitní vývoj softwaru podle vybraných předností, které nás v dané situaci právě zajímají.

3 Automatická dokumentace

Automatická dokumentace není nic jiného, než běžné komentáře, které jsou psány, místo za standardními znaky pro běžné komentování pro daný jazyk, do speciálních tagů, jako je to například u HTML. Díky tomu je pak možné, za pomocí externího programu, přečíst a vyhotovit z tagů dokumentační soubor, například v PDF nebo jiném podporovaném formátu.

Název tagu	Popis
c	Text v tomto tagu indikuje zdrojový kód.
code	Podobný jako tag c, ale určuje víceřádkový kód.
example	Popis příkladu.
exception	Odkaz na výjimku.
include	Díky tomuto tagu lze mít dokumentaci v externím souboru.
list	Umožňuje vytvořit seznam.
para	Vkládá text do odstavce.
param	Popis parametru.
paramdef	Odkaz na parametr.
permission	Popis přístupnosti metody.
remarks	Podrobnější popis třídy/metody.
returns	Popis návratové hodnoty.
see	Odkaz na jiný element.
seealso	Specifikuje text, který bude zobrazen v sekci See also.
summary	Krátký popis třídy/metody.
value	Popis vlastnosti třídy.

Obrázek 10 Seznam tagů [16]

```
/// <summary>
/// metoda slouzi k dekompressi vstupnich data
/// </summary>
/// <param name="data">zakomprimovana vstupni data</param>
/// <returns>pole listu obsahujici dekomprimovana data</returns>
```

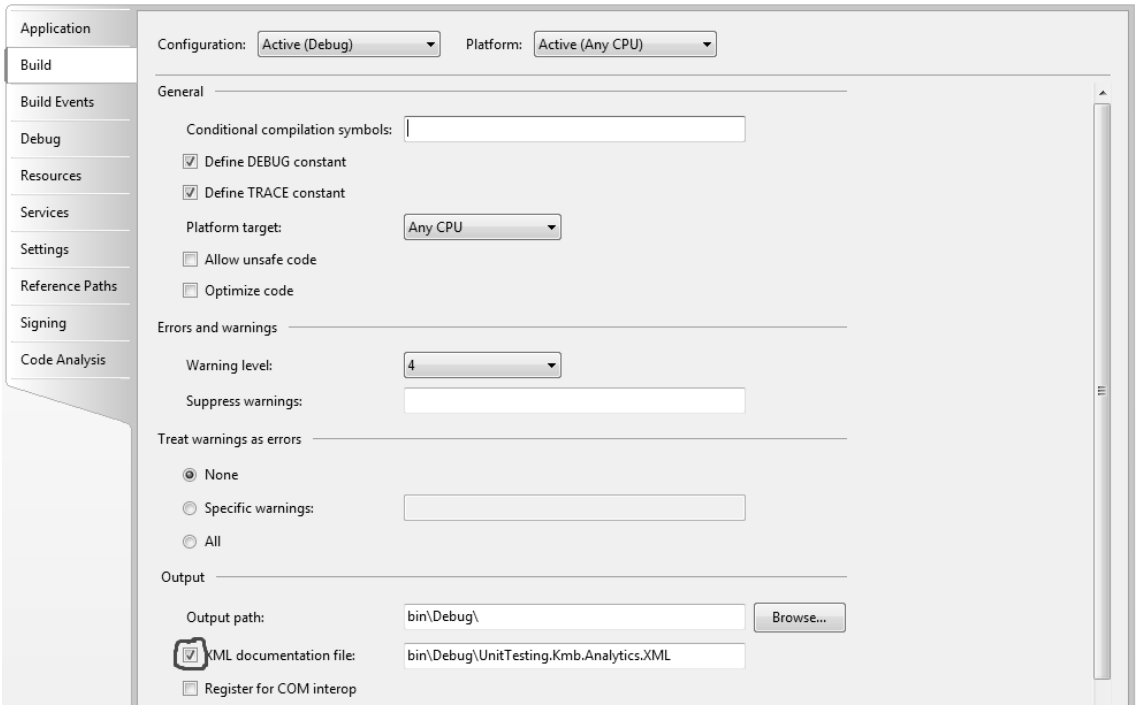
Obrázek 11 Ukázka použití

3.1 Sandcastle

Sandcastle patří asi mezi nejznámější programy pro vytvoření dokumentace z dokumentačních komentářů. Generátor je vytvořen firmou Microsoft, tudíž běží pod platformou .NET. Nahradil zastaralí NDoc, který přestal být dále vyvíjen. Ukládá dokumentaci ve formátu CHM, což je přesný formát stylu MSDN. Dokument je vytvořen z DLL a XML souborů. Přitom soubor XML vznikne vygenerováním vývojového prostředí a obsahuje pouze obsah a hierarchii dokumentačních komentářů.

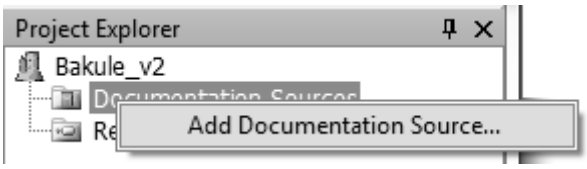
3.1.1 Návod pro Sandcastle za použití VS 2008

Jako první věc je potřeba nainstalovat do svého PC Sandcastle a Sandcastle Help File Builder, na internetu jsou volně ke stažení a obsahuje je i přílohové CD práce, na kterém jsou, přidány všechny potřebné a zmiňované aplikace. Instalace je jednoduchá a nepotřebuje tedy žádný komentář. Ve Visual Studiu nejprve musíme povolit vytváření XML souboru, který bude vytvořen z dokumentačních komentářů a je potřebný pro generování dokumentace. Klikneme pravým tlačítkem myši na daný projekt a vybereme vlastnosti, pak v záložce Build zaškrtneme políčko XML documentation file. Poté stačí už jen znova přeložit projekt a XML soubor se vytvoří.

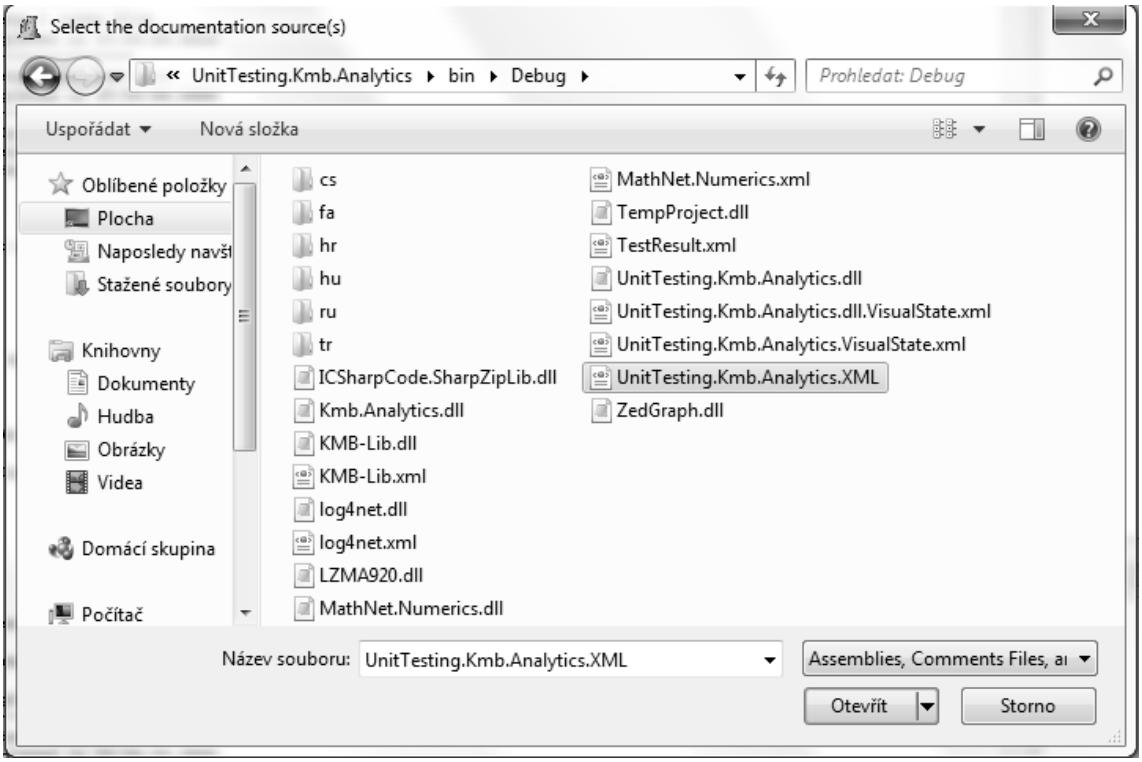


Obrázek 12 Povolení vytvoření XML souboru

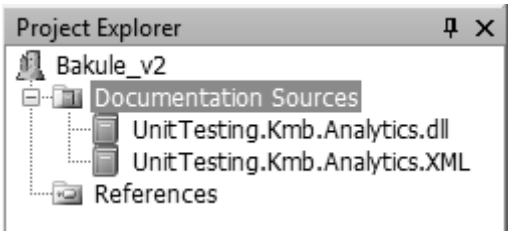
Když už máme konečně vytvořený XML soubor, spustíme Sandcastle Help File Builder přes SandcastleBuilderGUI.exe. V něm si velice jednoduše vytvoříme nový projekt. V pravém sloupci pak přes pravé tlačítko myši do položky Documentation Sources přidáme vytvořený XML soubor a automaticky se k němu přihodí i DLL soubor. Pokud se nenachází na stejném místě, je potřeba ho přidat ručně úplně stejně.



Obrázek 13 Přidání XML a DLL souboru



Obrázek 14 Výběr XML souboru



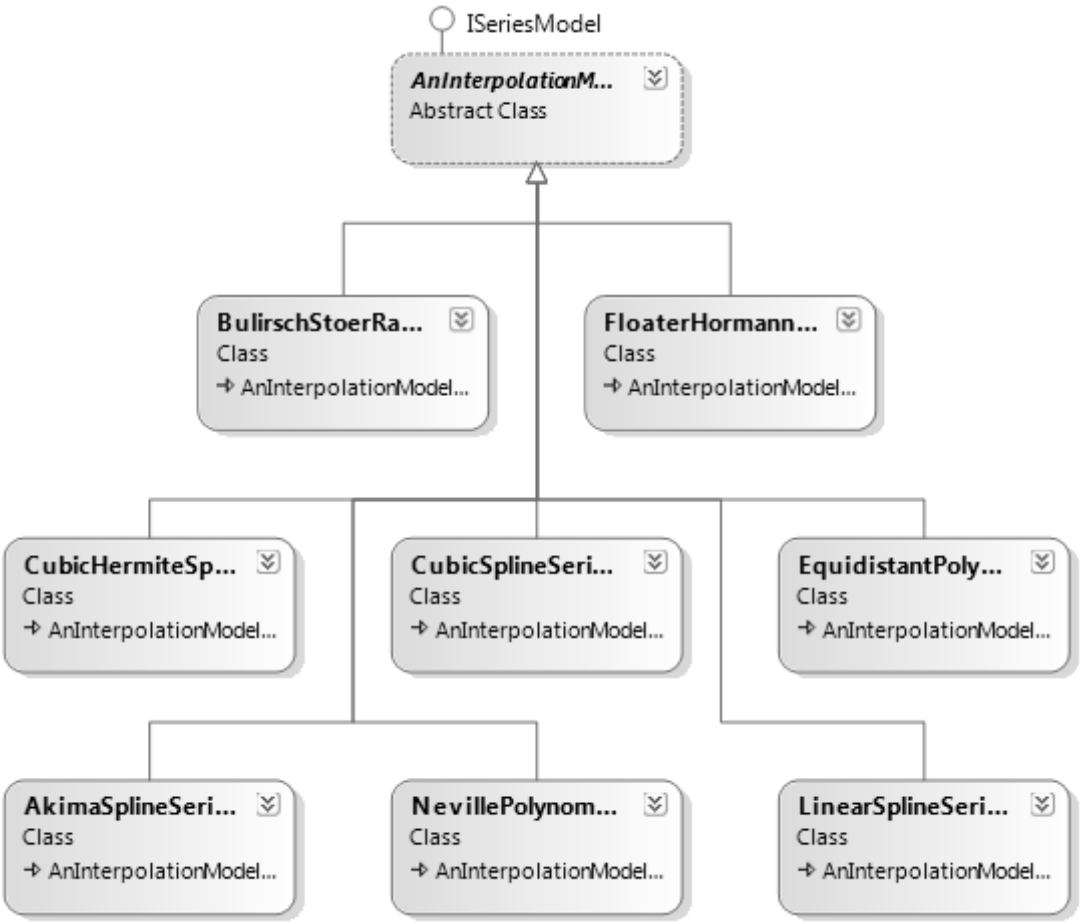
Obrázek 15 Vložené XML a DLL jako zdroje

Když už nakonec máme přidané všechny zdrojové soubory, spustíme vytvoření dokumentace přes tlačítko Build Help File nebo přes položku Build Project v záložce Documentation. Takto vytvoření soubor lze pak snadno převést do PDF formátu například programem CHM2PDF, jeho ovládání je velice intuitivní a nepotřebuje tudíž žádný komentář.

4 KMB.Analytics

Knihovna, jejímž úkolem je předpřipravit data pro budoucí grafické zobrazení. Obsahuje tři namespace, kde se každý stará o danou problematiku. Containers obstarává uložení dat, převod do a z pole bajtů, vytváření vlastních datových struktur, atd. Modelling se stará o vytváření modelů dat na logické úrovni, které se pak budou zobrazovat. Processors provádí s daty různé matematické a jiné úpravy, nebo zjišťuje různé informace, jako je např. histogram. Detailnější a bližší informace jsou uvedeny v podkapitolách.

Obrázek 16 Ukázka struktury



Obrázek 17 Ukázka class diagramu namespace Modelling

4.1 Slabá místa

Protože knihovna provádí různé datové úpravy, korekce, zaokrouhlování, atd., obsahuje zdrojový kód spousty slabých míst. Pro malou ukázku si zde pár příkladů slabých míst představíme a rozebereme si, kde a kvůli čemu můžou vznikat chyby.

Asi mezi nejzákladnější chyby patří neošetření vstupních dat, které metoda dostává. Obrázek 5 předvádí takovou to chybu na přehledném přikladu a je pěkně vidět, v čem může nastat problém. Když se budou zadávat validní data je všechno „OK“, ale v případě záporných hodnot, metoda vyhodí výjimku, která pokud není nikde odchycena, způsobí pád aplikace.

```
public static int[] vytvorPole(int _velikostPole)
{
    return new int[_velikostPole];
}
```

Obrázek 18 Ukázka

Když už nakonec metoda obsahuje podmínku pro kontrolu vstupních dat, je „natvrdo“ napsaná a kontroluje pouze jeden stav. Obrázek 6 demonstruje na ukázce zmiňovaný problém. Máme dvourozměrné pole a úkolem metody je v zadané matici, vynulovat sloupec, určený indexem. Jako podmínka je pouze test prvního řádku matice, jestli má požadovaný počet prvků na řádku, ale už se nestará, zda mají tento počet i ostatní řádky.

```
public static int[][] vynulujSloupec(int[][] _pole, int _index)
{
    if (_pole[0].Length -1 >= _index)
    {
        for (int i = 0; i < _pole[i]; i++)
            _pole[i][_index] = 0;
    }

    return _pole;
}
```

Obrázek 19 Ukázka

Těmito chybami je knihovna plná a mají za následek vyhazování výjimek, které můžou shodit aplikaci. Samozřejmě se pak otvírá diskuze, kdy vadí, že vstupní data nejsou testována nebo jen pouze na konkrétní hodnoty. Pokud si totiž například vývojáři vyvíjí software sami pro sebe, kde ho potřebí k další práci, není potřeba vstupní data

kontrolovat, protože zcela určitě budou zadávat validní vstupní data. Jejich úkolem určitě nebude způsobit pád aplikace. V takovém případě by doplnění několika podmínek způsobilo nárůst časové náročnosti. Ta by měla být na nejnižší úrovni, co nejmenší. Pokud se však software vyvíjí pro běžného uživatele, volí se strategie vývoje kódu a ta se už o tuhle problematiku stará po svém.

Knihovna a její namespace Kmb.Analytics.Modelling vytváří model na základě zpracování dat. Provádí s nimi různé úpravy, dokud nevznikne daný model. Ten se pak ukládá do vlastních datových struktur. Pokud se tedy uloží špatné hodnoty je výsledek nehodnověrný a k ničemu.

Mezi používané úpravy patřící ke slabým článkům je například zaokrouhlování, kde výsledek bývá velmi variabilní a dynamický, protože máme několik možných variant (nahoru, dolu, normální). Dále pak vytváření intervalů z určité délky pole, kde se nám můžou intervaly překrývat. Chyby velmi často vznikají, jakmile se někdy pracuje s desetinným číslem a jeho převodem do celého neznaménkového tvaru. Slabých míst může být samozřejmě víc, já sem zde zvolil pro ukázku ty nejdůležitější a nejzákladnější a hlavně ty, na které sem sám narazil.

4.2 Testy

Při testování zadané knihovny jsem si zvolil vlastní styl práce, který se trochu liší od standardních postupů a to z několika důvodů. Přidělená knihovna byla už, dá se říct, ve finální fázi vývoje. A moje práce byla nezávislá na vývojářském týmu, tudíž mnou nalezené chyby, pomocí unit testů, nikdo neopravoval. Snahou bylo zjistit, jestli knihovna obsahuje chyby a jestli testování má smysl. Proto zde není použit žádný výše zmiňovaný model jako TDD, waterfall, atd., který vyžadují určitý cyklus testování spojený s opravami chybného kódu.

Z těchto důvodů jsou také testy především zaměřeny na funkčnost knihovny a testují její vybrané pasáže a jiné potřebné části, jež mohou obsahovat zásadní chyby. Kdybych měl říci, která sada testů se svými vlastnostmi nejvíce přibližuje unit testům, byla by ta, jež se zabývá namespace Kmb.Analytics.Containers.

4.2.1 Namespace Kmb.Analytics.Containers

Namespace využívá jako hlavní datový typ IList<double> nebo IList<double>[], list nemusí být datovým typem double, ale object. Na prvním řádku bývají časové hodnoty a na zbylých už naměřené hodnoty různých fyzikálních veličin. Nachází se v něm důležitý c sharp soubor IListArrayHelpers.cs, obsahující třídy a metody obsluhující vlastní datový typ knihovny. Obstarává jeho vytváření, kopírování (klonování), dokáže z něho vytvořit podmnožinu. Umožňuje odstraňování dat nebo je podle zadaného filtru vyfiltrovat, převádět na pole bajtů, atd.

Dále obsahuje soubor SeriesContainer.cs, kde se nachází třída SeriesContainer. Ta slouží jako uložiště, pro model, původní data, popis hodnot na řádku, atd. Je tedy složitější datovou strukturou. Umí ukládat nejen původní data, ale obsahuje i data modelu, typ modelu, rozdíl modelovaných a původních hodnot, atd.

Níže se budu zabývat rozborem následujících metod: NewListArray, Clone, Subset, RemoveAt, Filter, DifferentialEncodingDouble, FromByteArray.

NewListArray

Metoda pro vytváření pole listů různých datových typů. Testování funkčnosti zde představovalo vytvořit testy typu, jestli vrácené pole mělo délky požadované velikosti. Testy parametrů poté zjišťovaly reakci metody na různé vstupy, jako jsou: kladná, nulová a záporná velikost pole.

Clone

Metoda pro vytvoření kopie pole listů. Testování funkčnosti zde představovalo vytvořit testy typu, jestli vrácené pole má stejné hodnoty jako vstupní pole. Hlavně také, aby žádný z prvků neukazoval na stejné místo v paměti, tzn., nešlo o mělkou kopii.

Subset

Metoda pro vytvoření podmnožiny dat ze zadaného vstupního pole listů. Testování funkčnosti zde představovalo vytvořit testy typu, jestli vrácený objekt obsahuje požadovaná data. Testy parametrů poté zjišťovaly reakci metody na různé vstupy, jako jsou: podmnožina větší než samotná množina dat, kladná, nulová a záporná

velikost pole, různý počet prvků na řádku, vstupní objekt bez hodnot nebo pouze jen s časovými údaji, atd.

RemoveAt

Metoda pro odstranění zvoleného sloupce z pole listů. Testování funkčnosti zde představovalo vytvořit testy typu, jestli vrácený objekt obsahuje požadovaná data, už bez určeného sloupce. Testy parametrů poté zjišťovaly reakci metody na různé vstupy, jako jsou: kladné, nulové a záporné vstupní hodnoty a také reakci metody na prázdný objekt.

Filter

Metoda pro odfiltrování zvolených dat z pole listů. Metoda dostane na vstup pole listů a dva listy. Jeden popisuje aktuální obsah pole, druhý požadované položky ve výsledku. Testování funkčnosti zde představovalo vytvořit testy typu, jestli vrácený objekt obsahuje požadovaná data. Testy parametrů poté zjišťovaly reakci metody na různé vstupy, jako jsou: různé délky listů popisující obsah, prázdné pole listů.

DifferentialEncodingDouble

Metoda slouží pro zmenšení amplitudy pole listů, slouží například k lepší kompresi dat. Testování funkčnosti zde představovalo vytvořit testy typu, jestli vrácený objekt po opětovné zavolání metody, obsahuje původní data a je tedy bezztrátový. Testy parametrů poté zjišťovaly reakci metody na různé vstupy, jako jsou: prázdný objekt nebo objekt bez hodnot.

FromByteArray

Metoda pro převedení pole bajtů do datové struktury listu. Testování funkčnosti zde představovalo vytvořit testy typu, jestli výstupní objekt obsahuje správná data. Testy parametrů poté zjišťovaly reakci metody na různé vstupy, jako jsou: různě dlouhá vstupní pole.

4.2.2 Namespace Kmb.Analytics.Modelling

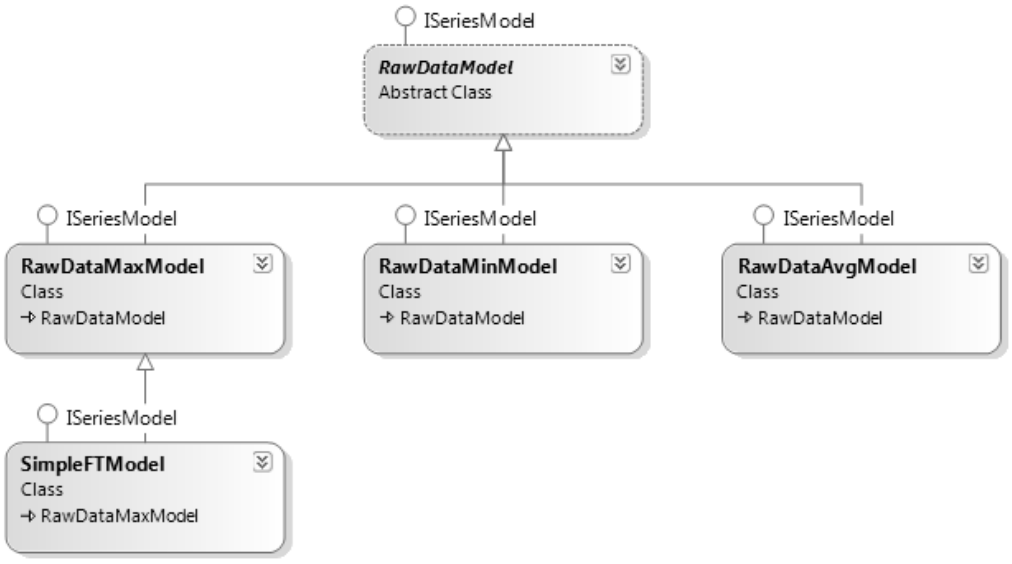
Tento prostor obsahuje celý seznam modelů, kde každý model podle svých vlastností reprezentuje data po svém. Jejich úkolem je logicky zpracovat data pro budoucí zobrazení grafickou vrstvou. Základním datovým typem modelů je pole listů typu double (IList<double>[]). Princip je celkem jednoduchý, model obdrží na vstup hodnoty, ten je zpracuje, a uloží si je. Dokáže však taky zpětně získat původní hodnoty.

Většina napsaných testů se zabývá problematikou právě zmiňovaného jmenného prostoru. Namespace obsahuje jednu důležitou statickou třídu, která mi při práci velice pomohla. Jmenuje se ModelFactory a obsahuje pro mě ještě důležitější metodu Create, která obstarává vytvoření jakéhokoliv zadaného modelu.

Protože modelů je hodně začal jsem psaním testů společných pro všechny modely, abych vyřadil ty, které mají chyby už v základu, a tím pádem budou mít špatný i výsledek. Takovým důležitým testem je test, který kontroluje správný počet prvků na řádku a správný počet řádků.

Dalším společným testem je test, který kontroluje, jestli sám model nebo jeho metoda (GetEstimatedValue), pro zpětnou rekonstrukci dat, obsahuje nebo vrací nežádoucí hodnoty NaN a INF. Tyto hodnoty by znehodnotily celý graf a to z důvodu, který již byl řečen (model slouží jako logická vrstva pro budoucí zobrazení).

RawDataModel.cs



Obrázek 20 Class diagram

Obsahuje abstraktní třídu RawDataModel a klasické třídy RawDataMaxModel, RawDataMinModel, RawDataAvgModel, které od ní dědí. Na tyto třídy jsem se víc zaměřil a přizpůsobil testy jejím potřebám. Pomocí testů zjišťuji nejen již výše zmiňované informace, ale dále také následující.

Testuji, jestli se data v modelu budou opakovat, pokud mají na vstupu neopakující se hodnoty. Délka intervalů, ze kterých se data vybírají, by se neměly lišit víc jak o jedna, dochází tím k nerovnoměrnému výběru dat. Testy funkčnosti zjišťují, jestli se ukládají správná data a jestli model při zavolání metody GetEstimatedValue vrací správná data. Testuji reakce modelu na různé vstupní parametry. Zjišťuji velikost šumu, to jsou originální data mínus model data.

4.2.3 Komprese a dekomprese

V této části knihovny se testy zaobírají hlavně funkčností, kde tuto vlastnost dokazují hned pro několik typů komprese a dekomprese. Patří mezi ně LZMA, ZIP, BZIP.

4.3 Výsledky testů

Celkem bylo napsáno 80 unit testů, kde 39 z nich prošlo testem úspěšně, a zbytek selhal, z toho vyplívá, že více jak půlka testů neprošla.

4.3.1 Namespace Kmb.Analytics.Containers

Tato část knihovny prošla testy nejlépe, a to z toho důvodu, že všechny testy, zabývající se funkčností metod, prošly. Všechny kromě jedné, kterou je metoda DifferentialEncodingDouble, jež nesplňovala podmínku, kdy při jejím dvojím zavolání, musí vrátit původní data. Metodě FromByteArray se zas nelíbilo, když na vstup dostala pole, jehož délka nebyla násobkem osmi. Selhala pouze většina testů zabývajících se testováním vstupních parametrů. A to jen kvůli tomu, že zde chyběli podmínky pro defenzivní programování, tzn. podmínky pro kontrolu vstupních dat.

4.3.2 Namespace Kmb.Analytics.Modelling

U modelů jsem začal společným testem, kontrolující počet řádků a počet hodnot na řádku, který mi ukázal celý seznam modelů, jež nesplňují mnou zadané vstupní parametry, ve výstupu se objevili i jiné chyby než jsem očekával.

Tabulka 1Výsledky testů

Špatný počet		Jiná chyba
řádků	**hodnot na řádku**	**System.NullReferenceException**
AkimaSpline	Fourier	EquidistantPolynomial
CubicSpline	BasicModel	SandboxCalculating
CubicHermiteSpline	AkimaSpline	
LinearSpline	CubicSpline	
NevillePolynomial	CubicHermiteSpline	
BulirschStoerRational	LinearSpline	
FloaterHormannRational	NevillePolynomial	
	BulirschStoerRational	
	FloaterHormannRational	

Dalším testem byl test zjišťující výskyt nežádocucích hodnot NaN a INF v modelu, pro jeho budocí grafické zobrazení. Samozřejmě zde už nebudu uvádět modely, které vyhazovaly jinou chybu než jsem čekal, ale stejnou jako v minulých testech. Je jasné, že když jsem se na ně nemohl dostat už předtím, tak se na ně nedostanu ani teď. Z modelů, které obsahují ve svých datech výše zmiňované hodnoty patří pouze model Fourier. Následující test byl velice podobný minulému, s tím rozdílem, že tento test zjišťuje výskyt nežádocucích hodnot NaN a INF v návrátovém objektu metody GetEstimatedValue. Samotný model totiž tyto hodnoty obsahovat nemusí, ale může je vracet jeho metoda, která odhaduje původní data.

Tabulka 2 Výsledky testů

Nežádoucí hodnoty		Jiná chyba
INF	**NaN**	**Index je mimo hranice**
AkimaSpline	FloaterHormannRational	Averaging
CubicSpline		
Fourier		
CubicHermiteSpline		
LinearSpline		
NevillePolynomial		
BulirschStoerRational		

RawDataModel a jejich potomky

Model RawDataModel a jeho potomky byly podrobeny detailnějším testů a jejich výsledky ukázaly nefunkční části modelů. Z předchozích výsledků testů je vidět, že model sám ani jeho metoda GetEstimatedValue neobsahuje nežádoucí hodnoty NaN a INF. Dále metoda úspěšně prošla v testech, které zjišťovaly správnost návratového objektu na různé hodnoty vstupu. To všechno platí, pokud se bavíme pouze o samotné metodě, která čte data z modelu.

Pokud se zaměříme na samotný model, který má dva konstruktory. Tak jestliže použijeme první, jež má vstupním parametrem pouze data. Neobsahuje model žádnou chybu a data jsou uložena správně pro daný typ modelu. Při použití druhého konstruktoru, kde ke vstupním datům, přidáváme ještě parametr počet intervalů, se objevují v modelu chyby.

Pokud je zbytek po dělení počtu prvků na řádku počtem intervalů roven nule (ModelData[0].Count % NumberOfIntervals = 0), jsou data do modelu uložena správně. V opačném případě nastává chybný zápis dat. A to z toho důvodu, že vnitřní metoda GetRangeReal, jež počítá a vytváří délku intervalů, špatně zaokrouhluje. Tím pádem nastává případ, kdy do intervalů zanáší data, která už tam jednou byla. Dokazují to testy, které testují opakování dat při neopakujících se vstupních datech a testy, které počítají délku intervalu. Když je zadán interval, tak i metoda GetEstimatedValue vrací špatné hodnoty modelu pro původní data. A to všechno kvůli tomu, že metoda BinarySearch neumí najít správné hodnoty.

4.3.3 Komprese a dekomprese

Výsledky testů zabývající se funkčností kompresních a dekompresních metod prošly pro třídy LZMA a ZIP v pořádku. Byly schopny zabalit data a rozbalit je bez jakékoliv ztráty. Naproti tomu třída BZIP, kterou jsem psal sám, dokázala zabalit pole bajtů a rozbalit je bezztrátově, ale její rozšíření, kterým je třeba komprese a dekomprese listu, už testem neprošlo. A to jen z důvodu, že metoda FromByteArray vyhazuje chybu nedostatečné délky pole. Nejspíš za to může metoda ToByteArray, která špatně převede list na pole bajtů a potom už metoda FromByteArray nedokáže správně pracovat a převést pole bajtů zpět na list.

4.4 Vlastní kódy

Moje práce na knihovně nekončila pouze jejím otestováním, ale opravením několika jejich chybných metod a doplněním i chybějícího kódu a jeho následného přezkoušení. Knihovnu KMB.Analytics jsem obohatil o metodu GetRangeRealMOJE, která opravuje stávající výše zmíněnou metodu a umožňuje správné vytváření intervalu a tím i ukládání správných hodnot modelů.

Dalším mým přírůstkem je metoda DifferentialEncodingDoubleMOJE, která opravuje funkčnost stávající. Umožňuje tím při jejím dvojím zavolání dostat původní data. Práce na třídě LZMACompressor bylo doplnění její dekompresní metody o různé vstupní parametry. U ZIPCompressor jsem vytvořil kompletně celou dekompresní část třídy. A nakonec jsem napsal celou třídu BZIPCompressor, tzn. s kompletní kompresní a dekompresní částí. Všechny moje doplněné kódy byly řádně otestovány unit testy na správnou funkčnost a poslouží jako náhrada za stávající.

4.5 Dokumentace

Jako dokumentační část práce byly vytvořeny dokumentační komentáře u všech mých vlastních kódů. Všechny testy obsahují tyto komentáře také a díky tomu je z nich možno za pomocí programu Sandcastle vytvořit dokumentaci.

Závěr

V kapitole testování jsem vysvětlil, co to vlastně testování je, k čemu je dobré a kdo ho provádí, jaká jsou jeho omezení. Na obrázcích jsem zobrazil nejčastější výskyt chyb a jejich náklady na nalezení a odstranění. Dále jsem pak testy rozdělil do několika skupin a popsal je. Nakonec jsem vybral jeden konkrétní typ testu, který byl zadán a provedl jeho detailnější analýzu v kapitole Unit testy. Dále vysvětluji, čím se tento typ testu zabývá, jaké jsou jeho výhody, nevýhody a jaká jsou pravidla, která musí dodržovat, aby splňoval podmínky své definice. V kapitole nástroje je pak mnou několik nástrojů představeno a porovnáno.

V kapitole strategie vývoje kódu jsem se zabýval, jaké jsou postupy při vytváření kódu a jakým způsobem snižují chybovost. Bylo jich hned několik. Ofenzivní a defenzivní programování snižuje výskyt chyb kontrolou vstupů a ošetřováním výjimek. TDD, vodopád nebo agilní metodiky se zabývají zase stylem a metodou vývoje kódu. V kapitole automatické dokumentace jsem vysvětlil, k čemu se používá, představil nástroj Sandcastle a vytvořil ukázky její tvorby.

Hlavní část se zabývá testováním zvolené knihovny. Jako první úkol bylo seznámení se samotnou knihovnou. Nejdříve bylo potřeba pochopit základní principy chování a používání. Když jsem pochopil, jak knihovna pracuje, mohl jsem se zaměřit na její slabá místa a otestovat ji detailněji. V celé knihovně se vyskytuje spousta neošetřených vstupů. Jejich ukázky jsou uvedeny v kapitole slabá místa. Otázkou je zdali se mají všechna vstupní data testovat a zanášet tak do kódu zpomalující podmínky. Pokud totiž budeme vyvíjet například engine, tak prioritou bude rychlost. Pokud však budeme vyvíjet software pro běžného zákazníka, není možné očekávat zadávání validních dat. Proto se používá strategie vývoje kódu. V kapitole testy jsem popsal, že knihovna byla především testována na funkčnost z důvodu, který jsem před chvílí jmenoval. Ale i tak bylo nutné testovat knihovnu na různá vstupní data, aby se zaručila správná funkcionalita. V této kapitole je také uvedeno, co daná část knihovny dělá a jak by se měla testovat, na jaké hodnoty a situaci. Příkladem může být test, kde se na vstupu data nebudou opakovat a tím pádem se nesmí opakovat ani ve výsledném modelu.

Výsledky testů prokázaly, že knihovna obsahuje velký počet chyb nebo alespoň při nejmenším nedodělané části. Příkladem může být počet řádků. Ten mohl být původně jiný, časem se však změnil a změna ještě neprošla celou knihovnou, pouze její částí. Příspěvky mojí práce na knihovně jsou uvedeny v kapitole vlastní práce. Popisuji v ní, čím vším jsem přispěl a že celá moje část, tzn. testy i vlastní kódy, jsou obaleny dokumentačním komentářem. Pokud bude potřeba, lze z nich vytvořit dokumentaci. Na přiloženém CD je uložena dokumentace ve formátu PDF, zabývající se testovací částí.

Do kapitoly příloha jsem chtěl vložit vytvořenou dokumentaci, ale z důvodu velkého počtu stran, je celá dokumentace uložena pouze na přiloženém CD spolu s dalšími souboru týkající se bakalářské práce. Pro ukázku je v příloze jedna stránka z celého dokumentu.

Literatura

[1] Weinberg, Gerald. M.: *Quality Software Management: Volume 1, Systems Thinking,* Dorset House Publishing Company, Incorporated, 1991

[2] *Http://en.wikipedia.org* [online]. 21 December 2010 [cit. 2011-05-16]. Test-driven development. Dostupné z WWW: <http://en.wikipedia.org/wiki/File:Test-driven_development.PNG>.

[3] *Http://cs.wikipedia.org/* [online]. 24. 6. 2010 [cit. 2011-05-16]. Vodopadovy model. Dostupné z WWW: <http://cs.wikipedia.org/wiki/Soubor:Vodopadovy_model.png>.

[4] *Http://zdrojak.root.cz* [online]. 11. 6. 2010 [cit. 2011-05-16]. Začít testovat je jednoduché. Dostupné z WWW: <http://zdrojak.root.cz/clanky/zacit-testovat-je-jednoduche/>.

[5] *Http://zdrojak.root.cz/* [online]. 14. 3. 2011 [cit. 2011-05-16]. Ještě k testování. Dostupné z WWW: <http://zdrojak.root.cz/clanky/jeste-k-testovani/>.

[6] *Testování webových aplikací : Základy testování* [online]. [s.l.] : [s.n.], 13.11.2006 [cit. 2011-05-16]. Dostupné z WWW: <http://www.poeta.cz/Zaklady_testovani.pdf>.

[7] *Http://zdrojak.root.cz* [online]. 11. 5. 2010 [cit. 2011-05-16]. Testování není nástroj, ale metoda vývoje. Dostupné z WWW: <http://zdrojak.root.cz/clanky/testovani-neni-nastroj-ale-metoda-vyvoje/>.

[8] *Http://zdrojak.root.cz/* [online]. 18. 8. 2010 [cit. 2011-05-16]. Vývoj řízený testováním: tutoriál pro nováčky. Dostupné z WWW: <http://zdrojak.root.cz/zpravicky/vyvoj-rizeny-testovanim-tutorial-pro-novacky/>.

[9] *Http://cs.wikipedia.org* [online]. 17. 4. 2011 [cit. 2011-05-16]. Testování softwaru. Dostupné z WWW: <http://cs.wikipedia.org/wiki/Softwarov%C3%A9_testov%C3%A1n%C3%AD>.

[10] *Http://testovanisoftwaru.blogspot.com/* [online]. 18. dubna 2011 [cit. 2011-05-16]. Testování softwaru . Dostupné z WWW: <http://testovanisoftwaru.blogspot.com/>.

[11] *Http://en.wikipedia.org/* [online]. 13 May 2011 [cit. 2011-05-16]. Test-driven development. Dostupné z WWW: <http://en.wikipedia.org/wiki/Test_driven_development>.

[12] *Http://www.fi.muni.cz/* [online]. 12. ledna 2006 [cit. 2011-05-16]. Test-Driven Development — programování řízené testy. Dostupné z WWW: <http://www.fi.muni.cz/usr/jkucera/pv109/2005/xvlcek1.htm>.

[13] *Http://cs.wikipedia.org* [online]. 12. 4. 2011 [cit. 2011-05-16]. Vodopádový model. Dostupné z WWW: <http://cs.wikipedia.org/wiki/Vodop%C3%A1dov%C3%BD_model>.

[14] *Http://zdrojak.root.cz/* [online]. 11. 12. 2009 [cit. 2011-05-16]. Agilní vývoj: Úvod. Dostupné z WWW: <http://zdrojak.root.cz/clanky/agilni-vyvoj-uvod/>.

[15] *Http://blog.wuwej.net* [online]. 27.05.2005 [cit. 2011-05-16]. Agilní programování. Dostupné z WWW: <http://blog.wuwej.net/2005/05/27/vaclav-kadlec-agilni-programovani.html>.

[16] *Http://www.rydval.cz* [online]. 03. 08. 2005 [cit. 2011-05-16]. Generování dokumentace pro C#. Dostupné z WWW: <http://www.rydval.cz/phprs/view.php?cisloclanku=2005123138>.

[17] *Http://www.vbnet.cz/* [online]. 7. 8. 2009 [cit. 2011-05-17]. Základy testování aplikací pomocí Visual Studia. Dostupné z WWW: <http://www.vbnet.cz/clanek--133-zaklady_testovani_aplikaci_pomoci_visual_studia.aspx>.

[18] *Http://d3s.mff.cuni.cz/* [online]. 9. září 2010 [cit. 2011-05-16]. Defenzivní programování. Dostupné z WWW: <http://d3s.mff.cuni.cz/teaching/programming_practices/lecture10.html>.

[19] *Http://www.dreamincode.net* [online]. 08 June 2009 [cit. 2011-05-16]. C# Unit Testing Basics. Dostupné z WWW: <http://www.dreamincode.net/forums/topic/108976-c%23-unit-testing-basics/>.

[20] *Http://msdn.microsoft.com/* [online]. March 2005 [cit. 2011-05-16]. A Unit Testing Walkthrough with Visual Studio Team Test. Dostupné z WWW: <http://msdn.microsoft.com/en-us/library/ms379625%28VS.80%29.aspx>.

[21] *Agilní metodiky vývoje software* [online]. Brno : [s.n.], květen 2005 [cit. 2011-05-16]. Dostupné z WWW: <http://is.muni.cz/th/39440/fi_m/dp.pdf>.

[22] *Http://www.exubero.com/* [online]. 2009-02-06 [cit. 2011-05-16]. JUnit Anti-patterns. Dostupné z WWW: <http://www.exubero.com/junit/antipatterns.html>.

[23] *Http://blogs.msdn.com* [online]. 29 Jul 2006 [cit. 2011-05-16]. Creating a Chm build using Sandcastle. Dostupné z WWW: <http://blogs.msdn.com/b/sandcastle/archive/2006/07/29/682398.aspx>.

[24] *Http://www.chm-to-pdf.com/* [online]. 19. února 2011 [cit. 2011-05-16]. Convert CHM to PDF. Dostupné z WWW: <http://www.chm-to-pdf.com/>.

[25] Christian Nagel: C# 2008 Programujeme profesionálne. Computer Press, Brno, 2009. ISBN: 978-80-251-2401-7

[26] John Sharp: Microsoft Visual C# 2008 Krok za krokem. Computer Press, Brno, 2008. ISBN: 978-80-251-2027-9

[27] Virius M.: C# - Hotová rešení. Computer Press, Brno, 2006, ISBN: 80-251-1084-2

[28] Application Architecture Guide 2.0 [online]. Dostupné z WWW: $<$http://apparchguide.codeplex.com/$>$.

Příloha

CompresorUT Members

CompresorUT Class Constructors Methods Properties See Also Send Feedback

The CompresorUT type exposes the following members.

- Constructors

	Name	Description
	CompresorUT	Initializes a new instance of the CompresorUT class

- Methods

	Name	Description
	CompressBZIPTestFunkcnost V00	BZIPCompressor::Compress(List(double) data) - test jestli po compresy a následné dekompresy bude data stejná
	CompressBZIPTestFunkcnost V01	BZIPCompressor::Compress(List(double)[] data) - test jestli po kompresy a následné dekompresy bude data stejná
	CompressBZIPTestFunkcnost V02	BZIPCompressor::Compress(byte[][] data, string NodeName) - test jestli po kompresy a následné dekompresy bude data stejná
	CompressBZIPTestParametruV_0	BZIPCompressor::Compress(List(double)[] data) - test reakce metody na objekt bez hodnot
	CompressLZMATestFunkcnostiV00	LZMACompressor::Compress(List(double) data) - test jestli po kompresy a následné dekompresy bude data stejná
	CompressLZMATestFunkcnostiV01	LZMACompressor::Compress(List(double)[] data) - test jestli po kompresy a následné dekompresy bude data stejná
	CompressLZMATestFunkcnostiV02	LZMACompressor::Compress(byte[][] data, string ModelName) - test jestli po kompresy a následné dekompresy bude data stejná
	CompressLZMATestParametruV_0	LZMACompressor::Compress(List(double)[] data) - test reakce metody na objekt bez hodnot
	CompressZIPTestFunkcnost V00	ZIPCompressor::Compress(List(double) data) - test jestli po compresy a následné dekompresy bude data stejná
	CompressZIPTestFunkcnost V01	CompressZIP::Compress(List(double)^ data) - test jestli po kompresy a následné dekompresy bude data stejná
	CompressZIPTestFunkcnost V02	CompressZIP::Compress(byte^[]^ data, string ModelName) - test jestli po kompresy a následné dekompresy bude data stejná
	CompressZIPTestParametruV_0	CompressZIP::Compress(List(double)^ data) - test reakce metody na objekt bez hodnot
	Equals	Determines whether the specified Object is equal to the current Object. (Inherited from Object.)
	Finalize	Allows an Object to attempt to free resources and perform other cleanup operations before the Object is reclaimed by garbage collection. (Inherited from Object.)
	GetHashCode	Serves as a hash function for a particular type. (Inherited from Object.)
	GetType	Gets the Type of the current instance. (Inherited from Object.)
	MemberwiseClone	Creates a shallow copy of the current Object. (Inherited from Object.)
	ToString	Returns a String that represents the current Object. (Inherited from Object.)

- Properties

	Name	Description
	TestContext	Gets or sets the test context which provides information about and functionality for the current test run.

- See Also

CompresorUT Class

Microsoft.VisualStudio.TestTools.UnitTesting Namespace