

**VYSOKÁ ŠKOLA FINANČNÍ A SPRÁVNÍ, o.p.s.**

Fakulta Ekonomických studií

Studijní obor: Aplikovaná informatika

Navazující magisterské studium kombinované

**Bc. Peter Zošiak**

**Principy a postupy vytváření automatizovaných  
testů v MS Visual Studio 2012**

**Principles and procedures for creating  
automated tests in MS Visual Studio 2012**

**Diplomová práce**

**Praha 2013/2014**

Vedoucí diplomové práce: RNDr. Jan Lánský, Ph.D.

## **Pod'akovanie**

Na tomto mieste by som sa chcela poďakovať vedúcemu diplomovej práce RNDr. Jan Lánský, PhD. za odbornú a metodickú pomoc pri koncipovaní mojej diplomovej práce a za jeho rady a pripomienky, ktoré boli pre mňa cenným prínosom, a Martine Nevolnej za neoceniteľnú pomoc pri štylizácii.

## **Prehlásenie**

### **Prehlasujem,**

že som túto záverečnú prácu vypracoval / a úplne samostatne a všetku použitú literatúru a ďalšie podkladové materiály, ktoré som použil / a, uvádzam v zozname literatúry a že zviazaná a elektronická podoba práce je zhodná. Súčasne prehlasujem, že súhlasím so zverejnením tejto práce podľa § 47b zákona č.111/1998Sb., O vysokých školách a o zmene a doplnení niektorých zákonov (zákon o vysokých školách), v znení neskorších predpisov.

15.04.2014 .....

## **Abstrakt**

Predmetom práce je uvedenie do teórie testovania prostredníctvom definovania základných pojmov, predstavenia základného členenia typológie testov, metód a techník testovania so zameraním na ich automatizáciu ako neoddeliteľnú súčasť pri modernom vývoji systému alebo aplikácie. V praktickej rovine sa práca zameriava na krátke predstavenie nástroju Microsoft Visual Studio 2012 a praktickú tvorbu všetkých podporovaných automatizovaných testov. Práca rieši postupy a princípy tvorby automatizovaných testov v jazyku C# v prostredí Microsoft Visual Studia 2012. Súčasťou štúdie je identifikovanie väčšiny reálnych potrieb a problémov, ktoré môžu pri tvorbe testov nastáť. Cieľom práce je poukázať na významnosť testovania ako na súčasť vývojového procesu a vďaka obsiahlej obrázkovej prílohe poskytnúť praktický návod ako vytvárať, spravovať a interpretovať výsledky automatizovaných testov.

## **Abstract**

The aim of thesis is to provide the introduction to the theory of testing by defining basic terms, classifying the basic typology, methodology and techniques of testing focusing on the automatization as the integral part of the modern system/application development. In its practical part, the thesis concentrates on the short introduction to the Microsoft Visual Studio 2012 tool and on the practical formation of automated tests supported in this tool. The thesis is discussing methods and principles of automated test's formation in the C# language in the Microsoft Visual Studio 2012 environment, as well as most of the real requests and issues that might arise during a test's formation. The purpose of the thesis is to point out the importance of testing as an essential part of the system/application development process and to provide a practical guide on how to create, manage and interpret the results of automated testing by virtue of the ample hooked illustration.

## **Kľúčové slova**

Microsoft, Visual, Studio, automatizovaný, test, tester, testovanie, Web, Performance, Unit, Zátťažový, Ordered, GUI, CodedUI, White Box, Black Box, Funkčné testovanie, Integrované testovanie, Systémové testovanie, Smoke, Regresné testovanie, Akceptačné testovanie, Výkonnostné testovanie, Testovacia stratégia, Testovací prípad, Dátovo riadené, Pokrytie kódu

## **Keywords:**

Microsoft, Visual, Studio, automated, test, tester, testing, Web, Performance, Unit, Load, Ordered, GUI, CodedUI, White Box, Black Box, Functional testing, Integration testing, System testing, Smoke, Regression testing, Acceptance testing, Performance testing, Test strategy, Test case, Data driven, Code coverage

# Obsah

1	DEFINÍCIA POJMOV.....	14
1.1	Chyba.....	14
1.2	Kvalita.....	15
1.3	Ekonomické aspekty.....	16
1.4	Tester.....	19
2.	TYPOLÓGIA TESTOV.....	20
2.1	Manuálne testovanie.....	20
2.2	Automatizované testovanie.....	20
3.	TESTOVACIE METÓDY.....	22
3.1	White Box Testing.....	22
3.2	Black box testovanie.....	23
4.	TESTOVACIE TECHNIKY.....	25
4.1	Functional testing - Funkčné testovanie.....	25
4.1.1	Unit testing - Test Jednotky.....	26
4.1.2	Integration testing - Integračné testy.....	27
4.1.3	System testing - Systémové testovanie.....	30
4.1.4	Smoke testing.....	30
4.1.5	Regression testing - Regresné testovanie.....	31
4.1.6	Acceptance testing - Akceptačné testovanie.....	32
4.1.7	Exploratory testing - Exploračné testovanie.....	33
4.2	Non-Functional Testing - Nefunkčné testovanie.....	34
4.2.1	Performance Testing.....	35
4.2.2	Security Testing.....	37
5.	Testovacia dokumentácia.....	39

5.1	Test Strategy - Testovacia stratégia .....	39
5.2	Test case - Testovací prípad.....	41
5.3	Scenár.....	42
6.	Tvorba testov v Microsoft Visual Studio .....	44
6.1	Tvorba Unit testov .....	44
6.1.1	Data driven testy - Dátovo riadene testy .....	47
6.1.2	Špeciálne typy unit testov .....	51
6.1.3	Expected Exception - Očakávané výnimky .....	53
6.1.4	Pokrytie kódu - Code Coverage .....	54
6.2	Tvorba Ordered testov .....	55
6.3	Tvorba CodedUI Testov.....	57
6.3.1	UIMap.Designer.cs .....	61
6.3.2	UiMap.uitest.....	61
6.3.3	Pridanie Assert .....	62
6.3.4	Data driven CUIT.....	65
6.3.5	Spustenie CUIT z príkazového riadku .....	65
6.4	Tvorba Web Performance Testov .....	67
6.4.1	Tvorba Web Performance Testu .....	68
6.4.2	Web Performance Test Editor.....	72
6.4.3	Panel nástrojov Web Testu .....	83
6.4.4	Ladenie a spúšťanie web testov .....	88
6.5	Tvorba Load Testov .....	93
6.5.1	Vytváranie Zát'azového testu .....	93
6.5.2	Spustenie testu a analýza výsledkov .....	104
7.	Záver .....	109
8.	Literatúra.....	111

## Zoznam obrázkov

Obrázok 1 Dôvod vzniku chýb (Vlastná tvorba) .....	14
Obrázok 2 Interné a externé kvality (Zošiak, 2013, s. 3).....	16
Obrázok 3 Cena chyby (Vlastná tvorba).....	17
Obrázok 4 Krivka nákladov (Perry, 2006, s. 48) .....	18
Obrázok 5 Strategická analýza pokrytia (Vlastná tvorba) .....	18
Obrázok 6 White Box (Vlastná tvorba) .....	23
Obrázok 7 Black Box (Vlastná tvorba).....	24
Obrázok 8 Typy testov počas životného cyklu (Vlastná tvorba) .....	38
Obrázok 9 Vytvorenie Test projektu (Vlastná tvorba).....	45
Obrázok 10 Trieda Fibonacci (Vlastná tvorba).....	46
Obrázok 11 Fibonacci unit test (Vlastná tvorba) .....	46
Obrázok 12 Fibonacci unit test s Assert (Vlastná tvorba).....	47
Obrázok 13 Funkcia Sčítania (Vlastná tvorba) .....	48
Obrázok 14 CSV dátový zdroj (Vlastná tvorba).....	48
Obrázok 15 XML dátový zdroj (Vlastná tvorba).....	48
Obrázok 16 Objekt TestContext (Vlastná tvorba) .....	49
Obrázok 17 CSV Data driven unit test (Vlastná tvorba).....	49
Obrázok 18 XML Data driven unit test (Vlastná tvorba) .....	50
Obrázok 19 app.config connectionString (Vlastná tvorba).....	51
Obrázok 20 Data driven unit test s app.config (Vlastná tvorba).....	51
Obrázok 21 Initialize a Cleanup (Vlastná tvorba).....	52
Obrázok 22 Delenie test (Vlastná tvorba).....	53
Obrázok 23 Unit test s vyvolanou výnimkou (Vlastná tvorba).....	53
Obrázok 24 ExpectedException (Vlastná tvorba).....	53
Obrázok 25 Code Coverage (Vlastná tvorba) .....	54
Obrázok 26 ExcludeFromCodeCoverage (Vlastná tvorba) .....	55



Obrázok 27 Ordered Test (Vlastná tvorba).....	56
Obrázok 28 Výsledky Ordered testu (Vlastná tvorba).....	57
Obrázok 29 Generovanie Coded UI Test (Vlastná tvorba).....	58
Obrázok 30 Coded UI Test Builder (Vlastná tvorba) .....	58
Obrázok 31 Zaznamenané kroky CUIT (Vlastná tvorba).....	59
Obrázok 32 Generovanie kódu (Vlastná tvorba) .....	59
Obrázok 33 Trieda CodedUITest (Vlastná tvorba).....	60
Obrázok 34 GUI Objekt vyjadrený kódom (Vlastná tvorba).....	61
Obrázok 35 UIMap Editor (Vlastná tvorba) .....	62
Obrázok 36 UIMap editácia (Vlastná tvorba).....	62
Obrázok 37 Štart CUIT Builder-u (Vlastná tvorba).....	63
Obrázok 38 Pridanie Assert-u (Vlastná tvorba).....	63
Obrázok 39 Assert v kóde (Vlastná tvorba).....	64
Obrázok 40 Assert v UIMap (Vlastná tvorba) .....	64
Obrázok 41 Automaticky vyplnené hodnoty v aplikácii (Vlastná tvorba).....	64
Obrázok 42 Data driven Coded UI Test (Vlastná tvorba).....	65
Obrázok 43 Spustenie CUIT z príkazovej riadky (Vlastná tvorba) .....	67
Obrázok 44 Vytvorenie Web Performance Testu (Vlastná tvorba).....	69
Obrázok 45 Web Test Recorder (Vlastná tvorba).....	70
Obrázok 46 Ukážka webovej stránky s formulárom (Vlastná tvorba).....	70
Obrázok 47 Ukážka webovej stránky s výsledkom (Vlastná tvorba) .....	71
Obrázok 48 Kód stránky s výsledkom (Vlastná tvorba) .....	71
Obrázok 49 Zaznamenané kroky Web Test Recorder-u (Vlastná tvorba) .....	71
Obrázok 50 Detail Web Performance Testu (Vlastná tvorba) .....	72
Obrázok 51 Panel nástrojov Web Test Editor (Vlastná tvorba).....	73
Obrázok 52 Okno Web Test Properties (Vlastná tvorba) .....	74
Obrázok 53 Okno Web Request Properties (Vlastná tvorba) .....	76

Obrázok 54 Nastavenie Form Post parametrov (Vlastná tvorba) .....	77
Obrázok 55 Pridanie extrakčného pravidla (Vlastná tvorba).....	78
Obrázok 56 Extrakčné pravidlo v Web Test Editore (Vlastná tvorba) .....	79
Obrázok 57 Pridanie Validačného pravidla (Vlastná tvorba) .....	80
Obrázok 58 Validačné pravidlo v Web Test Editore (Vlastná tvorba) .....	80
Obrázok 59 Pridanie transakcie (Vlastná tvorba) .....	81
Obrázok 60 Transakcia v Web Test Editore (Vlastná tvorba) .....	81
Obrázok 61 Pridanie Podmieneného pravidla (Vlastná tvorba).....	82
Obrázok 62 Podmienene pravidlo v Web Test Editore (Vlastná tvorba).....	82
Obrázok 63 Podmienene pravidlo v Test Result (Vlastná tvorba).....	83
Obrázok 64 Panel nástrojov Web Test Editoru (Vlastná tvorba).....	83
Obrázok 65 Pridanie Dátového zdroja (Vlastná tvorba) .....	84
Obrázok 66 Vyber Dátového zdroja a tabuľky (Vlastná tvorba) .....	84
Obrázok 67 Dátový zdroj v Web Test Editore (Vlastná tvorba).....	85
Obrázok 68 Linkovanie dátového zdroju (Vlastná tvorba).....	86
Obrázok 69 Dátový zdroj v Test Result (Vlastná tvorba).....	86
Obrázok 70 Pridanie Prihlasovacích údajov (Vlastná tvorba) .....	87
Obrázok 71 Pridanie Test Settings (Vlastná tvorba).....	89
Obrázok 72 Nastavenie Web Testov (Vlastná tvorba).....	90
Obrázok 73 Test Result Okno (Vlastná tvorba).....	91
Obrázok 74 Detail záložky Request (Vlastná tvorba).....	91
Obrázok 75 Detail záložky Context (Vlastná tvorba) .....	92
Obrázok 76 Detail záložky Details (Vlastná tvorba) .....	92
Obrázok 77 Pridanie Zát'azového testu (Vlastná tvorba).....	94
Obrázok 78 Definovanie Think Time (Vlastná tvorba) .....	94
Obrázok 79 Testovacia vzorka (Vlastná tvorba).....	95
Obrázok 80 Test Mix Model (Vlastná tvorba).....	96

Obrázok 81 Test mix – pridanie testov (Vlastná tvorba) .....	98
Obrázok 82 Test mix – priorita (Vlastná tvorba) .....	98
Obrázok 83 Network Mix (Vlastná tvorba) .....	99
Obrázok 84 Browser mix (Vlastná tvorba) .....	99
Obrázok 85 Sada meračov (Vlastná tvorba) .....	100
Obrázok 86 Run Settings – Nastavenia behu testu (Vlastná tvorba) .....	101
Obrázok 87 Load Test (Vlastná tvorba).....	102
Obrázok 88 Editovanie Zát'azového testu (Vlastná tvorba).....	102
Obrázok 89 Pridanie Pravidla rozsahu (Vlastná tvorba).....	103
Obrázok 90 Nastavenie Pravidla rozsahu (Vlastná tvorba) .....	104
Obrázok 91 Priebeh zát'azového testu (Vlastná tvorba).....	104
Obrázok 92 Detail sumárneho prehľadu (Vlastná tvorba) .....	106
Obrázok 93 Detail tabuľkového prehľadu (Vlastná tvorba) .....	107
Obrázok 94 Detailný prehľad (Vlastná tvorba) .....	108

## **Zoznam tabuliek**

Tabuľka 1 Initialize a Cleanup metódy .....	51
Tabuľka 2 Parametre pre MSTest.exe .....	66
Tabuľka 3 Web Test Properties .....	73
Tabuľka 4 Web Request Properties .....	74

## Úvod

Softvérové systémy a aplikácie sú v dnešnej informačnej spoločnosti prirodzenou súčasťou každodenného života. Svoje využitie našli nielen jednoduché systémy a aplikácie, využívané bežnými používateľmi, ale aj zložité či kritické softvéry, ktoré riadia strategické činnosti našej spoločnosti. S rastúcou veľkosťou a zložitosťou kódu a narastajúcim počtom programátorov, ktorí sa podieľajú na vývoji aplikácií, prirodzene narastá aj počet chýb v aplikáciách. Aj to je jeden z dôvodov, prečo sa kladú čoraz väčšie požiadavky na profesionálne testovanie.

Problémy a chyby vznikajú už v samotnom začiatku vývoja, a to pri tvorbe špecifikácie, ktorá je často nesprávna, neúplná alebo sa mení od logického návrhu aplikácie až po samotné programovanie. Úlohou testovania je dôkladné odhaľovanie a eliminácia chýb v každej fáze vývojového procesu. Napriek tomu, že komplexným a profesionálnym testovaním je možné odhaliť majoritnú časť chýb, ani pri tom najdetailnejšom testovaní nie je možné empiricky dokázať, že sa v aplikácii už žiadne ďalšie chyby nenachádzajú. Vzhľadom k tomu, že aj jednoduchšie aplikácie majú kvantitu kombinácií vstupov, funkcií a výstupov, nie je možné aplikácie otestovať stopercentne. Komplexné testovanie je zložitý proces nielen z technologického, ale aj z časového hľadiska a zaberá až 50% času potrebného na vývoj – cez koordináciu a komunikáciu medzi vývojovým a testerským tímom až po množstvo iných časovo náročných činností. Na druhej strane, vývoj a chod aplikácií je zároveň proces ekonomický, a preto dôležitú úlohu zohráva faktor nákladu na opravy chýb zistených pri testovaní. Ten má tvar logaritmickú krivku, a preto je veľmi dôležité detegovať chyby v skorých fázach vývoja aplikácie. Vzhľadom k rozsahu aplikácií, počtu možných vstupov a výstupov však nie je možné pokryť každú kombináciu. To je jeden z dôvodov, prečo sa hľadá optimálna hladina testovania, kedy sa náklady na samotné testovanie rovnajú peňažnému vyjadreniu počtu nájdených chýb.

K znižovaniu nákladov na testovanie pomáhajú viaceré techniky a nástroje, zamerané väčšinou na tvorbu automatizovaných testov. Jedným z takých nástrojov je Microsoft Visual Studio 2012, ktorý pomáha automatizovať celý proces testovania a poskytuje viaceré možnosti pre zefektívnenie a zrýchlenie celej testovacej fázy. Tento nástroj poskytuje podporu všetkých hlavných typov testov, používaných pri vývoji a taktiež plnú podporu pri agilnom vývojovom cykle. Spolu s integráciou bug-tracking nástrojov, nástrojov pre tvorbu testovacieho plánu, stratégie a test case poskytuje

ucelený ekosystém, ktorý prispieva k zvyšovaniu efektivity a znižovaniu nákladov a predstavuje výhodu oproti konkurenčným nástrojom (prevažne open source charakteru).

Tvorba automatizovaných testov patrí k náročným fázam vývoja a vyžaduje znalosť problematiky testovania, testovacích techník a metodík a samotnú tvorbu špecifických typov testov. Každý typ testu sa zameriava na inú časť vývojovej fázy a testuje rozdielnu funkcionálnosť systému. Cieľom práce je predstavenie procesu vytvárania jednotlivých testov a systematicky prezentovať postup testovania. Práca sa zameriava na komplexné pokrytie testovacích potrieb vyvíjaného systému, postupne od testovania zdrojového kódu cez testovanie grafického rozhrania až po testovanie robustnosti aplikácie a systému, na ktorom je spustená.

# 1 DEFINÍCIA POJMOV

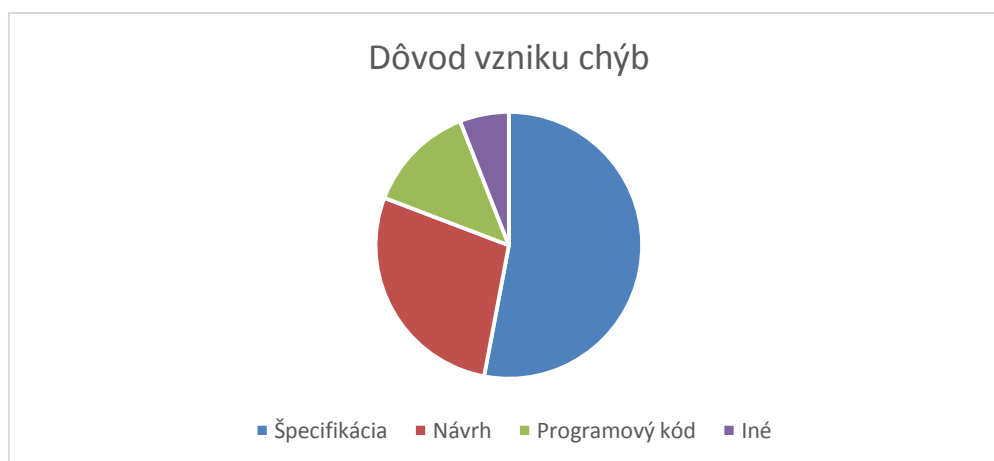
Definícia pojmov ako chyba alebo kvalita z oblasti testovania je dôležitá pre lepšie pochopenie celej problematiky. Keďže pojmy sú abstraktné, môžu u každého evokovať niečo iné, subjektívne, preto je nevyhnutná ich všeobecná definícia.

## 1.1 Chyba

Testovanie je proces overujúci neprítomnosti chýb v systéme. Ale čo je to chyba? Podľa profesora automatizácie a inteligentných systémov inžinierstva, Rona Pattona, o chybe hovoríme vtedy, keď je splnená minimálne jedna z týchto podmienok:

1. Aplikácia nerobí to, čo by podľa špecifikácie mala robiť.
2. Aplikácia robí niečo, čo by podľa špecifikácie robiť nemala.
3. Aplikácia robí niečo, čo nie je uvedené v špecifikácii.
4. Aplikácia nerobí to, čo nie je v uvedené špecifikácii ale malo by byť.
5. Aplikácia je ťažko zrozumiteľná, ťažko sa s ňou pracuje, je pomalá. (Patton, 2002, s.14)

Vychádzajúc z výsledkov štúdií odborníkov na systémové softvéry patrí medzi najčastejšie zdroje chýb špecifikácia a to prevažne z dôvodu jej neexistencie. Ďalším zdrojom chýb je nedostatočne podrobný návrh systému, v ktorom dochádza k neustálym zmenám, prípadne ho v dostatočnej miere nepochopia všetci členovia vývojového tímu. Chyby v programovom kóde sa podieľajú na vzniknutých chybách v menšej miere – ich zdrojmi môže byť celková zložitosť aplikácie, nedostatočná dokumentácia, nedostatok času kvôli termínom odovzdania, prípadne neúmyselnými chybami programátorov. (Patton, 2002, str. 15)



Obrázok 1 Dôvod vzniku chýb (Vlastná tvorba)

## 1.2 Kvalita

Talianska softvérová inžinierka, Antonie Bertolino, tvrdí, že: „*Testovanie sa všeobecne používa na zaistenie kvality*”(Bertolino, 2007, s. 15). Vychádzajúc z tejto teórie je zjavné, že kvalitu vníma každý inak – rozdielny pohľad má na kvalitu programátor, manažér či koncový zákazník.

Norma IEEE Std 610.12-1990 stanovuje pojem kvalita nasledovne: „*Stupeň, do akej miery systém, komponent alebo proces spĺňa špecifikované požiadavky, prípadne spĺňa zákazníkove alebo užívateľove potreby a jeho očakávania.*” (IEEE Standard Glossary of Software Engineering Terminology, 1990, online). Peter Zošiak vo svojej práci uvádza, že norma z roku 1990 nehovorí nič konkrétne o špecifikáciách atribútov kvality, a preto bol roku 1991 publikovaný prvý medzinárodný konsenzus v terminológii charakteristiky kvality pri procese hodnotenia softvérového produktu. (Zošiak, 2013, s. 3)

Tento štandard bol pomenovaný: Hodnotenie softvérového produktu – Charakteristiky kvality a ich použitie (Software Product Evaluation - Quality Characteristics and Guidelines for Their Use (ISO 9126: 1991)). Od roku 2001 do roku 2004 bola zverejnená rozšírená verzia dokumentu, ktorá obsahovala modely kvality a zásady navrhovaných opatrení pre tieto modely. (ISO/IEC IS 9126, 1991). Aktuálna verzia ISO 9126 séria sa skladá z jednej medzinárodnej normy a z troch technických správ:

1. ISO 9126-1: Model kvality
2. ISO TR 9126-2: Externé metriky
3. ISO TR 9126-3: Interné metriky
4. ISO TR 9126-4: Prevádzkové metriky

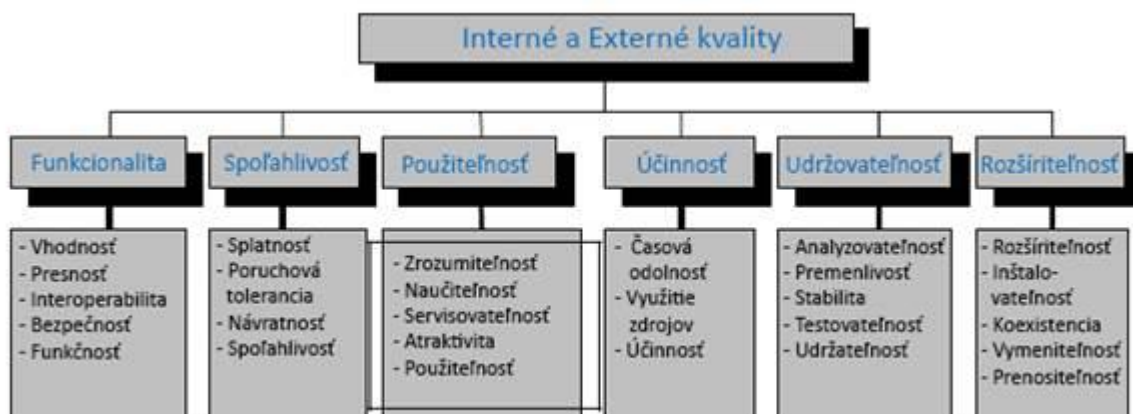
### **ISO 9126-1: Model kvality**

Prvý dokument (Model kvality) obsahuje dve časti:

1. Interné a Externé kvality
2. Prevádzková kvalita

Prvá z dvoch častí modelu kvality určuje šesť charakteristík, ktoré sú rozdelené do 27 pod-charakteristík pre určenie vonkajšej a vnútornej kvality. Tieto čiastkové

vlastnosti sú výsledkom vnútorných softvérových atribútov a pokiaľ je softvér súčasťou systému, sú viditeľné aj navonok.



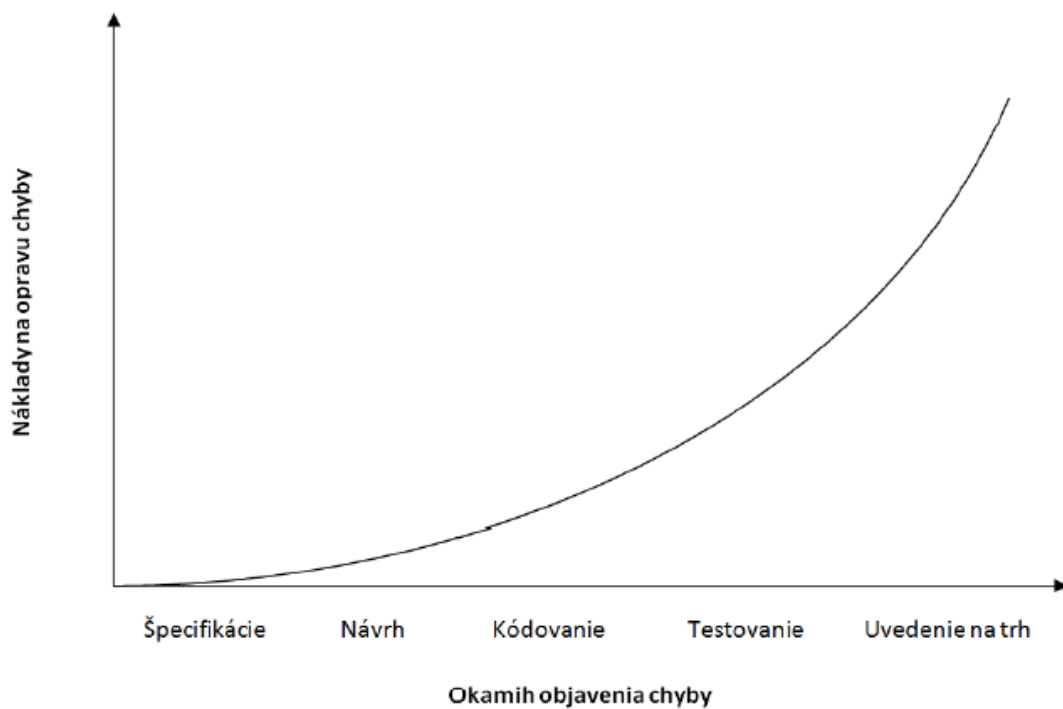
Obrázok 2 Interné a externé kvality (Zošiak, 2013, s. 3)

### 1.3 Ekonomické aspekty

Vzhľadom k tomu, že vývoj softvéru je hlavne ekonomická záležitosť a primárnym cieľom každej spoločnosti je generovať zisk, testovanie je nutné hodnotiť aj z aspektu ekonomického.

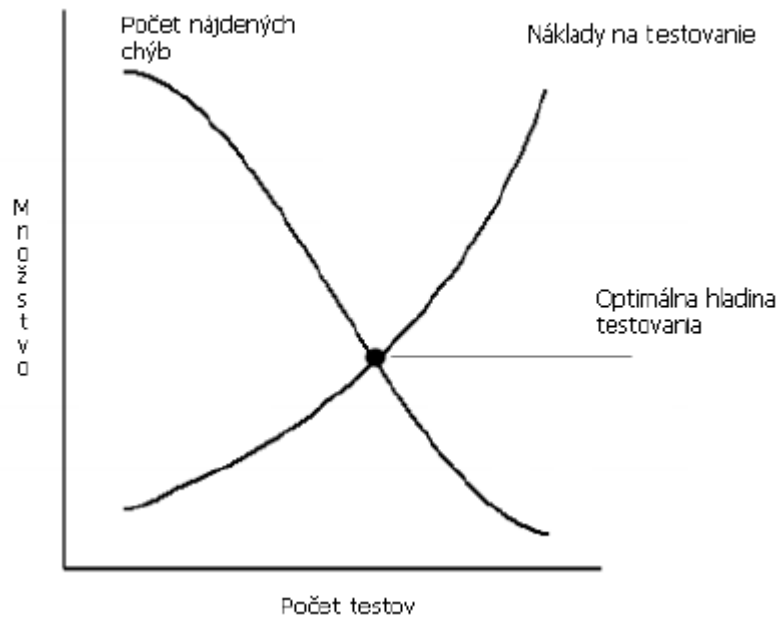
Software vzniká plánovitým, metodickým vývojovým postupom, a preto sa chyby môžu vyskytnúť už prvotných fázach – od zahájenia vývoja až po publikovanie verejnosti. Náklady na opravu chyby rastú s časom exponenciálne, preto je veľmi dôležité kedy chybu odhalíme. Včasným testovaním môžeme ušetriť finančné alebo iné zdroje. Aj malá chyba odhalená príliš neskoro môže v konečnom dôsledku spôsobiť veľké problémy pri vývoji. Stráca sa čas a iné prostriedky potrebné na nápravu a nie je možné korektne napredovať vo vývoji softvéru.





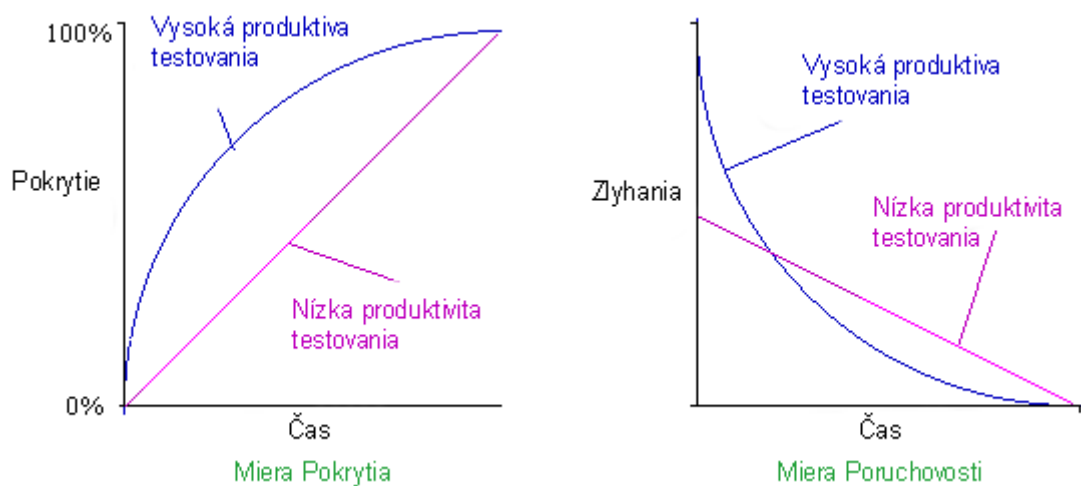
**Obrázok 3 Cena chyby (Vlastná tvorba)**

V prípade testovania aplikácii platí vzťah, že s rastúcim počtom narastá aj percento objavených chýb (klesá počet neobjavených chýb) v aplikácii, avšak narastá množstvo prostriedkov, ktoré je potrebné investovať do samotného procesu testovania. Je preto vhodné nájsť hranicu, kedy je testovanie aplikácie ešte dostatočne efektívne vzhľadom k množstvu neodhalených chýb a primeranosti vynaložených prostriedkov, ako je znázornené na obrázku č. 4.



Obrázok 4 Krivka nákladov (Perry, 2006, s. 48)

Najvyššia úroveň produktivity testovania sa dostiahne pri nájdení čo najviac chýb pri čo najmenšom úsilí. Úsilie je merané časom potrebným na vytvorenie testovacích scenárov, testovacích prípadov a ich následnej exekúcii. Preto je potrebné používať strategickú analýzu pokrytia (coverage analysis strategy), pre zvýšenie pokrytia kódu, tak rýchlo ako je to možné, pretože poskytuje najväčšiu pravdepodobnosť nájdenia chyby skôr v čase.



Obrázok 5 Strategická analýza pokrytia (Vlastná tvorba)

## 1.4 Tester

Hlavnou úlohou testera je objavovanie a vyhľadávanie chýb v aplikácii a overovať, že testovaný softvér splňuje špecifikácie. Od riešenia elementárnych problémov produktu, komunikáciu s programom až po hľadanie a podávanie správ o význame chyby. Tester vytvára popis chyby a zaznamenáva kroky k jej zreprodukovaniu pre jednoduchšie pochopenie podstaty a problému chyby. Ako náhle je objavený určitý problém, tester musí oznámiť jeho vplyv na systém a navrhnúť vhodné riešenie, ktoré by ho pomohlo znížiť, prípadne odstrániť. Tester by nemal testovať iba funkcie, ktoré by za normálnych okolností mali fungovať, ale navrhovať testy tak, aby nimi dokázali odhaliť možné chyby v aplikácii. Neobjavená chyba v priebehu testovania znamená predrazenie celého projektu, je preto potrebné motivovať testera, aby sa snažil uvažovať, ako by bolo možné dané chyby objaviť v procese vývoja softvéru čo najskôr, a tým pádom znížiť celkové náklady potrebné na ich prípadnú opravu. Mal by rozumieť kontextu projektu a napomáhať pri rozhodnutiach, ktoré z neho vyplývajú. Zúčastňuje sa aj rokovaní, kde sa stanovujú normy kvality pre daný výrobok, pretože práve testovaním môže preukázať, že známe funkcie pracujú správne.

## 2. TYPOLÓGIA TESTOV

### 2.1 Manuálne testovanie

Manuálne testovanie je proces ručného testovania softvéru bez použitia automatizačného nástroja alebo skriptu. To si vyžaduje, aby tester hral úlohu koncového užívateľa a aby použil čo najviac zo všetkých funkcií aplikácie, čím sa zaistí správne správanie (Manual Testing, 2001, online). Vo svojej podstate je to veľmi jednoduchý spôsob testovania, pretože tester sleduje kroky pred vytvoreného testovacieho prípadu a identifikuje, či bol daný krok vykonaný úspešne alebo zlyhal preto je to najčastejšia metóda v testovaní software. Odhaľuje veľký počet chýb, ktoré sú v kóde dlhodobé a neprejavajú sa pri automatickom testovaní. Vzhľadom k tomu, že sa dané scenáre testujú vždy znovu a znovu je táto metóda náročná na zdroje a čas a zvyšuje riziko užívateľských chýb.

(Manual Testing, 2008, online) Tester by mal obvykle vykonať nasledujúce kroky:

1. Pochopiť funkčnosť programu.
2. Pripraviť testovacie prostredie.
3. Vykonať testovací prípad.
4. Overiť skutočný výsledok.
  1. Zaznamenať výsledok ako Pass (Vyhovelo) alebo Fail (Nevyhovelo).
  2. Vytvoriť súhrnnú správu o vyhovení alebo nevyhovení testovacieho prípadu.
  3. Publikovať report.
  4. Zaznamenať nové chyby, ktoré sa vyskytli pri realizácii testovacieho prípadu.

### 2.2 Automatizované testovanie

Na testovanie sa používa špecializovaný software, ktorý vykonáva, porovnáva a interpretuje vopred pripravené testy s očakávanými výsledkami (Test automation, 2001, online). V minulosti, kvôli absencii nástrojov, menej častá metóda v testovaní software, v dnešnej dobe naberá na význame. Keďže vývoj softvéru je dynamická činnosť, je

nevyhnutná častá údržba automatických testov, ktorá musí reflektovať zmeny v softvéri aby sa zabezpečila čo najmenšia kolísavosť spoľahlivosti testov.

## 3. TESTOVACIE METÓDY

### 3.1 White Box Testing

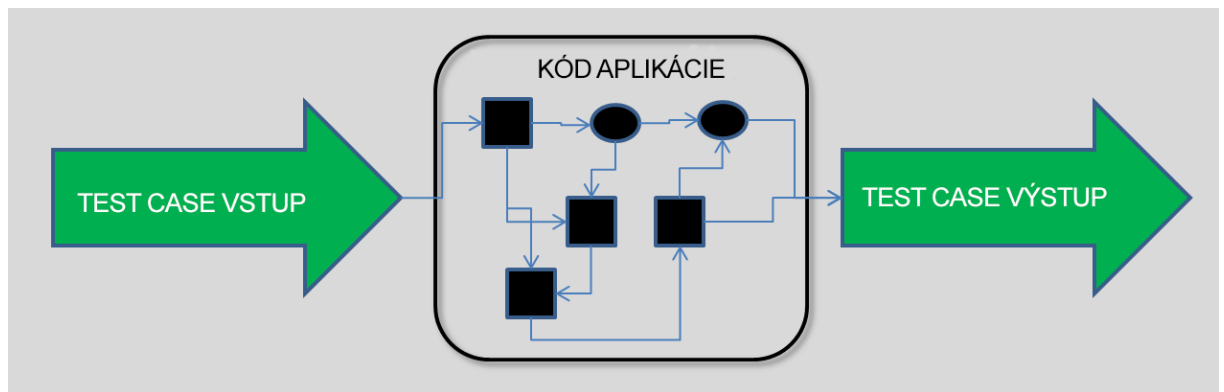
White box testovanie sa zameriava na podrobný prieskum vnútornej logiky a štruktúry kódu. V tomto prípade musí byť tester z časti programátor, pretože je vyžadovaná znalosť kódu a fungovania modulu, dátových a programových štruktúr, aby dokázal identifikovať, ktorá časť kódu sa môže chovať nevhodne. Testerovi sú poskytnuté všetky informácie, dokumentácia a zdrojový kód.

*Výhody:*

- Znalosť kódu vylučuje nevhodné alebo nevalidné scenáre
- Optimalizácia kódu pri oprave -> menší výskyt chýb v ďalšom cykle vývoja
- Odstránenie nepotrebného kódu, ktor7 by mohol byť použitý k prieniku do systému
- Núti vývojára písať komentáre a dokumentáciu
- Široké možnosti použitia rôznych framework-ov

*Nevýhody:*

- Tester musí mať expertné znalosti
- Tester musí výborne poznať samotný modul a jeho nadväznosti na okolité prostredie
- Testovanie je veľmi komplexné, jedná sa o druh vývoja
- Tester môže stratiť nadhľad nad testovaným komponentom
- Vyššie náklady
- Nemožnosť zistiť všetky skryté chyby vzhľadom k vysokému počtu možných alternatív
- Zložitá údržba



Obrázok 6 White Box (Vlastná tvorba)

White box testovanie býva podľa internetových zdrojov často označovaná aj ako audit zdrojového kódu alebo code-review. Môže prebiehať ručne alebo automatizovane za pomoci analytických nástrojov, často sa však oba spôsoby kombinujú (ČERMÁK, 2008, online).

### 3.2 Black box testovanie

Black box testovanie je testovanie bez akýchkoľvek znalostí fungovania vnútorného kódu, dátovej alebo programovej štruktúry aplikácie. Tester nepozná architektúru a taktiež nemá prístup ku kódu. Typický tester interaktívne komunikuje s užívateľským rozhraním, kontroluje vstupné a výstupné dáta na základe testovacieho scenára. Vzhľadom k tomu, že sa oddeľuje pohľad užívateľa (testera) od pohľadu vývojára, je potrebné mať špecifikáciu zadania, v ktorom sú zadané rozsahy vstupných a výstupných hodnôt. Samotné chyby tester neopravuje, len ich reportuje. Pri reportingu je nutné správne oddeliť domnelé chyby od návrhu (feature vs. bug). Tester zodpovedá za prípravu a údržbu testovacieho prostredia.

*Výhody:*

- Vhodné pre testovanie veľkých častí aplikácie (modulov)
- Nie je potrebný prístup ku kódu
- Oddeluje užívateľskú perspektívu od vývojárskej cez priame definovanie rolí
- Väčšie množstvo menej skúsených testrov môže otestovať veľkú časť aplikácie aj bez implementačných znalostí, programovacieho jazyka alebo operačného systému.

*Nevýhody:*

- Obmedzený rozsah závislý na počte testovacích scenárov
- Neefektívnosť vzhľadom na obmedzene znalosti aplikácie
- „Testovanie na slepo“, pretože tester sa nemôže zamerať na konkrétne segmenty kódu
- Nutný návrh testovacích prípadov a scenárov



**Obrázok 7 Black Box (Vlastná tvorba)**

Black box testovanie je možné vykonávať ručne alebo automatizovane, záleží od typu aplikácie.



## 4. TESTOVACIE TECHNIKY

### 4.1 Functional testing - Funkčné testovanie

Testy (skupina testov), ktoré overujú, že aplikácia správne plní všetky úlohy, spĺňa všetky funkčné požiadavky zadané v špecifikácii. Testujú sa všeobecne všetky funkcie, ktoré sú v aplikácii implementované a overuje sa, že fungujú správne, a že zodpovedajú požiadavkám zákazníka. Zároveň sa overuje, či daný softvér disponuje funkciami, ktoré sú uvedené v požiadavkách zákazníka.

Je primárne určené na overenie, že daná časť softvéru spĺňa požiadavky koncového alebo biznis užívateľa. Typicky funkčné testovanie zahŕňa hodnotenie a porovnávanie jednotlivých funkcií aplikácie s biznis požiadavkami. Softvér je testovaný hodnotením výstupov, v akej miere sa zhodujú s požiadavkami. Funkčné testovanie navyše testuje použiteľnosť softvéru, a to tým, že zabezpečuje, aby hlavné funkcie pracovali podľa potreby. Táto technika zahŕňa smoke testy, white box, black box, unit testy, integračné testy, systémové testy, smoke a regresné testy, akceptačné a exploračné testy. Princiipiálne sa používa black box metóda a testujú sa funkcie podľa špecifikácie. Tester nie je v interakcii s programátorom, postupuje podľa testovacieho scénára, pripraveného buď ním alebo iným skúsenejším testerom.

Typ black box testovania je založený na špecifikáciách požiadaviek na softvér, ktorý má byť testovaný. Aplikácia je testovaná tým, že po vložení vstupných dát sa skúmajú a porovnávajú výstupné dáta tak, aby zodpovedali danej funkcionalite. Funkčné testovanie sa vykonáva na kompletnom a už integrovanom systéme. Pozostáva z 5 krokov (Software Testing Tutorial, 2008, online):

1. Stanovanie funkcionality v závislosti na špecifikácií.
2. Tvorba testovacích dát v závislosti na špecifikácií.
3. Výstupné dáta založené na vstupných dátach a špecifikácii.
4. Tvorba testovacích scenárov a vykonávanie testovacích prípadov.
5. Porovnanie skutočných a očakávaných výsledkov na základe vykonaných testovacích prípadoch a špecifikácii.

### 4.1.1 Unit testing - Test Jednotky

Unit testing je metóda, pomocou ktorej sa testujú samotné kusy (unit) zdrojového kódu, sady jedného alebo viacerých modulov spolu s asociatívnou kontrolou dát. Za „unit“ môže byť považovaná aj najmenšia testovateľná časť kódu – typicky jeden riadok kódu, jedna celá trieda alebo aj celý modul aplikácie. Zvyčajne sa jedná o jednu funkciu alebo metódu. Unit testy by mali byť na sebe nezávislé, čiže každý test by mal byť spustiteľný nezávisle na zvyšku. Unit testy sú vytvárané priamo programátormi alebo white box testerami v priebehu procesu vývoja (nie testovania) ako súčasť automatizácie a sú vykonávané ešte pred odovzdaním produktu testovaciemu tímu k exekúcii testovacích prípadov. Vývojári používajú testovacie dáta, ktoré sú rozdielne od testovacích dát používaných testovacím tímom.

Cieľom unit testov je izolovať jednotlivé časti kódu a overiť, že dané časti fungujú správne v závislosti na požiadavky a špecifikácie na funkčnosť (Unit Testing, 2001, online).

- vytvára programátor
- cieľom je testovať minimálne funkčné moduly v kóde
- izolácia unit testov od zvyšku kódu
- unit test je v podstate test case
- v produkčnom kóde je možné unit testy odstrániť (neovplyvnia funkčnosť aplikácie)
- samotný unit test je vykonávaný priamo v kóde

Unit testy nemôžu zachytiť a odhaliť každý bug (chybu) v aplikácií, pretože je nemožné vyhodnotiť všetky možné exekučné cesty pri behu aplikácie. Rovnako je limitovaný aj počet testovacích prípadov, testovacích scenárov a hlavne testovacích dát, ktoré môže vývojár použiť pre overenie kódu.

Dôležitosť unit testov vzrastá od začiatku projektu a zvyšuje sa počas celej doby životného cyklu aplikácie, pretože:

1. zvyšovanie kvality kódu od začiatku,
2. menej bug-ov (chýb),
3. samo-popisovací kód,
4. znižovanie nákladov na opravenie chyby pri neskoršom odhalení.

### 4.1.2 Integration testing - Integračné testy

Táto fáza sa označuje aj ako testovanie vnútornej integrácie. Ide totiž o testovanie správnej komunikácie jednotlivých komponentov vo vnútri aplikácie. Bez ohľadu na to, akým konkrétnym technickým riešením je táto integrácia realizovaná, obsahom týchto testov je overenie, že jednotlivé komponenty si v správny okamih odovzdávajú správnym spôsobom správy, ktoré majú správny obsah aj formát (Systémové a integrační testy, online).

Táto fáza nastáva po Unit testovaní a pred Validáčnými testami. Ako vstupy sú použité moduly, definované integračným plánom, ktorého výstupy určujú, či je daný systém pripravený na systémové testovanie.

Účelom integračných testov je overiť funkcionality, výkon a požiadavky na spoľahlivosť. Testovanie je vykonávané black box metódou – simulované pomocou zadávaním vhodných parametrov a vhodných testovacích dátových vstupov.

Dve funkcie, ktoré sú už otestované unit testami sa spoja do jednej komponenty s tým, že je testovaná ich vzájomná komunikácia a spolupráca. Komponenta v tomto zmysle slova je agregát (zoskupenie) viac ako jednej jednotky (unit – funkcia alebo metóda atď.). Kombinovaných je veľké množstvo Unitov do komponent, ktoré sú kombinované do ešte väčších častí programu. Z toho vyplýva, že zámerom je testovať kombinácie týchto komponent a rozšíriť tento proces na testovanie modulov alebo skupiny modulov. Nakoniec sú všetky moduly, ktoré tvoria nejaký proces, testované spoločne.

Integračné testy nepripravuje programátor, ale predovšetkým testovací tím. Niekedy bývajú označované ako "testy vnútornej integrácie". Musí byť overená bezchybná komunikácia medzi jednotlivými komponentmi vo vnútri aplikácie. Integráciu však možno overovať nielen medzi komponentmi, ale tiež medzi komponentom a operačným systémom, hardvérom či rozhraním rôznych systémov. V tejto fáze sa tak testuje integrácia doteraz jednotlivo overených častí. Postupne sa začína testovať integrácia medzi dvomi komponentmi a postupne sa pridávajú ďalšie. Integračné testy môžu byť ako manuálne aj automatizované.

Úroveň integračného testovania je svojím spôsobom obsiahnutá vo väčšine testovacích postupov softvéru. Pri menších projektoch je však na tieto testy kladený veľmi malý dôraz. Má to svoje logické odôvodnenie. Integračné testovanie možno v testovacom cykle úplne vynechať. Na výslednú bezporuchovosť softvéru to pritom nebude mať žiadny vplyv, teda aspoň za predpokladu, že korektne vykonáme

nasledujúce úrovne testovania. Chyba, ktorú by sme odhalili počas integračných testov, sa celkom iste prejaví v priebehu ďalších úrovní testovania. Ako už bolo spomenuté v kapitole 1.3, počas testovania platí: "čím skôr sa chyba objaví, tým menej prostriedkov stojí jej oprava" (Fáze a úrovně provádění testů, 2010, online). Z toho vyplýva, že integračné testy majú svoj význam a ich použitie nemožno nijako podceňovať.

- sú logickým rozšírením unit testov
- testuje interakciu jednotlivých modulov alebo jednotiek (units)
- testuje integráciu modulov (ich funkcií a metód) v kóde
- pomocou integračných testov možno spojiť moduly vo funkčný celok (podľa špecifikácie)
- vytvára ich programátor aj tester
- možno použiť rôzne metódy SDLC (software development life cycle)
- typicky sa používajú stretnúť top-down, bottom-up a big-bang prístupy
- jedná sa o veľmi dôležitú techniku v testovanie softvéru
- v dobe funkcionálneho programovania bola funkcia sama sebe testom (TP, Delphi)

Existuje niekoľko druhov integračného testovania: big bang, top-down, bottom-up a niekoľko integračných vzorov: Collaboration integration, Backbone integration, Layer Integration, Client/Server Integration, Distributed Services Integration and High-frequency Integration.

#### **4.1.2.1 Big Bang**

V tomto spôsobe sú všetky alebo väčšina vývojových modulov spojené dohromady, aby vytvorili kompletný softvér systém alebo hlavnú časť systému, ktorá sa použije pre integračné testovanie. Metóda Big Bang je veľmi efektívna pre svoju úsporu času pri integrácii. Kládne sa veľký doraz na správne zaznamenanie testovacích prípadov a ich výsledkov (Integration testing, 2001, online).

Druhým typom Big Bang testovania je Usage Model Testing, ktoré sa používa aj pri software, ale aj pri hardware testovaní. Základom tohto typu je vytvorenie užívateľsky podobného prostredia a podmienok pre beh testov. Tento model má „optimistický prístup“, pretože očakáva už vopred niekoľko problémov v kľúčových komponentoch. Stratégia testovania spoľieha na vývojárov komponent a ich izolované unit testy. Tým sa zabráni prerábaniu testov vývojármi na miesto riešenia problémov spôsobených interakciou modulov v produkčnom prostredí. Práve preto je veľmi

zásadná tvorba realistických scenárov. To dáva istotu, že integrované prostredie bude fungovať rovnako ako sa očakávalo pre prostredie produkčné (Integration testing, 2001, online).

#### **4.1.2.2 Top-down and Bottom-up**

Testovanie Zdola na Hor (Bottom-up) je metóda, kedy sa komponenty z nižších vrstiev testujú ako prvé, aby uľahčili testovanie komponenty vyššej úrovne. Tento proces sa opakuje až dovtedy, kým sa nedosiahne vrchol hierarchie. Tým sa zaručí, že všetky nízkoúrovňové funkcie a metódy budú otestované a integrované. Táto metóda je však nápomocná iba vtedy, ak sú všetky alebo väčšina modulov na rovnakej vývojovej úrovni (Integration testing, 2001, online).

Top Down Testing je typ, kde sú testované moduly najvyššej úrovne a každá jedna branch (vetva) modulu je testovaná krok za krokom až do konca príslušného modulu.

#### **4.1.2.3 Sandwich Testing**

Sandwich testing kombinuje Top Down a Bottom-up. Integračné testovanie preto identifikuje problémy, ktoré nastávajú, keď sa jednotlivé časti programu kombinujú. Práve preto vyžaduje testovací plán testovať každý modul (unit) osobitne. Tak sa zaručí ich správnosť pred samotnou kombináciou do komponent a bude možné zhodnotiť, že chyby súvisia pravdepodobne s interakciou medzi komponentmi. Uvedenie delenia:

- Top-down metóda vyžaduje aby boli otestované a integrované najprv moduly najvyššej úrovni. To umožňuje overiť a otestovať logiku a tok dát už na začiatku procesu. Nevýhodou je, že nízkoúrovňové funkcie sú testované relatívne neskoro počas vývojového cyklu.
- The bottom-up vyžaduje, aby boli otestované integrované nízkoúrovňové unity najskôr.
- Tretia metóda kombinuje obe predošlé a vyžaduje funkčné testovacie dáta pre kontrolu toku týchto dát. Ako prvé, vstupy pre funkcie sú integrované vzorom bottom-up. Ich výstupy sú pre každú funkciu potom integrované vzorom top-down

### 4.1.3 System testing - Systémové testovanie

Po overení správnej integrácie nastáva čas na systémové testovanie. Počas týchto testov je aplikácia overovaná ako funkčný celok. Tieto testy sú používané v neskorších fázach vývoja a overujú aplikáciu z pohľadu zákazníka. Podľa pripravených scenárov sa simulujú rôzne kroky, ktoré v praxi môžu nastať. Zvyčajne prebiehajú v niekoľkých kolách. Nájdené chyby sú opravené a v ďalších kolách sú tieto opravy opäť otestované. Súčasťou tejto úrovne sú ako funkčné tak nefunkčné testy. Systémové testy tak predstavujú poslednú úroveň testov, ktoré sa vykonávajú pred odovzdaním produktu zákazníkovi. Táto úroveň testov väčšinou slúži ako výstupná kontrola softvéru. Systémové testovanie je obsiahnuté prakticky v každom procese testovania. Bez tejto úrovne by celé testovanie softvéru nemalo žiadny význam. Bezporuchovosť výsledného produktu by bola významne ohrozená (Fáze a úrovně provádění testů, 2010, online). Preto je táto úroveň testov považovaná za kľúčovú v celom postupe testovanie softvér. Na realizáciu týchto testov by sa malo myslieť už v rannom štádiu návrhu postupu testovania tak (Software Testing Tutorial, 2013, online).

Na systém testing nadväzuje „systém integration testing“, ktorý charakterizujú nasledovné znaky:

- testuje interakciu viac programov alebo modulov v rámci už existujúcich inštalácií,
- zabezpečuje, že program bude fungovať v cieľovom prostredí,
- interakcia s OS aplikáciami,
- interakcia s vlastnými modulmi,
- integrácia s vlastnými verziami v danom OS.

### 4.1.4 Smoke testing

V okamihu, keď je aplikácia dodaná v stave, kedy je testovateľná (a teda je vyvinutá, nainštalovaná, spustená a prístupná) sa obvykle ako prvé spúšťajú Smoke testy. Tie majú za úlohu overiť, že aplikácia je skutočne vhodná k testom. Obsah týchto testov sa líši vo vzťahu tester – tester či firma od firmy. Smoke testy sa zvyčajne zameriavajú iba na hlavné funkcie aplikácie, a to iba v ich pozitívnom priebehu. Netestuje sa validácia vstupov ani formáty výstupov. Smoke testy môžu byť automatizované, a to aj v prípade, keď je zvyšok testov vykonávaný manuálne. To je dané práve tým, že rozsah týchto testov je podstatne menší a tým, že sa zameriavajú na funkčnosti, u ktorých sa nepredpokladajú veľké zmeny. Práve preto nie je potreba

automatizovaný testovací skript často upravovať (Systémové a integrační testy, online).

Kategória smoke testov sa využíva v okamihu, kedy je dokončený vývoj aplikácie a možno ju spustiť – na konci úrovne integračného testovania. Jedná sa o krátky test, ktorý slúži ako rýchle overenie, či je vyvíjaná aplikácia pripravená pre ďalšiu fázu testovania. Zvyčajne sa vykoná len jednoduché overenie, že všetky časti aplikácie sú implementované, nainštalované a spustené. Smoke testy sa zameriavajú iba na hlavné funkcie programu, ktoré nebývajú príliš často upravované. Keďže je rozsah smoke testov menší ako v ostatných kategóriách a pokrývajú len hlavné funkcie, sú veľmi často automatizované. Pokiaľ nie sú automatizované v plnom rozsahu, zvyšok testov je vykonávaný manuálne. Úspešné vykonanie smoke testov je podmienkou pre vstup do systémovej úrovne testovania (Hlava, 2010, online).

Smoke testing sa vykonáva na uzavretých systémoch za účelom nájdenia kritických zlyhaní, pred uvedením do prevádzky, aby sa overilo, či má zmysel testovať. Overuje sa základná funkcionálna, ale nejde sa do hĺbky tak, aby sa pokryla väčšina hlavných funkcií. Výsledok tohto testu sa používa na rozhodnutie, či pokračovať s ďalším testovaním, pretože ak test zlyhá, ďalšie testovanie predstavuje stratu času a úsilia (Smoke test, 2001, online).

Smoke testy sú nápomocné pri skorej fáze projektu, keď je možné odhaliť chyby integrácie. Napriek tomu, že môžu byť vykonávané aj ručne, bývajú väčšinou automatizované.

- Test, ktorý verifikuje, že všetky požadované komponenty sú v programe prítomné.
- Overuje, že všetky kritické komponenty fungujú správne.
- Testuje stabilitu komponentov pri rôznych prechodoch ich použitia.
- Používajú ho vývojári a tester.
- Mal by byť automatický a dobre dokumentovaný.
- Je súčasťou regression testing.
- Smoke test testuje celú aplikáciu "end to end".
- Občas sa nazýva "general health check up"

#### **4.1.5 Regression testing - Regresné testovanie**

Kedykoľvek sa vykoná aj malá zmena v kóde aplikácie, môže táto zmena ovplyvniť a zasiahnuť viaceré oblasti aplikácie. Overenie, že opravená chyba nespôsobila novú chybu alebo inak nenarušila funkcionálnu, sa nazýva Regresné

testovanie. Zámerom tohto druhu testovania je zabezpečiť, aby vykonané zmeny nevedli k ďalším chybám (Software Testing Tutorial, 2013, online).

Regresné testy sa využívajú pri opätovnom testovaní funkcií a vlastností aplikácie. Regresné testy sa používajú na overenie, že vykonané zmeny či implementácia nových vlastností v aplikácii nemá žiadny vplyv na existujúce funkcie a vlastnosti. Jedná sa predovšetkým o oblasti, ktoré zostali v programovom kóde nezmenené. Oblasti, ktoré boli predmetom opráv, by správne mali byť už otestované funkčnými testami. Takéto situácie spravidla nastávajú po opravení chýb či po novom release. Regresné testy je vhodné automatizovať (Hlava, 2010, online).

V praxi sú regresné testy veľmi rozšírené a používajú sa v rôznych formách takmer vo všetkých projektoch. Využívajú sa hlavne pri pretestovaní opravených chýb. Oproti tomu regresné testy nie sú príliš rozšírené a často je táto kategória testov rozložená do iných kategórií.

#### **4.1.6 Acceptance testing - Akceptačné testovanie**

Cieľom akceptačného testu je overenie úplnosti a funkčnosti informačného systému, a to zo strany zákazníka aj zo strany klienta. Akceptačný test vychádza z návrhu akceptačného testu, kde sú stanovené základné podmienky pre realizáciu jednotlivých testov. Testy boli navrhnuté tak, aby bola pokrytá celá problematika, riešená v rámci projektu (JANDORA, 2011, s 4).

Testy majú overiť, že aplikácia spĺňa všetky zákazníkove požiadavky. Tieto testy môže vykonávať rovnaký tím, ktorý vykonal systémové testy. Nie je výnimkou aj prípad, že zákazník týmito testami poverí iný tím, vytvorený špeciálne pre akceptačné testy. Požiadavky, ktoré zákazník na funkciu aplikácie má, je vhodné ešte pred začatím vývoja (a celkom iste pred začatím testovania) zhrnúť do tzv. akceptačných kritérií. Tie môžu obsahovať požiadavku na maximálny počet chýb nájdených pri testovaní, výkonové a záťažové požiadavky a podobne. Splnením týchto požiadaviek vývoj aplikácie v istej miere končí a aplikáciu preberá zákazník (Druhy testování v procese vývoje SW, online).

Nájdené nezrovnalosti medzi aplikáciou a špecifikáciou sú reportované späť vývojovému tímu. Opravené chyby sú nasadené do prostredia u zákazníka. V tejto úrovni je pravdepodobne najdôležitejšie definovať dopredu, akou formou bude prebiehať oznamovanie chýb od zákazníka a ako zabezpečiť opravenie týchto chýb v čo možno najkratšom čase. Ak sú už nejaké chyby v úrovni akceptačných testov objavené,



je nutné v čo najkratšom čase tieto chyby opraviť a odovzdať zákazníkovi k ďalším testom. Veľké oneskorenia v nájdení a opravení chyby (v akceptačnej úrovni testovania) môžu viesť k oneskoreniu termínu nasadenia softvéru do prevádzky (Fáze a úrovně provádění testů, 2010, online).

#### **4.1.6.1 Alpha Testing**

Alpha Testing predstavuje prvú fázu testovania, počas ktorej sa vykonáva unit, integračné a systémové testovanie. Podľa internetových zdrojov sa toto testovanie neodporúča na iné činnosti, než je testovanie vývojármi. Po odhalení základných chýb sa program presunie do beta verzie (Software testing, 2001, online).

#### **4.1.6.2 Beta Testing**

Beta testing sa vykonáva po úspešnom alfa testingu a je označované ako pre-release štádium, kde aplikáciu testuje väčšia vzorka zväčša koncových užívateľov v reálnom prostredí.

#### **4.1.6.3 Usability test**

Usability test sa zameriava predovšetkým na zistenie stupňa náročnosti produkt prevádzkovať a používať. Hodnotenie použiteľnosti vychádza z kritérií:

- jednoduchosť - ako rýchlo sa užívateľ naučí daný produkt používať;
- efektívnosť - ako rýchlo dokáže s daným produktom po tom, čo sa ho naučil používať, splniť zadané úlohy;
- presnosť - koľko chýb užívateľ urobí, ako sú závažné, a či má možnosť ich napraviť;
- zapamätateľnosť - ako rýchlo si dokáže spôsob ovládania vybaviť, keď produkt dlho nepoužíval;
- spokojnosť - či je užívateľ s produktom spokojný, páči sa mu a bude ho rád používať.

#### **4.1.7 Exploratory testing - Exploračné testovanie**

V exploračnom testovaní tester pracuje s aplikáciou akýmkoľvek jemu vhodným spôsobom tak, aby preskúmal celú aplikáciu bez akýchkoľvek obmedzení. Vhodný pre testovanie je aj skúsený tester, ktorý vie, na ktoré časti aplikácie sa má primárne zamerať, ale aj neskúsený tester, ktorého chovanie je náhodné a poskytuje tak programátorom nový uhol pohľadu používania aplikácie. V druhom prípade dokáže

tester odhaliť chyby, na ktoré sa pri návrhu nemyslelo.

Nevýhodou je „lack of time“, strata času, keď sa niektoré funkcie testujú znova a znova rovnakým spôsobom.

Medzi hlavné ciele exploračného testovania patrí:

- Pochopiť ako aplikácia funguje, vypadá a chová sa jej rozhranie a aké funkcie implementuje. Tieto ciele sú zväčša prijaté na začiatku vývoja aplikácie, aby sa určili vstupné testovacie body, identifikovali sa návrhové problémy a pripravil sa podklad pre tvorbu testovacích prípadov. Aby sa získal pohľad na potrebnú hĺbku testovania, využívajú sa väčšinou skúsení testerí.
- Identifikovať hranice schopnosti aplikácie a dokázať, že daná funkcia vykonáva to, na čo bola navrhnutá a spĺňa požiadavky.
- Identifikovať potenciálne slabé miesta aplikácie.

Hlavnou myšlienkou prieskumného testovania je použitie reálne existujúceho scenára, ktorý v tomto prípade predstavujú skutoční užívatelia. Scenár je všeobecný návod, aké kroky vykonať počas testovania, ktoré vstupne dáta použiť a s ktorými výstupnými dátami porovnať výsledky. Popisuje spôsob, akým daný prípad testovať. Pri exploratívnom testovaní sa testuje funkčnosť popísaná v scenári, a to rôznymi a alternatívnymi spôsobmi. Kvantita možností zaručí, že čím viac možností je otestovaných, tým je väčšia istota, že aplikácia bude pracovať robustne aj v rukách neskúsených užívateľov. V exploračnom testovaní sa kladie doraz na osobnú slobodu a zodpovednosť testera (Integration testing, 2001, online).

Najväčším problémom exploračného testovania je fakt, že tester je človek, ktorý sa časom stráveným testovaním učí a vytvára si návyky, ako aplikáciu používať. Skôr či neskôr funkcie testuje rovnakým spôsobom, a preto sa tento spôsob s pribúdajúcim časom stáva neefektívnym.

## **4.2 Non-Functional Testing - Nefunkčné testovanie**

Nefunkčné testy spočívajú v testovaní všetkých vlastností aplikácie, ktoré priamo nesúvisia s jej funkciami, ale zároveň sú podstatné pre jej správne fungovanie. Patrí sem predovšetkým výkonové testovanie (Performance testing), ktoré má napríklad overiť, že aplikácia aj pod záťažou (väčší počet súčasne pracujúcich užívateľov, väčší objem dát, atď.) bude pracovať dostatočne "svižne". Testuje sa tiež pripravenosť

aplikácie pre budúci nárast záťaže. Testovanie sa zameriava tiež na správanie aplikácie k počítaču, na ktorom sa aplikácia nainštalovala. Testovaním je potrebné preskúmať, či aplikácia zbytočne nezaťažuje pamäť a procesor. Nefunkčné testovanie sa zaoberá testovaním z pohľadu rýchlosti, bezpečnosti, spoľahlivosti, použiteľnosti dokumentácie a podobne. (Hlava, 2010, online)

Nefunkčné testy zahŕňajú testovanie výkonu, záťažové testovanie, stres testovanie, testovanie použiteľnosti, testovanie udržateľnosti, testovanie spoľahlivosti a testovanie prenositeľnosti. Je testovaním toho, "ako" systém pracuje (ako celok).

#### **4.2.1 Performance Testing**

Medzi príčiny, ktoré prispievajú pri znižovaní výkonu softvéru, patrí:

- oneskorenie siete,
- spracovanie na strane klienta,
- databázy spracovanie transakcií,
- rozloženie záťaže medzi servermi,
- vizualizácia dát.

Testovanie výkonu je považované za jeden z dôležitých typov nefunkčného testovania a medzi hlavné parametre patria aspekty:

- rýchlosť (tj. doba odozvy, vizualizácie dát a prístupu k nim),
- kapacita,
- stabilita,
- škálovateľnosti.

Záťažové testy pomáhajú určiť, či bude testovaná aplikácia a systém pracovať správne aj pri záťaži. Ako predlohy slúžia typické užívateľské úkony a situácie (scenáre), ktoré sa v aplikácii vyskytujú, a preto sa zameriavajú na vysoko frekventované oblasti aplikácie.

Existujú rôzne nástroje, ktoré dokážu simulovať práve túto záťaž, a to aj s ohľadom na jej reálne rozloženie v rámci celého dňa (napríklad nízku prevádzku v noci, opakované vrcholy v priebehu dňa, atď.). Pri takto narastajúcej záťaži by sa nemali výraznejšie predlžovať odozvy aplikácie. Performance testy sa opäť robia ako overenie splnenia požiadaviek definovaných zákazníkom. Ten v rámci svojich požiadaviek na aplikáciu uvádza tiež odhad predpokladanej záťaže a jej nárast do budúca. Okrem toho

je vhodné jasne určiť, aká doba odozvy je ešte prijateľná. Performance testy potom skúmajú správanie aplikácie predovšetkým v hraničných hodnotách a po ich prekročení (Funkční vs. nefunkční testování, online).

Medzi základné typy Performance testov patrí:

- **Zát'azový test** (Load Test) – cieľom tohto testu je zistiť, ako sa systém bude správať v reálnej prevádzke, kedy k nemu bude súčasne pristupovať rôzne množstvo užívateľov. Je to technika testovania, ktorou sa miera správanie systému pri normálnych a v predpokladaných extrémnych podmienkach. To pomáha určiť maximálnu prevádzkovú kapacitu aplikácie. Tento test je nevyhnutný pre multi-užívateľské aplikácie a pre aplikácie typu CLIENT-SERVER. Testy sa vykonávajú zvyčajne v testovacom prostredí, ktoré je totožné s produkčným, prípadne priamo v produkčnom prostredí počas minimálnej prevádzky (Load testing, 2014, online).
- **Test hraničnej zát'aže** (Stress Test) – cieľom testu je určiť robustnosť softvéru pomocou testovania za hranice bežnej prevádzky. Stresové testovanie sa používa pri testovaní softvéru alebo hardvéru, aby sa určilo, či je stabilita systému vyhovujúca v extrémnych a nepriaznivých podmienkach, ktoré môžu nastať v dôsledku zosilnenej prevádzky na sieti, procesoch, pri pretaktovaní alebo pri iných maximálnych požiadavkách na využitie zdrojov.(Stress Testing, 2014, online).
- **Test odolnosti** (Soak Test) – test zahŕňa testovanie systému s výrazným zaťažením, rozšíreným po dlhú dobu, rádovo niekoľko dní alebo týždňov. Test overuje stabilitu a výkonové charakteristiky systému po dlhšiu dobu. Tento typ testu pomáha identifikovať problémy týkajúce sa alokácie pamäti (memory-leak) atď.
- **Test zlyhania** (Failover Test) – test overuje schopnosť systému presunúť operácie na záložný systém. Tento typ testovania sa používa pre overenie schopnosti systému pokračovať v činnosti, zatiaľ čo schopnosť spracovania je prevedená na záložný systém. Tento typ testovania určuje, či je systém schopný alokovať ďalšie zdroje ako je ďalší procesor alebo server počas kritických porúch. Tento typ testovania vyžaduje špecialistu na daný systém, aby sa mohla nasimulovať Failover podmienka. Zároveň je potrebný k určeniu, čo v praxi Failover znamená.

Rozoznávajú sa dva druhy Failover testov:

Transparentný – keď užívateľ alebo správca nevie o tom že systém zlyhal.

Netransparentný – keď si je užívateľ vedomí zlyhania systému a aktívne sa snaží systém opäť uviesť do prevádzky (Failover Tests, 2004, online).

- **Test citlivosti siete** (Network Sensitivity Test) – testy sú variáciou na Load a Performance testy a zameriavajú sa výhradne na obmedzenie a výkon sieťovej aktivity. Vďaka nim sa môže predpovedať vplyv segmentu siete na aplikáciu/systém. Testujú sa rôzne latencie siete, rôzna priepustnosť, rýchlosť a ich vplyv na chod aplikácie (Network Sensitivity Tests, 2004, online).
- **Test objemu dát** (Volume Test) – testovanie aplikácie s určitým množstvom dát. Množstvo dát môže byť všeobecný pojem a môže sa jednať napr. o veľkosť databázy, veľkosť súboru (XML, DOC atď.) s ktorým aplikácia alebo jej funkcia pracuje. Testuje sa výkon v čase práce s danými dátami a bod, pri ktorom sa degraduje výkon a stabilita aplikácie (Volume Testing, 2013, online).

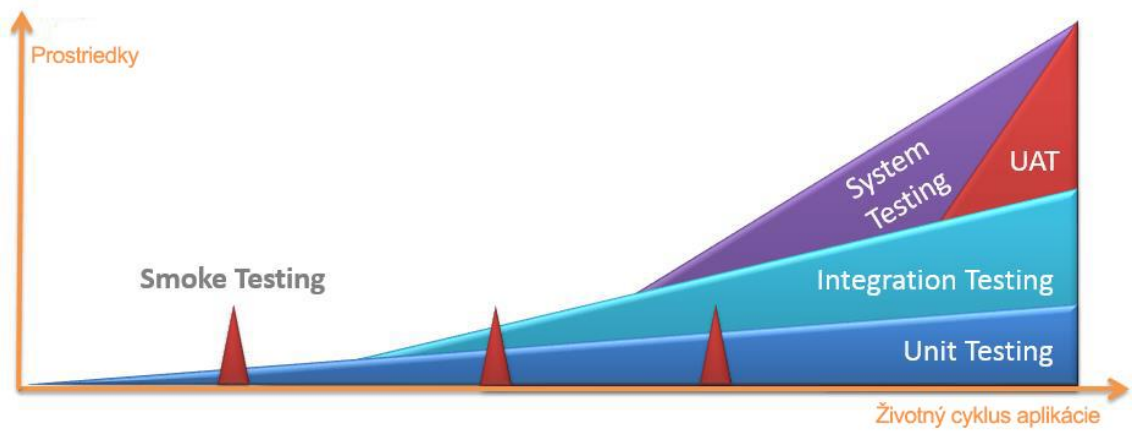
#### 4.2.2 Security Testing

Bezpečnostné testy vedú k objaveniu nových zraniteľností systému alebo aplikácie, ktoré sa neskôr definujú v bezpečnostných požiadavkách. Tento druh testovania je obzvlášť náročný a zložitý, pretože vyžaduje od testera, aby myslel a správal sa ako útočník.

Táto forma testovania je rozdelená na 3 časti:

1. **Prieskum** - identifikácia vstupných bodov systému a ochranných funkcií, zhromaždenie informácií z procesu modelovania na určenie očakávanej cesty útoku.
2. **Identifikácia chýb** – tvorba testovacích prípadov (test case) s riadenými alebo polo-náhodnými dátami (získanie session, modifikácia URL, atď.).
3. **Zneužitie** – simulácia zneužitia.

Testovanie má zabezpečiť predovšetkým dôvernosť, integritu, autentifikáciu a dostupnosť.



Obrázok 8 Typy testov počas životného cyklu (Vlastná tvorba)

## 5. Testovacia dokumentácia

Testovacia dokumentácia v sebe zahŕňa dokumenty a artefakty, ktoré by mali byť vypracované pred alebo počas testovania softvéru a popisuje plánovanie a dizajn testov, vykonávanie, výsledky testov a závery testovacej činnosti. Dokumentácia k testovaniu softvéru pomáha pri odhadovaní úsilia potrebného na testovanie so zameraním na zdroje, odhaduje code coverage a pridáva možnosť sledovania priebehu celého testovacieho procesu, čím uľahčuje možnosť spätnej kontroly. Táto časť obsahuje popis niektorých bežne používaných zdokumentovaných artefaktov, vzťahujúcich sa na testovanie softvéru, ako sú:

- testovacia stratégia,
- testovací prípad,
- testovací scenár.

### 5.1 Test Strategy - Testovacia stratégia

Je to dokument, ktorý opisuje testovaciu časť vývoja softvéru. Mal by poskytovať informácie pre projektových manažérov a testerov v otázkach procesu testovania ako sú hlavné ciele testovania, metódy testovania nových funkcií, celkový čas a zdroje, testovacie nástroje a prostredie. Rovnako popisuje aj možné rizika vyvíjanej aplikácie. Dokument má zvyčajne nasledujúce štruktúru:

- Úvod (Introduction/Management summary)  
Stručné informácie o testovanej aplikácii s definovaním parametrov ako konkrétny tester aplikácie, použitie aplikácie a popis jej základných častí. V úvode testovacej stratégie sa zvyčajne sa uvádza aj HW a SW konfigurácia.
- Dokumentácia (Test items)  
Obvykle sa jedná o špecifikáciu softvérových požiadaviek (Software Requirements Specification), návod (User Guide), príručku operátora (Operator Guide) a inštalačnú príručku (Installation Guide) (ČERMÁK, 2009, online).

- **Vlastnosti (Features to be tested)**  
Komplexný sumár vlastností, ktoré sa majú testovať. Pri každej vlastnosti by mali byť popísané parametre ako funkčnosť, vstupné a výstupné hodnoty a pod.
- **Priebeh testov (Approach)**  
Stručný opis toho, ako jednotlivé testy budú prebiehať a aký spôsob testovania bude u jednotlivých typov testov použitý. V tejto kapitole je potrebné tiež uviesť nad akými dátami a v akom prostredí bude testovanie prebiehať a či sa budú dáta obsahujúce napr. osobné údaje nejakým spôsobom modifikovať (ČERMÁK, 2009, online).
- **Výstupy testovania (Test deliverables)**  
Zoznam dokumentov, ktoré by mali v rámci testovania vzniknúť. Jedná sa o dokumenty, ktoré popisujú testovacie prípady, testovacie procedúry, testovacie logy a záverečnú správu o testovaní. (ČERMÁK, 2009, online)
- **Harmonogram testovania (Schedule)**  
Vzhľadom k tomu, že každý projekt disponuje obmedzenými zdrojmi, nie je možné otestovať všetko a rovnako detailne. Určujú sa preto prioritné oblasti, ktoré sa testujú dôkladne a oblasti, ktoré sa testujú zbežne. Dôkladnosť testovania väčšinou závisí od toho, ktoré funkčnosti sú pre zákazníka najdôležitejšie. To je zároveň dôvod, prečo by mal byť zákazník do testovania zapojený. (ČERMÁK, 2009, online)
- **Zodpovednosti (Responsibilities)**  
Rovnako ako v akomkoľvek inom projekte aj pri testovaní možno identifikovať rôzne skupiny alebo jednotlivcov, ktorí majú zodpovednosť za splnenie jednotlivých úloh. Sú to predovšetkým osoby zodpovedné za riadenie celých testov, prípravu podmienok pre testovanie, vlastný test tím, správca systému, vývojárov a v neposlednom rade aj vlastníka dát. Zodpovednosti jednotlivých osôb a skupín je vhodné uviesť práve v tejto časti dokumentu. (ČERMÁK, 2009, online)



- Rizika (Risks and contingencies)  
V tejto časti sa popisujú riziká, ktoré by mohli vzniknúť pri vývoji a testovaní aplikácie.
- Súhlas s testovaním (Approvals)  
Na poslednej strane dokumentu sa zvyčajne uvádza dátum a mená osôb, ktoré svojim podpisom odsúhlasili daný testovací plán (ČERMÁK, 2009, on-line).

## 5.2 Test case - Testovací případ

Testovací případ představuje pro testera východiskový podklad, který hovorí o spôsobe a postupe testovania konkrétneho miesta v aplikácii. Definuje, za akých podmienok je danú lokalitu možné otestovať a zároveň definuje typ vstupných dát a popisuje očakávaný výsledok.

Testovací případ (test case) indikuje postup, ako otestovať danú požiadavku. Testovací případ popisuje, ako otestovať požadovanú funkčnosť, t.j. čo má byť zadané na vstupe a čo možno očakávať vo výstupe. Testovacie prípady si môžeme rozdeliť na logické a fyzické: (ČERMÁK, 2009, online)

- Logický testovací případ (logical test case) je abstraktný opis toho, čo sa má otestovať, definuje obor a množinu hodnôt.
- Fyzický testovací případ (physical test case) je konkrétny opis toho, čo sa má otestovať, definuje aké hodnoty sa majú zadať. (ČERMÁK, 2009, online)

Každý testovací případ by mal obsahovať:

- Stanovenú prioritu  
Vychádza sa z toho, aký vplyv by mala chyba na podnikového zákazníka a podľa toho sa stanovuje výsledná priorita testovacieho prípadu.
- Opis predmetu testu  
U každého testovacieho prípadu je potrebné uvádzať okrem stručného popisu aj odkaz na dokumentáciu, kde je uvedený detailný popis vlastnosti, ktorá má byť testovaná. Tak je možné zabezpečiť, že predmetné vlastnosti budú pokryté zodpovedajúcimi testovacími prípadmi. (ČERMÁK, 2009, online)

➤ Vstupné hodnoty

Za vstup môžeme považovať hodnotu zadanú z klávesnice, vybranú myšou alebo načítanú zo súboru. Testujú sa rôzne situácie, ako sa aplikácia zachová a aký je výsledok operácie ak je napríklad zadané číslo mimo rozsah, desatinná čiarka miesto bodky a naopak, ak je číslo sa zápornou hodnotou, matematický výraz alebo textový reťazec tam, kde aplikácia očakávala číslo. (ČERMÁK, 2009, online)

➤ Kroky testu

Jednotlivé kroky testu, ktoré je potrebné vykonať.

➤ Doba spracovania

Medzi vstupom a výstupom prirodzene dochádza k určitému spracovaniu. Pri niektorých typoch testov je podstatná aj doba spracovania. V tomto prípade by mala byť uvedená konkrétna hodnota, ktorá sa má dosiahnuť. (ČERMÁK, 2009, online)

➤ Výstupné hodnoty

Výstup je obvykle zobrazený na obrazovke alebo zapísaný do súboru. Výstup sa porovnáva s očakávanou hodnotou.

➤ HW a SW konfigurácia

Ak má mať testovanie nejakú výpovednú hodnotu, musí byť definované na akej HW a SW konfigurácii sa majú jednotlivé testovacie prípady vykonávať.

➤ Pre rekvizity

V niektorých prípadoch nemôže test začať skôr, než sa skončí iný test. Táto informácia nadobúda na význame obzvlášť v prípadoch, kedy sa testovania zúčastňuje viac testerov. (ČERMÁK, 2009, online).

### 5.3 Scenár

Scenár popisuje kroky užívateľa v systéme za účelom dosiahnutia určitého výsledku s vopred danými dátami. Definuje typické znalosti, skúsenosti, potreby a pracovné návyky užívateľov a zbiera reálne dáta, popisujúce dôležité charakteristiky správania sa určitej skupiny. Scenár pozostáva z niekoľkých testovacích prípadov, vykonaných v logickej postupnosti. Môžu to byť testovacie prípady, ktoré na seba naväzujú a musia byť vykonané v presne uvedenom poradí alebo prípady, kedy na poradí, v akom sú jednotlivé testovacie prípady vykonávané, nezáleží. Testovací scenár môže mať

rovnakú hierarchickú štruktúru ako má špecifikácia požiadaviek. Cieľom test dizajnéra by malo byť pokrytie všetkých požiadaviek zodpovedajúcimi testovacími prípadmi.

Ak má testovací scenár formu dokumentu, zvyčajne pozostáva z nasledujúcej štruktúry:

- Vlastnosti, ktoré budú testované (Features to be tested);
- Prístup k testovaniu (Test approach);
- Testovacie prípady;
- Kritériá (Pass / Fail criteria).

## 6. Tvorba testov v Microsoft Visual Studio

Visual Studio (ďalej len VS) je hlavný nástroj s integrovaným vývojovým prostredím (IDE), v ktorom Microsoft zúročuje dlhoročné skúsenosti s vývojom softvéru. Visual Studio je na trhu už roku 1995 a je dostupné vo viacerých edíciách, zameraných na jednotlivých členov vývojového tímu. Vo verzii 2012 ponúka v edíciách Premium a Ultimate plnú podporu pre tvorbu všetkých typov automatizovaných testov. Edícia Professional ponúka čiastočnú podporu, a to len unit testov (Microsoft Visual Studio, 2013, online).

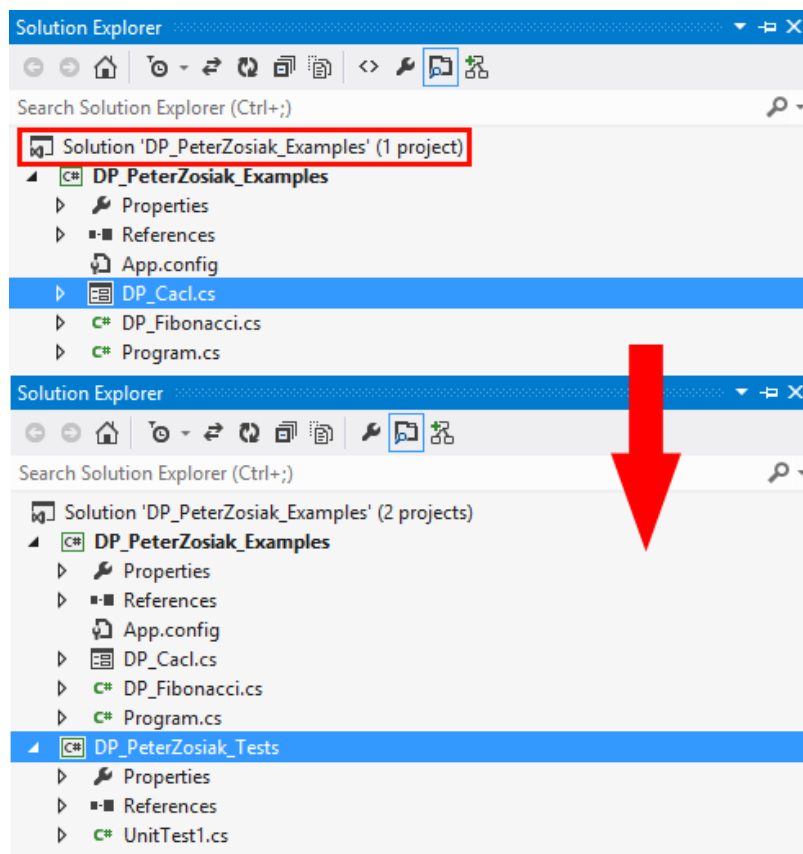
### 6.1 Tvorba Unit testov

Unit testy predstavujú súbor tried a metód, ktoré sa odlišujú od klasických C# tried definíciou atribútov, ktoré ich určujú ako [TestClass] a [TestMethod]. Tieto triedy môžu byť písané ručne alebo môžu byť generované automaticky Visual Studiom, a to výberom z kontextovej ponuky. Všetky testovacie triedy a metódy sú definované Microsoft.VisualStudio.TestTools.UnitTesting menným priestorom (namespace), ktorý je deklarovaný pomocou „using“ na začiatku každej triedy.

K samostatnému testovaniu je potrebný prístup k zdrojovému kódu aplikácie. Ak je tento predpoklad splnený, nasleduje vytvorenie Testovacieho projektu, v ktorom sa budú samotné unit testy vytvárať.

Vytvorenie testovacieho projektu pozostáva z nasledujúcich fáz:

1. V ponuke File je potrebné kliknúť na možnosť Add a následne na možnosť New Project.
2. V dialógovom okne záložka Test.
3. Zo zoznamu šablón je potrebné vybrať Unit Test Project.
4. Po vyplnení názvu a stlačení OK sa vytvorí nový testovací projekt.



Obrázok 9 Vytvorenie Test projektu (Vlastná tvorba)

Vzhľadom k tomu, že sa bude testovať kód v projekte Examples, je potrebné do testovacieho projektu Tests pridať referenciu na projekt Examples, a to kliknutím na položku References a následným zvolením Add References. Tým bude možné v Unit testoch používať objekty (triedy atď.) z projektu Examples. Aby VS mohlo identifikovať triedu ako triedu, ktorá obsahuje Unit testy, a ktorá pre slúži ako kontajner, je potrebné tejto triede priradiť atribút [TestClass]. Napriek tomu, že sa Unit testy vytvárajú ako klasické funkcie/metódy, musia byť označené atribútom [TestMethod]. Unit testy nemajú návratovú hodnotu, musia byť zároveň verejné, nestatické a nemôžu obsahovať žiadne vstupné parametre. Kód samotného Unit testu by mal byť čo najjednoduchší a mal by maximálne využívať testovane objekty.

Unit test pre otestovanie triedy DP\_Fibonacci.cs, ktorá obsahuje len jednu funkciu a vypočíta Fibonacciho postupnosť pre zadané číslo s číselným vstupným parametrom, je možné znázorniť prostredníctvom vzorcov:

```

public class DP_Fibonacci
{
    public static int Fibonacci(int factor)
    {
        if (factor < 2)
            return factor;

        int x = Fibonacci(--factor);
        int y = Fibonacci(--factor);

        return x + y;
    }
}

```

Obrázok 10 Trieda Fibonacci (Vlastná tvorba)

Príslušný unit test.

```

[TestClass]
public class UnitTest1
{
    [TestMethod]
    public void FibonacciTest()
    {
        const int FACTOR = 8;
        int result = DP_Fibonacci.Fibonacci(FACTOR);
    }
}

```

Obrázok 11 Fibonacci unit test (Vlastná tvorba)

Obrázok 11 znázorňuje zápis textu, ktorý síce zavolá funkciu Fibonacci a jej výsledkom bude hodnota, ktorú je ale nutné porovnať, pre overenie správnosti funkcie, s hodnotou očakávanou a podľa výsledku porovnania nastaviť test na Vyhovel (Pass) alebo Nevyhovel (Fail). Na túto úlohu slúži trieda Assert, ktorá je súčasťou TestFrameworku. Trieda Assert obsahuje niekoľko porovnávacích metód:

- **AreEqual a AreNotEqual**

Triedy overujú očakávanú hodnotu s dodanou hodnotou a sú používané pre typy string, double, int, float, object a generic. Obsahujú dva povinné parametre (očakávaná hodnota, skutočná hodnota) a niekoľko voliteľných parametrov (delta pre stupeň nepresnosti napríklad pri počítaní odmocnín, ignoreCase pri porovnaní dvoch string hodnôt kvôli ignorovaní malých a veľkých písmen)

- **AreSame a AreNotSame**

Triedy fungujú rovnako ako AreEqual, s tým rozdielom, že porovnávajú referencie dodaných argumentov. Ak oba argumenty ukazujú na rovnakú inštanciu objektu, test bude vyhodnotený ako Passed.

- **IsTrue a IsFalse**

Triedy overujú, či je porovnávaný výraz boolovskeho typu (true alebo false).

- **IsNull a IsNotNull**

Triedy porovnávajú, či je objektový typ null (má hodnotu) alebo hodnotu nemá.

- **InstanceOfType a IsNotInstanceOfType**

Triedy overujú zadanosť objektu instanciou očakávaného typu.

- **Fail**

Označí okamžite Unit test ako Failed.

- **Inconclusive**

Označí Unit test ako nevyhodnotiteľný, ani Passed, ani Failed.

Pre správne overenie funkcie Fibonacci je nutné do Unit testu doplniť overenie očakávanej hodnoty s hodnotou vygenerovanou funkciou ako je znázornené na obrázku 12.

```
[TestMethod]
public void FibonacciTest()
{
    int FACTOR = 8;
    int EXPECTED = 22;
    int result = DP_Fibonacci.Fibonacci(FACTOR);
    Assert.AreEqual(EXPECTED, result);
}
```

Obrázok 12 Fibonacci unit test s Assert (Vlastná tvorba)

Unit test vytvorený týmto spôsobom je síce správny, ale vždy, keď sa spustí, bude testovať rovnaké hodnoty a výstupom budú rovnaké výsledky. Vo väčšine prípadov je tento spôsob nevyhovujúci, nakoľko testovaním je potrebné pokryť najväčšiu množinu vstupov s reálnymi dátami.

### 6.1.1 Data driven testy - Dátovo riadene testy

Visual Studio poskytuje možnosť, ako automaticky viazať dáta z dátového zdroju (DB, XML, Excel, CSV) ako vstupný parameter unit testu. Tento druh testov je vhodný pre testovanie rovnakej funkcie niekoľkokrát za sebou s rôznymi vstupnými hodnotami z dátového zdroju. Miesto explicitného viacnásobného volania testu vykoná Visual Studio daný test pre každý jeden riadok z dátového zdroju. Dátový zdroj musí byť vytvorený pred spúšťaním testov a musí byť bind-ovaný (linkovaný / viazaný) na

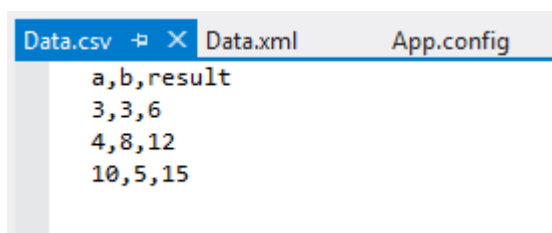
Unit test. Visual Studio podporuje rôzne formáty – CSV, XML, MS Access, MS-SQL databázu, Oracle Database, Excel.

Pre otestovanie funkcie Sčítania znázornenej na obrázku 13 je potrebné vytvoriť CSV v podobnom tvare ako je na obrázku 14.

```
public int Scitanie(int a, int b)
{
    return a + b;
}
```

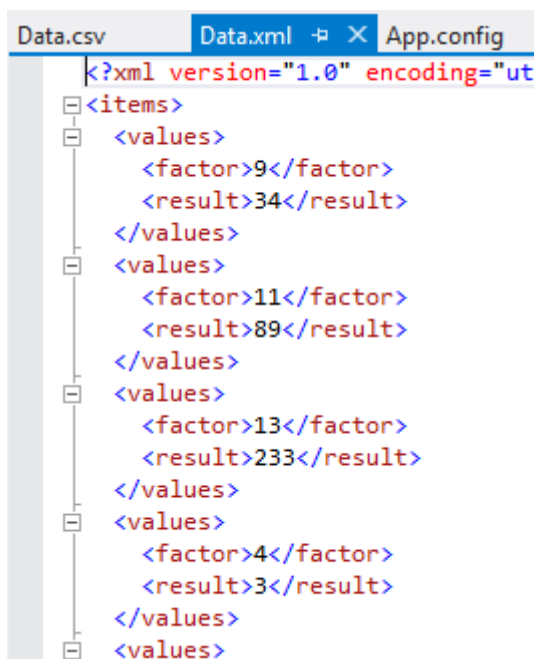
Obrázok 13 Funkcia Sčítania (Vlastná tvorba)

Súbor obsahuje hodnoty pre dve zadané čísla “a” a “b” a očakávaný výsledok „result“.



Obrázok 14 CSV dátový zdroj (Vlastná tvorba)

Ekvivalent dátového zdroja na obrázku 15 zapísaný v XML formáte pre otestovanie funkcie Fibonacci.



Obrázok 15 XML dátový zdroj (Vlastná tvorba)

Po pridaní dátového zdroja do projektu, framework vytvorí objekt TestContext, ktorý uchováva informácie o zdroji dát a nastaví korektnú hodnotu pre každý riadok



dátového zdroju. Pre prístup k týmto hodnotám je potrebné vytvoriť novú inštanciu objektu:

```
private TestContext testContextInstance;
public TestContext TestContext
{
    get { return testContextInstance; }
    set { testContextInstance = value; }
}
```

Obrázok 16 Objekt TestContext (Vlastná tvorba)

Pre nalinkovanie dátového zdroju k testu je potrebné špecifikovať `connectionString` – hodnotu pripojenia a meno tabuľky, v ktorej sú uložené testovacie dáta. Pri definovaní dátového zdroju sa uvádza korektný konektor, ktorý má VS použiť k pripojeniu a čítaniu dát. V prípade použitia CSV je potrebné zvoliť `Microsoft.VisualStudio.TestTools.DataSource.CSV`. Ako tretí voliteľný parameter sa uvádza spôsob prístupu k dátam, ktorý môže byť sekvenčný alebo náhodný. Pre načítanie hodnoty aktuálneho riadku sa používa metóda `testContextInstance.DataRow` riadku pre aktuálnu inštanciu testu. Ak je v dátovom zdroji 5 dátových riadkov, bude existovať presne 5 inštancií tejto triedy.

Výsledná podoba dátovo riadených Unit testov pre funkciu Sčítania

```
[DataSource("Microsoft.VisualStudio.TestTools.DataSource.CSV",
            "|DataDirectory|\\Data.csv", "Data#csv",
            DataAccessMethod.Sequential),
DeploymentItem("Data.csv"),
TestMethod()]
public void ScitanieTest()
{
    DP_Math math = new DP_Math();
    int a = Convert.ToInt16(testContextInstance.DataRow["a"]);
    int b = Convert.ToInt16(testContextInstance.DataRow["b"]);
    int result = Convert.ToInt16(testContextInstance.DataRow["result"]);

    Assert.AreEqual(result, math.Scitanie(a, b));
}
```

Obrázok 17 CSV Data driven unit test (Vlastná tvorba)

Výsledná podoba dátovo riadených Unit testov pre Fibonacci funkciu:

```
[DataSource("Microsoft.VisualStudio.TestTools.DataSource.XML",
           "|DataDirectory|\\Data.xml", "values",
           DataAccessMethod.Random)]
[TestMethod]
public void FibonacciTest()
{
    int FACTOR      = Convert.ToInt16(testContextInstance.DataRow["factor"]);
    int EXPECTED    = Convert.ToInt16(testContextInstance.DataRow["result"]);
    int result      = DP_Fibonacci.Fibonacci(FACTOR);
    Assert.AreEqual(EXPECTED, result);
}
```

Obrázok 18 XML Data driven unit test (Vlastná tvorba)

Dáta vytvorené spôsobom, ako znázorňuje obrázok 18 (tzv. dáta-drivern), sú správne, avšak sú vhodné len pri malých projektoch. Pri väčších projektoch, na ktorých pracuje niekoľko programátorov vo viacerých tímoch, ktorí ku kódu prístupujú z viacerých lokácií, by pri zmene umiestnenia dátového zdroju (zmena serveru, premenovanie súboru atď.) by bolo nutné prepisovať ručne connectionString pre každý unit test zvlášť. Preto je potrebné definovať toto nastavenie v aplikačnom konfiguračnom súbore nazývanom app.config (alternatíva k \*.ini súborom). Tento XML súbor sa defaulte používa pre uchovanie alebo zmenu nastavení (aplikácie, servera atď) aj bez nutnosti rekompilácie aplikácie. Súbor sa štandardne používa pre uchovanie hodnoty connectionString.

App.config do projektu je možné pridávať nasledujúcim spôsobom:

1. V menu je potrebné vybrať položku Project, Add New Item.
2. V zobrazenom dialógovom okne na záložke je potrebné zvoliť Application Configuration File.
3. Názov by mal zostať default, a to app.config, ktorý bude následne pridaný do projektu.

Do novo vytvoreného súboru je potrebné zdefinovať sekciu connectionStrings, ktorá uchováva všetky nastavenia pripojení použitých v aplikácii. V porovnaní s

definovaním connection string v testovacej triede je v tomto prípade nutné definovať aj providerName, čo je typ dátového spojenia. (Obr. 19)

```
<connectionStrings>
  <add name="MyCSVConn"
    connectionString="Driver={Microsoft Text Driver (*.txt; *.csv)};Dbq=c:\;Extensions=asc, csv, tab, txt"
    providerName="System.Data.Odbc" />
</connectionStrings>
<microsoft.visualstudio.testtools>
  <dataSources>
    <add name="MyCSVDataSource" connectionString="MyCSVConn"
      dataTableName="Data#csv" dataAccessMethod="Sequential"/>
  </dataSources>
</microsoft.visualstudio.testtools>
```

Obrázok 19 app.config connectionString (Vlastná tvorba)

Na obrázku 20 je znázornený zápis úpravy atribútov Unit testu, v ktorej sa zmení atribút DataSource a použije názov z app.config je nasledovný:

```
[DataSource("MyCSVDataSource")]
[DeploymentItem("Data.csv")]
[TestMethod()]
public void ScitanieTest()
{
    DP_Math math = new DP_Math();
    int a = Convert.ToInt16(testContextInstance.DataRow["a"]);
    int b = Convert.ToInt16(testContextInstance.DataRow["b"]);
    int result = Convert.ToInt16(testContextInstance.DataRow["result"]);

    Assert.AreEqual(result, math.Scitanie(a, b));
}
```

Obrázok 20 Data driven unit test s app.config (Vlastná tvorba)

### 6.1.2 Špeciálne typy unit testov

Pri tvorbe Unit testov je nutné nakonfigurovať objekt, ktorý budú Unit testy zdieľať. Týmto objektom môže byť databázové pripojenie, logovací súbor a zdieľaná trieda. Unit Test Framework ponúka riešenie na 3 úrovniach: Test, Class a Assembly.

Tabuľka 1 Initialize a Cleanup metódy

Atribút	Frekvencia a úroveň
TestInitialize	Vykonajú sa vždy predtým než sa vykoná ktorýkoľvek unit test
TestCleanup	z danej triedy, resp. potom
ClassInitialize	Vykonajú sa vždy jeden krát, predtým alebo potom čo prebehnú
ClassCleanup	testy v aktuálnej triede
AssemblyInitialize	Vykonajú sa vždy jeden krát, predtým alebo potom čo prebehnú
AssemblyCleanup	testy v ktorejkoľvek z tried Assembly

Obrázok 21 znázorňuje možnú implementáciu Initialize a Cleanup metód v kóde.

```
DP_Math math;
static TextWriter log;
[ClassInitialize]
public static void Initialize()
{
    log = new StreamWriter("logfile.txt");
}
[TestInitialize]
public void TestInitialize()
{
    math = new DP_Math();
    log.WriteLine("Math Class vytvorena");
}

[DataSource("Microsoft.VisualStudio.TestTools.DataSource.CSV",
            "|DataDirectory|\\Data.csv", "Data#csv",
            DataAccessMethod.Sequential),
DeploymentItem("Data.csv"),
TestMethod()]
public void ScitanieTest()
{
    log.WriteLine("Scitanie test");
    int a = Convert.ToInt16(testContextInstance.DataRow["a"]);
    int b = Convert.ToInt16(testContextInstance.DataRow["b"]);
    int result = Convert.ToInt16(testContextInstance.DataRow["result"]);

    Assert.AreEqual(result,math.Scitanie(a, b));
}

[TestMethod()]
public void OdcitanieTest()
{
    log.WriteLine("Odcitanie test");
    Assert.AreEqual(2,math.Odcitanie(5, 3));
}
[TestMethod()]
public void NasobenieTest()
{
    log.WriteLine("Nasobenie test");
    Assert.AreEqual(25,math.Nasobenie(5, 5));
}
[TestCleanup]
public void TestClean()
{
    math = null;
}
[ClassCleanup]
public static void Clean()
{
    log.Close();
}
```

Obrázok 21 Initialize a Cleanup (Vlastná tvorba)

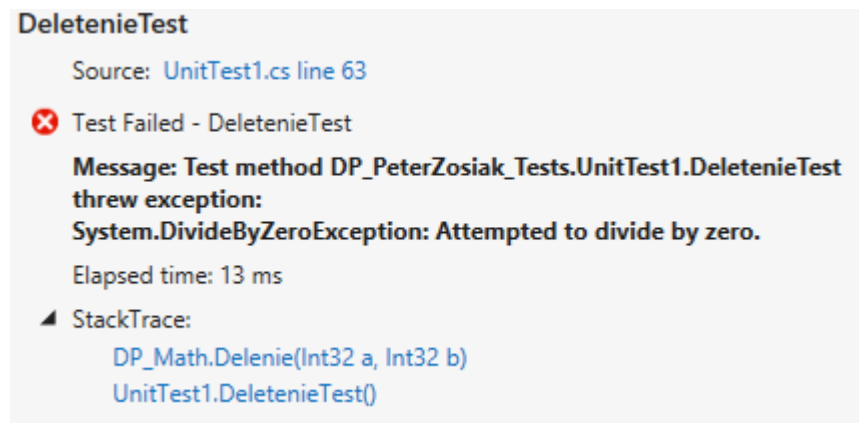
### 6.1.3 Expected Exception - Očakávané výnimky

Vo všeobecnosti sa Unit test, ktorý vygeneruje výnimku (exception) považuje za neúspešný (failed). Napriek tomu je v mnohých situáciách potrebné otestovať false positive, a teda situáciu, kedy je výnimka očakávaná alebo je možné overiť robustnosť systému pri chybe. Príkladom môže byť delenie nulou, pri ktorom sa očakáva, že metóda skončí chybou. Po spustení Unit testu systém zareaguje výnimkou a označí test za neúspešný, ako je znázornené na obrázku 22.

```
[TestMethod()]
public void DeletenieTest()
{
    Assert.AreEqual(25, math.Delenie(10,0));
}
```

Obrázok 22 Delenie test (Vlastná tvorba)

Test skončil neúspešne, pretože sa vyvolala výnimka DivideByZeroException.



Obrázok 23 Unit test s vyvolanou výnimkou (Vlastná tvorba)

Pridaním atribútu `ExpectedException` pred testovaciu metódu bude systém indikovať test ako úspešný len v tom prípade, ak sa počas behu daná výnimka vyskytne. Tento proces znázorňuje obrázok 24.

```
[TestMethod()]
[ExpectedException(typeof(DivideByZeroException))]
public void DeletenieTest()
{
    Assert.AreEqual(25, math.Delenie(10,0));
}
```

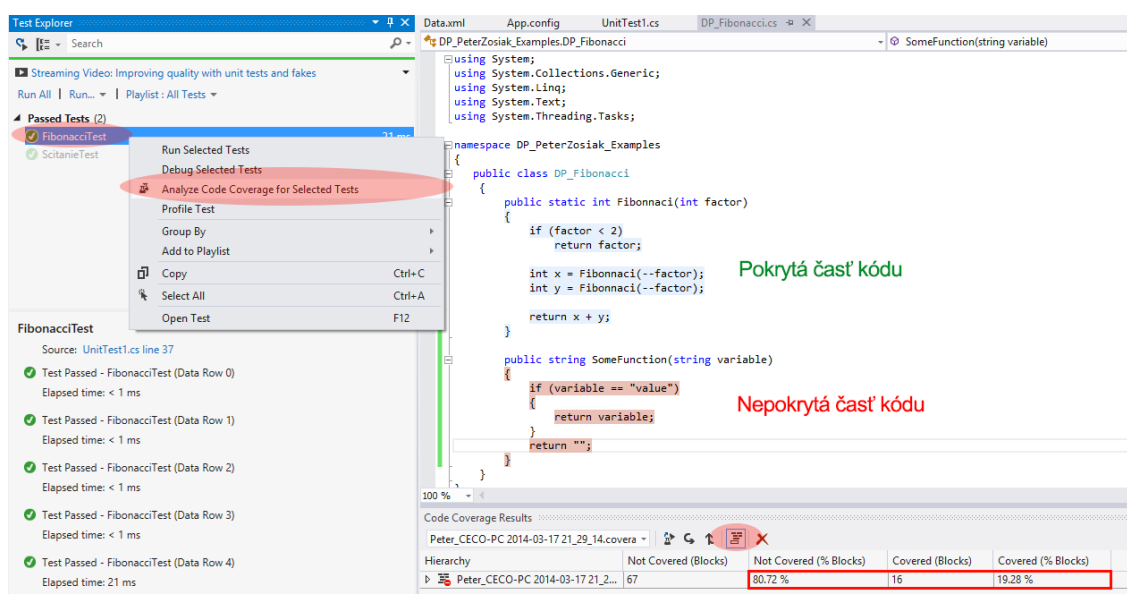
Obrázok 24 ExpectedException (Vlastná tvorba)

Ak sa daná výnimka nevyskytne alebo sa vyskytne výnimka iného typu, bude test označený ako neúspešný.

### 6.1.4 Pokrytie kódu - Code Coverage

Pri procese testovania je dôležité vedieť, koľko kódu je pokryté Unit testami. Nástroj Code Coverage poskytuje prehľad, aké percento kódu je pokryté testami, pričom nepokryté časti farebne zvýrazní. Nízke pokrytie znamená, že určité časti aplikačného kódu nie sú testované.

Na rozdiel od predošlých verzií VS bola táto funkcia zjednodušená a upravená tak, aby poskytovala čo najviac informácií ohľadom pokrytia. Zároveň bola rozšírená jej podpora aj na unmanaged (nemanajovaný) kód. Funkcia je dostupná v Test Explorer okne a je možné ju vyvolať kontextovou ponukou pri kliknutí na daný test. Pre zistenie pokrytia celého projektu je nutné označiť všetky vytvorené unit testy. Po analýze sa zobrazí Code Coverage Result – výsledok pokrytia kódu.



Obrázok 25 Code Coverage (Vlastná tvorba)

Namerané výsledky možno exportovať alebo importovať z predošlých meraní pre lepšie porovnanie progresu testovania. Ak je potrebné vyňať určitú funkciu z analýzy pokrytia, definuje sa atribút ExcludeFromCoverage pred deklaráciou metódy (Obrázku 26).

```

[System.Diagnostics.CodeAnalysis.ExcludeFromCodeCoverage]
public string SomeFunction(string variable)
{
    if (variable == "value")
    {
        return variable;
    }
    return "";
}

```

Obrázok 26 ExcludeFromCodeCoverage (Vlastná tvorba)

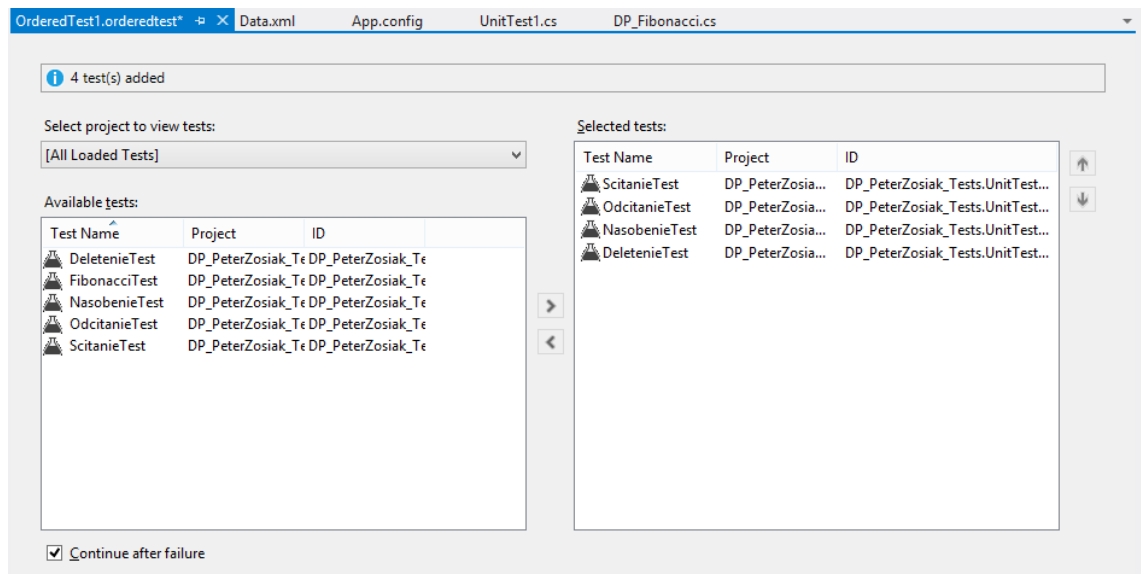
## 6.2 Tvorba Ordered testov

Ordered test je možné charakterizovať ako akýsi kontajner, v ktorom sa vytvárajú testovacie sady Unit testov, zoradených do logických častí, aby tvorili sled udalostí vyskytujúcich sa v aplikácii, tzv. Scenárov (Scenarios). Podstatou Ordered testov je, že Unit testy prebehnú v konkrétnom poradí len raz. Pre zostavovanie Scenárov je potrebná znalosť aplikácie, použitých funkcií a logiky správania. Tvorba samotného Ordered testu je pomerne jednoduchá. Na pravej strane sa nachádzajú všetky testy vytvorené v Testovacím projekte a na strane ľavej sú pridané Unit testy (z pravej strany), ktoré sa majú vykonať v správnom poradí. Primárnou vlastnosťou tohto testu je možnosť pokračovať po neúspešnom teste "Continue after failure", a teda či sú alebo nie sú testy na sebe striktné závislé usporiadaním. Ak jeden z Unit testov zlyhá, ostatné Unit testy, ktoré nasledujú po ňom, sa vykonajú naďalej.

Vytvorenie Ordered testu sa vykonáva nasledovne:

1. V Solutions Explorer okne je potrebné vybrať testovací projekt.
2. Pravým kliknutím myši vybrať a označené Add Ordered Test.

Ordered test sa vytvorí v rámci testovacieho projektu a následne sa zobrazí okno s nastavením. V tomto okne je možné vyberať z už existujúcich testov a usporiadať ich podľa logickej nadväznosti. (Obrázok 27)

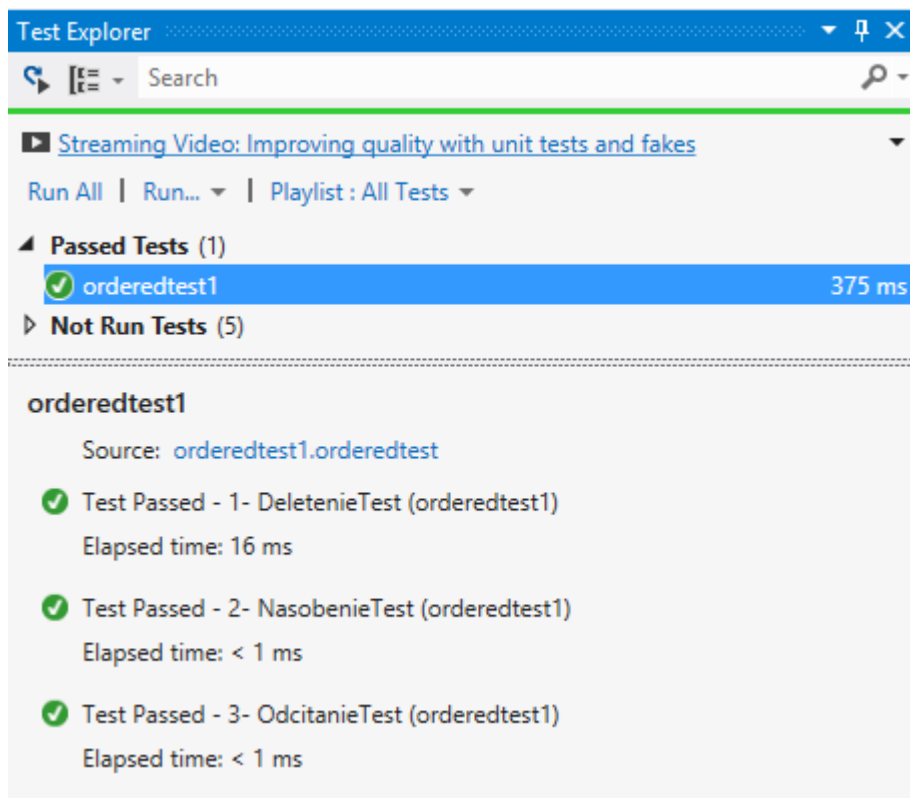


**Obrázok 27 Ordered Test (Vlastná tvorba)**

Prvý riadok zobrazuje počet vybraných a vložených testov v Ordered teste. Select project to view tests je zoznam pre výber testovacieho projektu, z ktorého sa budú vkladať testy. Okno je rozdelené na dve časti – v ľavej časti sa nachádzajú všetky testy daného testovacieho projektu a v pravej časti sú už vybrané testy, ktoré sú použité v Ordered teste. Podľa toho, v akom poradí sa má test vykonať, je možné tieto testy posúvať smerom nahor a nadol. Posledná možnosť je Continue after failure, ktorá umožňuje pokračovať vo vykonávaní zvyšných testov, a to v prípade, že predošlý test zlyhá alebo je neúspešný.

Ordered testy sa spúšťajú ako zvyšné testy z Test Explorer okna výberom príslušného testu – po vyvolaní kontextovej ponuky pravým tlačidlom a následnom voľbou Run Selected test. Po skončení testov sa zobrazí okno s výsledkami pre každý jeden test individuálne.





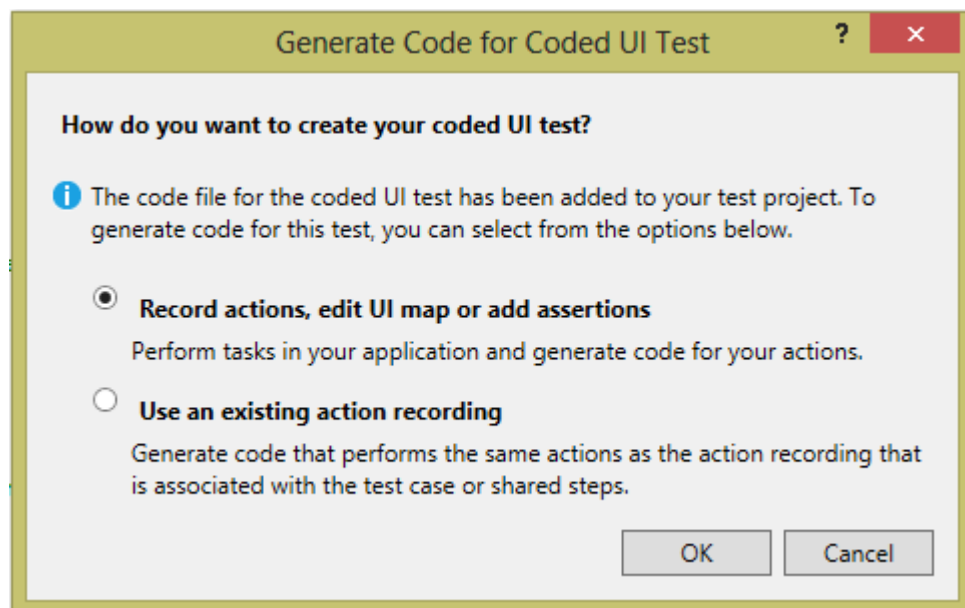
Obrázok 28 Výsledky Ordered testu (Vlastná tvorba)

### 6.3 Tvorba CodedUI Testov

Coded UI Test (CUIT) je automatizovaný spôsob, ako testovať užívateľské rozhranie (UI – user interface) aplikácie. UI sa vo väčšine prípadov testuje ručne, kedy tester vykonáva sadu krokov, popísaných v testovacom prípade. Negatívom manuálnych testov je potreba ľudského faktora pri spúšťaní, vykonávaní a následnej validácii testu. Pri malých aplikáciách je tento faktor zanedbateľný, ale pri väčších projektoch sa stáva manuálne testovanie nákladným, či už po finančnej stránke alebo po stránke zaťaženia ľudských zdrojov. Aby sa tento proces čo najviac zautomatizoval, VS 2012 ponúka podporu tzv. kódových testov UI (user interface = užívateľské rozhranie), ktorých účelom je testovanie funkcií a objektov užívateľského rozhrania. Tento druh testov poskytuje komfortný mechanizmus pre vykonanie a overenie určitého testovacieho prípadu (test case) v UI. Test simuluje užívateľa, ktorý ovláda aplikáciu pomocou klávesnici a myši. Tento test je možné nahrat' (zaznamenať) alebo naprogramovať tak, aby kedykoľvek počas priebehu testovania overoval prvky užívateľského rozhrania (tlačidlá, textové polia atď.), ich správanie a korektné zobrazovanie.

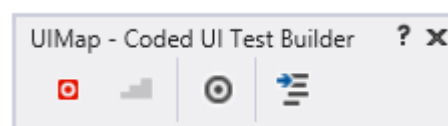
Vytvorenie nového CUIT je možné v okne Solutions Explorer, a to stlačením pravého tlačidla myši na Testovacom projekte a následnom zvolení možnosti Add. Po tomto úkone sa zobrazí okno s ponukou nových typov testov, z ktorých sa vyberie Coded UI Test.

Po potvrdení nového Coded UI testu sa na obrazovke zobrazí okno s povinným výberom spôsobu generovania testu. Pri vytváraní nových testov je potrebné zvoliť možnosť Record actions, edit UI Map or add assertion. Táto voľba umožňuje zaznamenávať test od začiatku spôsobom, ako by sa v aplikácii správal reálny užívateľ.



Obrázok 29 Generovanie Coded UI Test (Vlastná tvorba)

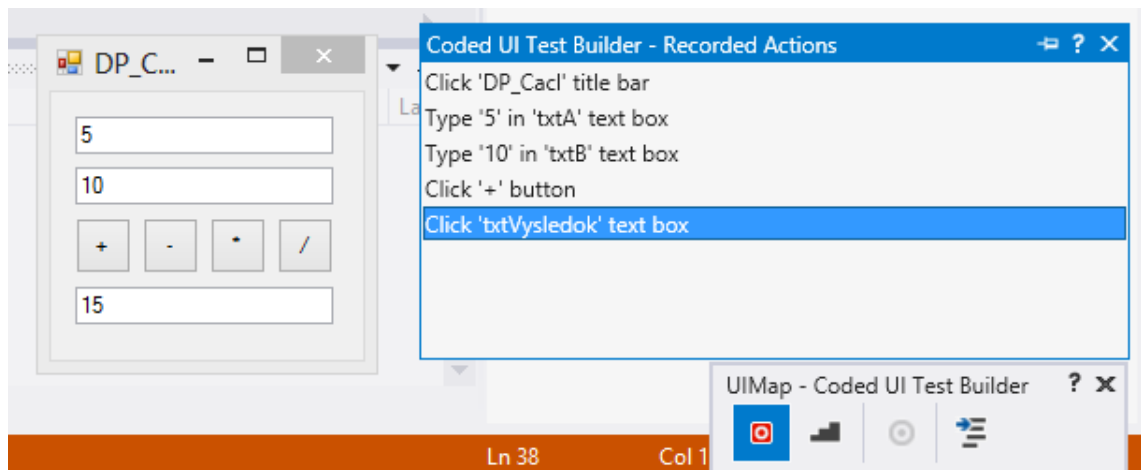
V pravom dolnom rohu obrazovky sa zobrazí Coded UI Test Builder, čo je v podstate robot, ktorý zaznamenáva všetky vykonané akcie, identifikuje všetky objekty a prvky užívateľského rozhrania, na ktoré sa počas záznamu testu klikne a taktiež ich vlastnosti, ktoré sa budú verifikovať.



Obrázok 30 Coded UI Test Builder (Vlastná tvorba)

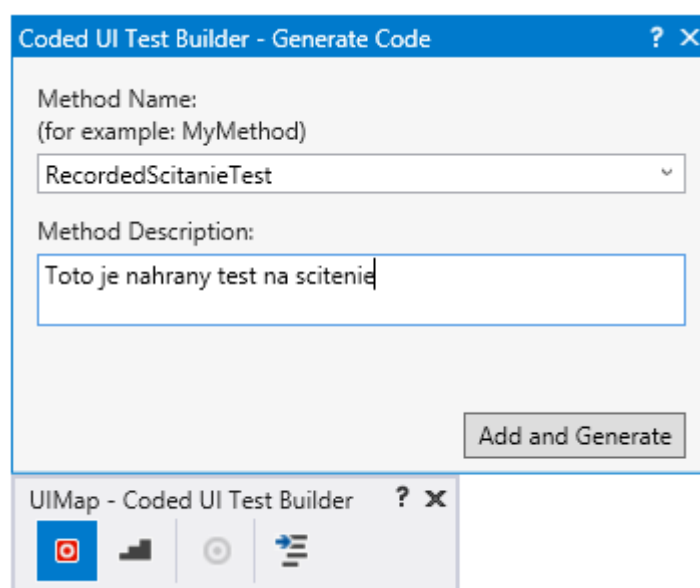
Následne sa vykonáva súbor krokov, ktorými sa otestuje daná funkcionálna GUI a podobne. Nahrávanie je možné kedykoľvek stopnúť a zároveň je možné sledovať

a vizuálne kontrolovať všetky vykonané akcie, a to stlačením tlačidla Show Recorder steps. Je potrebné pripomenúť, že počas nahrávania testu Test Builder zachytí všetky akcie v systéme, bez ohľadu na to, či sa týkajú testovanej aplikácie. Ak sa počas nahrávania spustí iná aplikácia alebo je vyžadovaná interakcia užívateľa, zaznamenaná sa aj tento bod, dôsledkom čoho môžu vzniknúť chyby pri neskoršom regresnom prehrávaní testu. Na obrázku 31 sú znázornené zaznamenané kroky, kliknutie do textového poľa, zadanie hodnôt a kliknutie na tlačidlo “+”.



Obrázok 31 Zaznamenané kroky CUIT (Vlastná tvorba)

Po kontrole zaznamenaných krokov je možné vygenerovať kód, s ktorým sa v testoch bude pracovať.



Obrázok 32 Generovanie kódu (Vlastná tvorba)

CUIT Builder generuje niekoľko pomocných súborov používaných v rámci testovania:

- *UIMap*: Reprezentuje zoznam ovládacích prvkov, okien, tlačidiel a pod. Pomocou týchto prvkov je možné vykonávať akcie (klik tlačidla, myši) potrebné pre automatizáciu testu.
- *CodedUITest.cs*: Obsahuje testovaciu triedu, testovacie metódy a Assert metódy.
- *UIMap.uitest*: XML model pre UIMap triedu.
- *UIMap.cs*: Upravovateľný kód pre UIMap.

```
using Microsoft.VisualStudio.TestTools.UITesting;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Microsoft.VisualStudio.TestTools.UITest.Extension;
using Keyboard = Microsoft.VisualStudio.TestTools.UITesting.Keyboard;

namespace DP_PeterZosiak_Tests
{
    [CodedUITest]
    public class CodedUITest1
    {
        public CodedUITest1()
        {
        }

        [TestMethod]
        public void CodedUITestMethod1()
        {
            this.UIMap.RecordedScitienieTest();
            this.UIMap.AssertMethod1();
        }

        Additional test attributes
        public TestContext TestContext
        {
            get
            {
                return testContextInstance;
            }
            set
            {
                testContextInstance = value;
            }
        }
    }
}
```

Obrázok 33 Trieda CodedUITest (Vlastná tvorba)

### 6.3.1 UIMap.Designer.cs

CUIT Builder automaticky vytvorí kód pre daný nahraný (recorded) test. Súbor a príslušný kód sa aktualizujú vždy pri úprave testu. Všetky triedy v tomto súbore sú generované automaticky, a preto má každá trieda priradený atribút `GeneratedCode`. Trieda `UIMap` obsahuje komplexnú definíciu metód, ktoré boli identifikované počas nahrávania všetkých testov. Definícia každej metódy má preddefinovanú štruktúru, ktorá obsahuje definíciu premenných (GUI objektov), volanie metód (`Click`) a nastavovanie vlastnosti (`Text`).

```
public void RecordedScitienieTest()
{
    #region Variable Declarations
    WinClient uIDP_CaclClient = this.UIDP_CaclWindow.UIDP_CaclClient;
    WinEdit uITxtAEdit = this.UIDP_CaclWindow.UITxtAWindow.UITxtAEdit;
    WinEdit uITxtBEdit = this.UIDP_CaclWindow.UITxtBWindow.UITxtBEdit;
    WinButton uIItemButton = this.UIDP_CaclWindow.UIItemWindow.UIItemButton;
    WinEdit uITxtVysledokEdit = this.UIDP_CaclWindow.UITxtVysledokWindow.UITxtVysledokEdit;
    #endregion

    // Click 'DP_Cacl' client
    Mouse.Click(uIDP_CaclClient, new Point(322, 205));

    // Type '5' in 'txtA' text box
    uITxtAEdit.Text = this.RecordedScitienieTestParams.UITxtAEditText;

    // Type '10' in 'txtB' text box
    uITxtBEdit.Text = this.RecordedScitienieTestParams.UITxtBEditText;

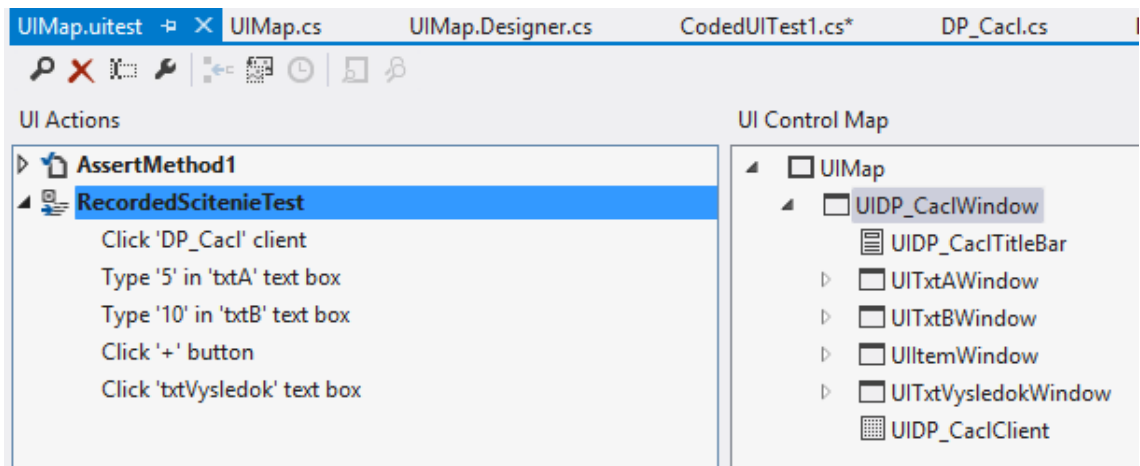
    // Click '+' button
    Mouse.Click(uIItemButton, new Point(13, 19));

    // Click 'txtVysledok' text box
    Mouse.Click(uITxtVysledokEdit, new Point(44, 6));
}
```

Obrázok 34 GUI Objekt vyjadrený kódom (Vlastná tvorba)

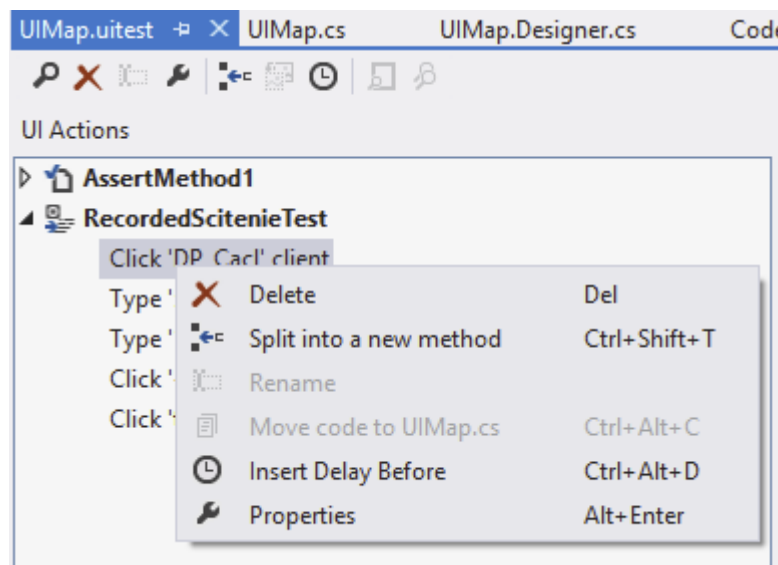
### 6.3.2 UiMap.uitest

XML súbor, ktorý reprezentuje štruktúru zaznamenaných CUIT, obsahuje definíciu všetkých metód, vlastností a akcií, ktoré vygeneroval Coded UI Test Builder. Vzhľadom k tomu, že je štruktúra generovaná automaticky, nie je vhodné ju upravovať priamo v kóde ale pomocou `UIMap` editora, pretože pri zmene záznamu testu alebo ovládacieho objektu sa XML re-generuje a vlastný kód sa prepíše.



Obrázok 35 UIMap Editor (Vlastná tvorba)

Editor obsahuje voľby pre zmazanie alebo premenovanie metódy, nastavenie vlastností, rozdelenie do novej metódy alebo presun kódu do novej UIMap. Aby bola simulácia užívateľa reálna, zároveň je možné v editore vkladať omeškanie medzi jednotlivými krokmi.

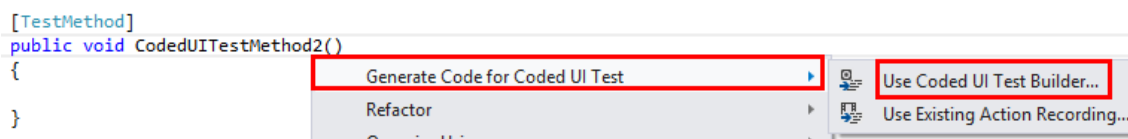


Obrázok 36 UIMap editácia (Vlastná tvorba)

### 6.3.3 Pridanie Assert

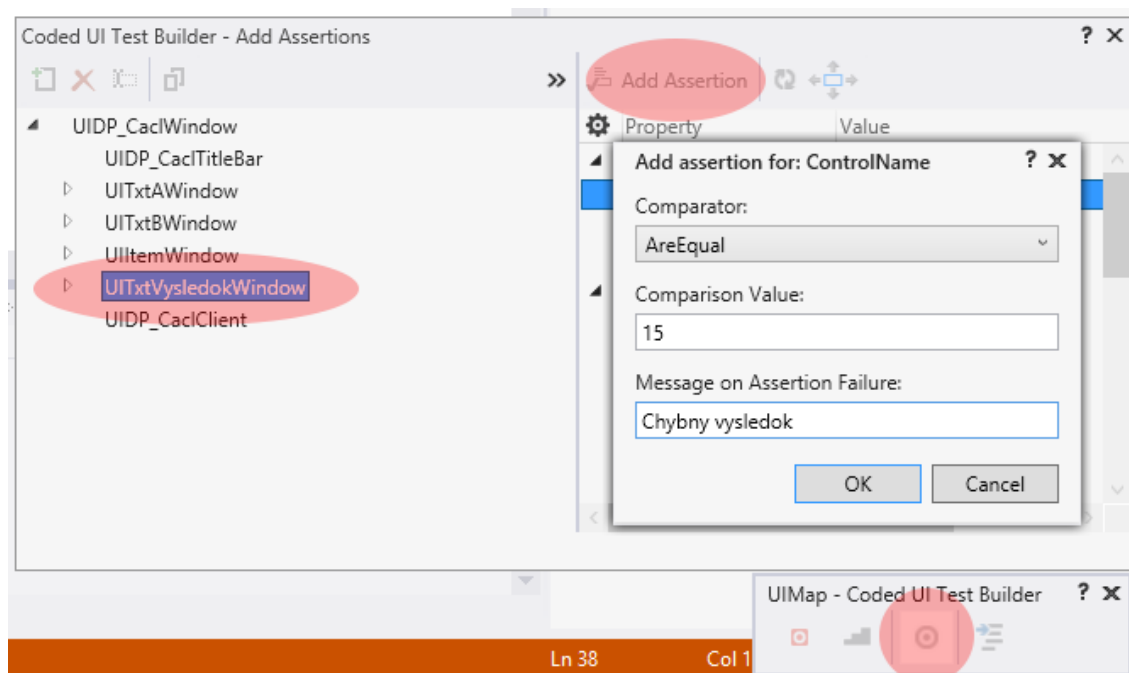
Podobne ako v unit testoch aj pri CUIT je nutný určitý typ validácie, či už na existenciu UI objektov alebo hodnôt v nich obsiahnutých. Pre pridanie validačného Assert do už vytvoreného CUIT je potrebné kliknúť pravým tlačidlom myši na názov

testovacej metódy a z kontextovej ponuky vybrať možnosť Generate Code for Coded UI Test, prípadne použiť klávesovú skratku „Ctrl \“ + „Ctrl C“



Obrázok 37 Štart CUIT Builder-u (Vlastná tvorba)

Po zobrazení CUIT Builder-u je potrebné zvoliť „Add assertion“ tlačidlo, ktoré má podobu „crosshair“. Následne sa zobrazia všetky zaznamenané objekty v UIMap, z ktorých je možné vybrať práve ten, ktorého hodnotu je potrebné overiť. Napriek tomu, že validovať možno všetky vlastnosti (Properties) objektu, väčšinou sa validuje hodnota daného objektu, v prípade textboxu sa overuje vlastnosť Text. V poslednom kroku sa nastavuje Comparison Value (očakávaná hodnota na porovnanie) a taktiež text chybového hlásenia (ak sa hodnota líši od očakávanej).



Obrázok 38 Pridanie Assert-u (Vlastná tvorba)

Po potvrdení sa Assert metóda automaticky pridá do UIMap súboru a taktiež do vybranej testovacej metódy. Je k nej avšak možné pristupovať z ktoréhokolvek testu práve cez objekt UIMap.

```
[TestMethod]
public void CodedUITestMethod1()
{
    this.UIMap.RecordedScitenieTest();
    this.UIMap.AssertMethod3();
}
```

Obrázok 39 Assert v kóde (Vlastná tvorba)

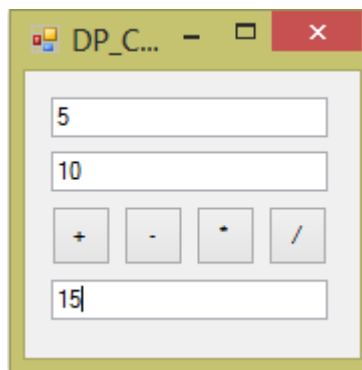
Assert sa zároveň vygeneruje aj v objekte UIMap.

```
/// <summary>
/// Scitanie assert
/// </summary>
public void AssertMethod3()
{
    #region Variable Declarations
    WinEdit uITxtVysledokEdit = this.UIDP_CaclWindow.UITxtVysledokWindow.UITxtVysledokEdit;
    #endregion

    // Verify that the 'Text' property of 'txtVysledok' text box equals '15'
    Assert.AreEqual(this.AssertMethod3ExpectedValues.UITxtVysledokEditText, uITxtVysledokEdit.Text, "Chybny vysledok scitania");
}
```

Obrázok 40 Assert v UIMap (Vlastná tvorba)

Test po spustení automaticky nájde zobrazenú aplikáciu a postupne vyplní jej ovládacie prvky hodnotami, ktoré sa zadali pri nahrávaní (recording) testu, vykoná akcie a skontroluje hodnotu výsledku.



Obrázok 41 Automaticky vyplnené hodnoty v aplikácii (Vlastná tvorba)



### 6.3.4 Data driven CUIT

Pri Datadriven CUIT sa používa rovnaký postup ako pri data driven uni testoch (kapitola 6.1.1.). Dátový zdroj sa nalinkuje na testovaciu metódu. Hodnota datarow pre príslušný stĺpec dátového zdroja sa uloží do premennej, ktorou sa naplní vlastnosť (property) „Text“ príslušného objektu. V poslednom kroku sa prepíše očakávaná hodnota v metóde AssertMethod3, nahraná pri vytvorení testu hodnotou načítanou z dátového zdroja.

```
[DataSource("Microsoft.VisualStudio.TestTools.DataSource.XML",
            "|DataDirectory|\\Data.xml", "add",
            DataAccessMethod.Sequential),
TestMethod]
public void CodedUITestMethod1()
{
    string a = this.testContextInstance.DataRow["a"].ToString();
    string b = this.testContextInstance.DataRow["b"].ToString();
    string result = this.testContextInstance.DataRow["result"].ToString();

    this.UIMap.UIDP_CaclWindow.UITxtAWindow.UITxtAEdit.Text = a;
    this.UIMap.UIDP_CaclWindow.UITxtBWindow.UITxtBEdit.Text = b;

    this.UIMap.AddAction();

    this.UIMap.AssertMethod3ExpectedValues.UITxtVysledokEditText = result;
    this.UIMap.AssertMethod3();
}
```

Obrázok 42 Data driven Coded UI Test (Vlastná tvorba)

### 6.3.5 Spustenie CUIT z príkazového riadku

V súčasnosti existuje niekoľko majoritných verzií operačného systém Windows, pre ktoré Microsoft vydáva aktualizácie. Aktualizácia každého systému prirodzene mení už zavedené systémy. Zmeny môžu byť malé i veľké a každá jedna môže ovplyvniť chod aplikácie. Preto je často potrebné, aby sa užívateľské rozhranie testovalo na majoritnej časti operačných systémov (Windows 7, Windows 8 a pod.). Aby sa na každý systém nemusela inštalovať celá produktová sada Visual Studio, na ktorú sa vzťahuje licenčná politika, je možné púšťať CUIT priamo z príkazového riadku. Ako „emulátor“ posluží nástroj MSTest, ktorý je bezplatne stiahnuteľný z webových stránok spoločnosti Microsoft.

Program MSTest.exe slúži na spúšťanie automatizovaných testov v testovacej Assembly z príkazového riadku. Prostredníctvom tohto programu je možné prehliadať testovacie výsledky, ukladať ich na disk alebo do TFS (Team Foundation Server).

Pred spustením testov je potrebná ich DLL Assembly, teda “zbuildovaná” verzia, ktorá sa nachádza v Debug priečinku testovacieho projektu. Tento DLL súbor môže byť umiestnený ľubovoľne na disku akéhokoľvek systému (virtuálne PC a pod.).

Nástroj sa default nachádza v adresári “C:\Program Files (x86)\Microsoft Visual Studio 11.0\Common7\IDE\MSTest.exe” a má niekoľko parametrov (Tabuľka 2).

Tabuľka 2 Parametre pre MSTest.exe

Parameter	Popis
/testcontainer	Súbor obsahujúci testy napr.: /testcontainer:tests.dll
/testmetadata	Súbor obsahujúci testovacie metadáta
/testlist	Špecifikovanie testovacieho zoznamu, ktorý sa zadaný v test metadáta
/test	Špecifikácia názvu testu, ktorý ma byť spustený
/runconfig	Špecifikácia konfiguračného súboru napr.: /runconfig:localtestrun.Testrunconfig
/resultsfile	Uloženie výsledkov do súboru napr.: /resultsfile:testResults.trx
/unique	Test sa spusti iba vtedy ak je nájdená unikátna zhoda s názvom
/detail	Špecifikovanie názvů vlastností, pre ktorú je nutné zobrazit hodnoty
/help	Zobrazenie nápovede
/nologo	Nezobrazí sa copyright sprava na začiatku behu testov

Príslušné testy sa spustia otvorením príkazového riadku a zadaním príkazu, v ktorom sa naprv definuje cesta k MSTest.exe, následne cesta k DLL súboru s testami a nakoniec názov samotného testu. Za predpokladu, že by bolo dané DLL v koreňovom „C“ adresári, príkaz na spustenie testov by vypadal nasledovne:

```
"c:\Program Files (x86)\Microsoft Visual Studio 11.0\Common7\IDE\MSTest.exe"
/testcontainer:"c:\DP_PeterZosiak_Tests.dll" /test:CodedUITestMethod2
```

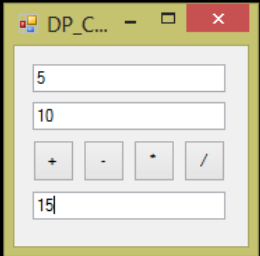
```
c:\>"c:\Program Files (x86)\Microsoft Visual Studio 11.0\Common7\IDE\MSTest.exe" /testcontainer:"c:\
DP_PeterZosiak_Tests.dll" /test:CodedUITestMethod2
Microsoft (R) Test Execution Command Line Tool Version 11.0.50727.1
Copyright (c) Microsoft Corporation. All rights reserved.

Loading c:\DP_PeterZosiak_Tests.dll...
Starting execution...

Results
-----
Top Level Tests
-----
Passed DP_PeterZosiak_Tests.CodedUITest1.CodedUITestMethod2
1/1 test(s) Passed

Summary
-----
Test Run Completed.
Passed 1
-----
Total 1
Results file: c:\TestResults\Peter_CECO-PC 2014-04-01 20_36_54.trx
Test Settings: Default Test Settings

c:\>
```



Obrázok 43 Spustenie CUIT z príkazovej riadky (Vlastná tvorba)

## 6.4 Tvorba Web Performance Testov

Web Performance tests (ďalej len WPT) sa používajú na testovanie funkcionality a výkonnosti webových stránok, webových aplikácií, webových služieb alebo ich vzájomnej kombinácii. WPT môže byť vytvorený zaznamenaním HTTP request-u (požiadavky) alebo akcie počas interakcie užívateľa s webovou aplikáciou. Zaznamenáva sa taktiež presmerovanie (redirect), validácie, view state, autentifikačné informácie a podobne.

WPT obsahuje niekoľko rozličných validačných pravidiel, ktoré sa používajú na validáciu názvov, hodnôt formulára, textov, tagov a i. na požadovanej webovej stránke a niekoľko extrakčných pravidiel, ktoré slúžia na zber dát z webovej stránky, používaných na overenie reálnej hodnoty s hodnotou očakávanou.

WPT poskytuje rôzne spôsoby overovania webových stránok pre každý request a response (požiadavka a odpoveď), v ktorých je možné simulovať rôzne situácie ako pomalá rýchlosť siete, rôzne prehliadače a ich verzie a rôzny počet užívateľov v danom okamihu. Všetky tieto faktory vplývajú na výkonnosť webových stránok a dobu odozvy a aj prostredníctvom nich je možné identifikovať problémy.

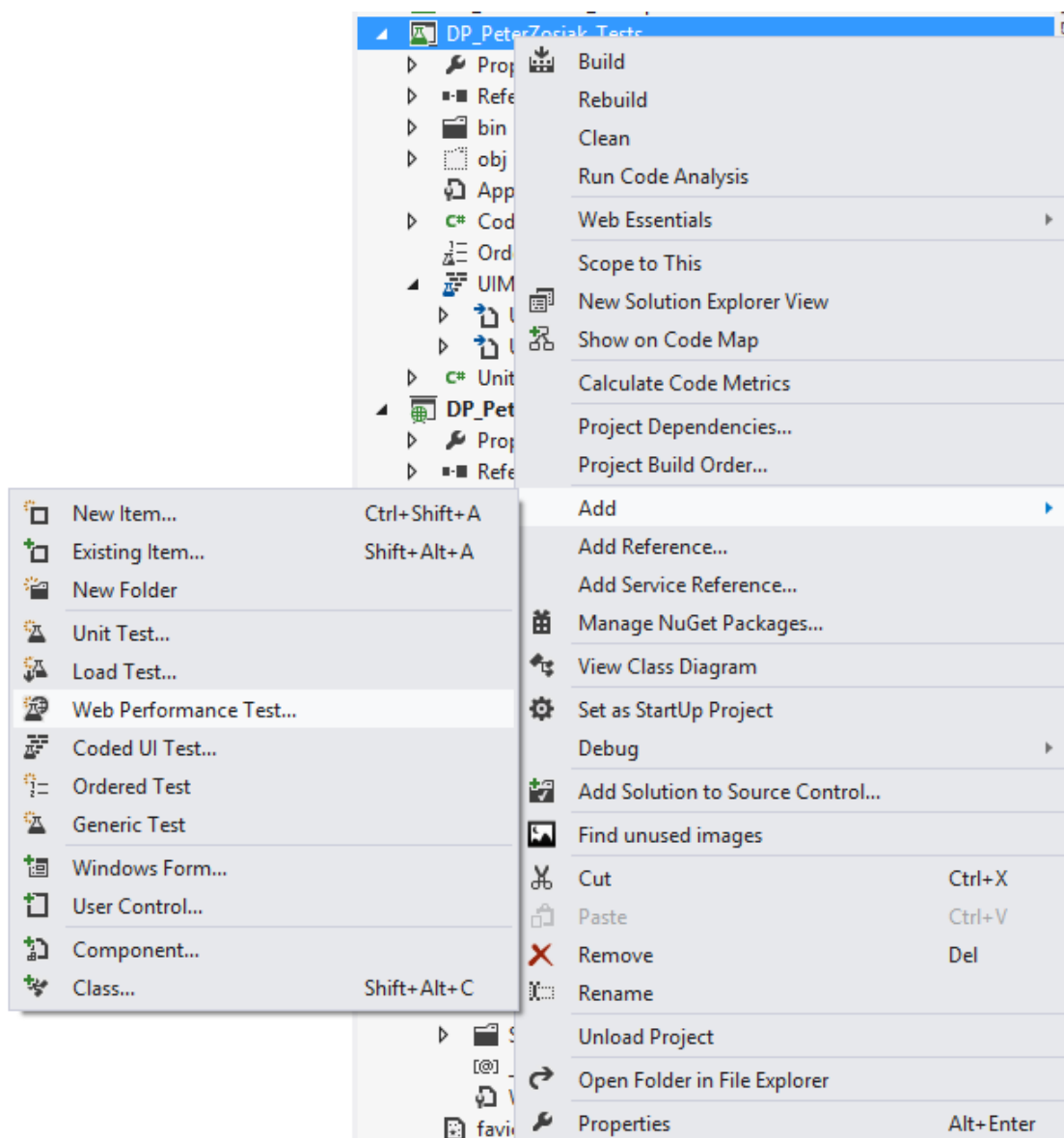
WPT možno využiť ako:

- Validačné a verifikačné testy pomáhajú overiť vstupy a očakávané výstupy
- Usability testy (Testy Použitelnosti), ktoré simulujú reálneho užívateľa, jeho spôsob používania aplikácie, klikanie na odkazy, overovanie zobrazovania obsahu, dialógov a podobne.

- Security testy (Bezpečnostné testy) pomáhajú overiť ako webová aplikácia reaguje na rôznych užívateľov, v rôznych užívateľských skupinách a s rôznym stupňom prístupu (admin., registrovaný užívateľ, neregistrovaný užívateľ), ktorí môžu pristupovať buď z lokálnej siete, domény alebo inej siete.
- Performance testy (Výkonnostné testy) overujú rýchlosť a stabilitu webovej aplikácie pri zväčšenej záťaži a pri simulácii rôznych rýchlostí siete.
- Compatibility testy (Testy Kompatibility), kedy sa webová aplikácia testuje v rôznych druhoch prehliadačov v rôznych verziách.

#### **6.4.1 Tvorba Web Performance Testu**

Web Performance Test aktivuje Web Performance Test Recorder, ktorý zaznamená všetky akcie vykonané počas prezerania webovej stránky a následne ich pridá do samotného testu. Vytvorenie nového CUIT je možné v okne Solutions Explorer, a to stlačením pravého tlačidla myši na Testovacom projekte a následnom zvolení možnosti Add – zobrazí sa okno s ponukou nových typov testov, z ktorých sa vyberie Web Performance Test.



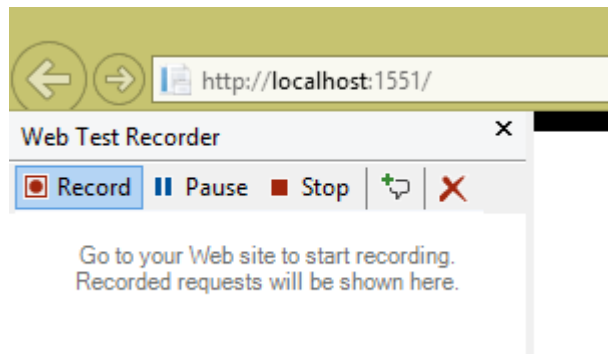
Obrázok 44 Vytvorenie Web Performance Testu (Vlastná tvorba)

V momente, keď sa zvolí možnosť Web Performance Test a kľime sa na tlačidlo OK, vytvorí sa nový test v Testovacom projekte a nová inštancia prehliadača (Internet Explorer), ktorý obsahuje Web Test Recorder. Ten slúži na zaznamenanie akcií užívateľa pri prehliadaní webovej stránky, pretože zaznamenáva všetky request-y a response.

Recorder má nasledujúce funkcie:

- *Record* – spustí nahrávanie.
- *Pause* – zastaví nahrávanie, pretože v niektorých prípadoch nie je potrebné zaznamenať všetky request-y.

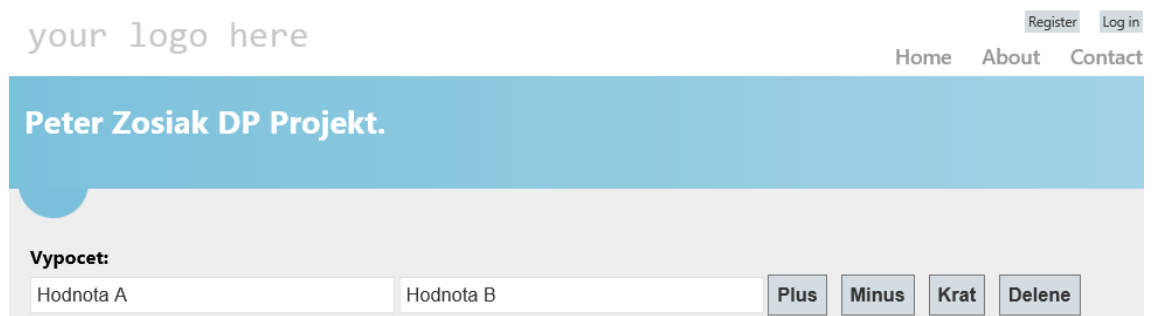
- *Stop* – zastaví nahrávanie, zatvorí prehliadač a ukončí nahrávaciú reláciu (recording session).
- *Add a Comment* – pridá komentár k danej nahrávacej relácii.
- *Clear all requests* – vymaže všetky zaznamenané požiadavky v zázname.



Obrázok 45 Web Test Recorder (Vlastná tvorba)

Po otvorení prehliadača je potrebné zadať príslušnú URL adresu, či už lokálnu (<http://localhost:1551>) alebo verejnú ([www.nieco.sk](http://www.nieco.sk)). Zobrazí sa stránka a pokračuje sa podľa vopred pripraveného testovacieho prípadu, zadávajú sa hodnoty a vykonávajú sa akcie s očakávanými výsledkami.

Po spustení aplikácie a stlačení odkazu Login sa zobrazí prihlasovací formulár, kde sa vyplnia prihlasovacie údaje a aplikácia sa presmeruje na úvodnú stránku, kde je možné zadať hodnoty na výpočet. Po vložení všetkých hodnôt (v tomto prípade zadaní dvoch hodnôt) a stlačení príslušného tlačidla funkcie sa web presmeruje na stránku s výsledkom.



Obrázok 46 Ukážka webovej stránky s formulárom (Vlastná tvorba)

## Vysledok:

5

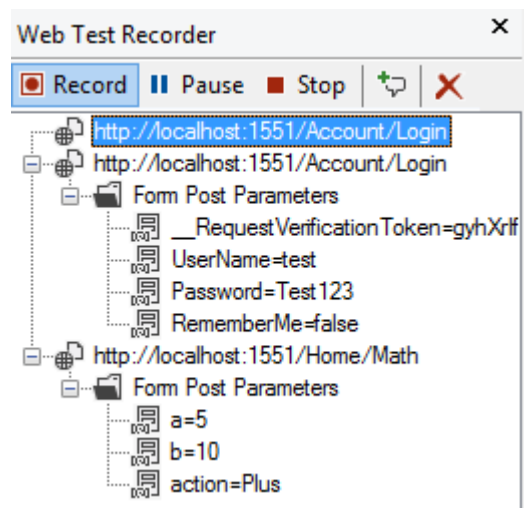
Obrázok 47 Ukážka webovej stránky s výsledkom (Vlastná tvorba)

HTML kód pre dané zobrazenie je znázornený na obrázku 48.

```
<h2 class="Info" id="Info">Vysledok:</h2>  
<input type="text" name="result" value="@ViewBag.result" />
```

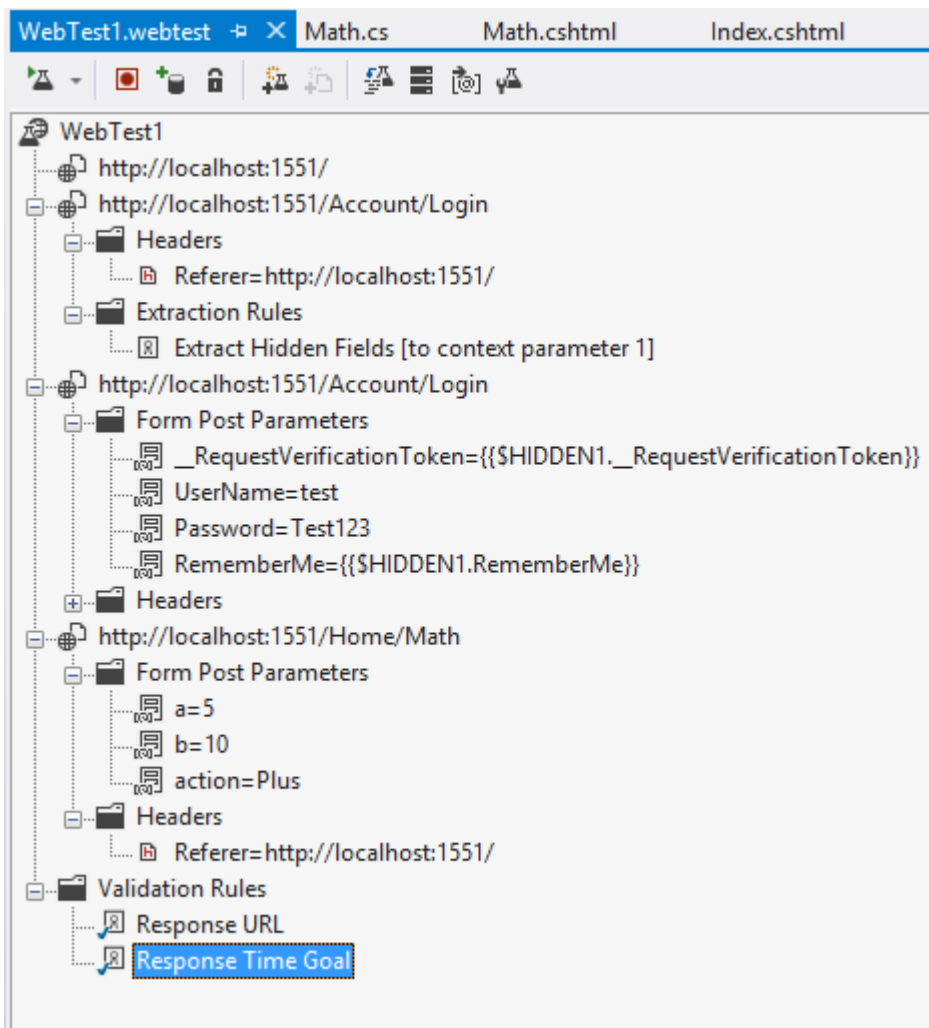
Obrázok 48 Kód stránky s výsledkom (Vlastná tvorba)

Vo Web Test Recorder sú zaznamenané všetky vykonané akcie.



Obrázok 49 Zaznamenané kroky Web Test Recorder-u (Vlastná tvorba)

Stlačením tlačidla Stop sa zatvorí Web Test Recorder aj Internet Explorer a vygeneruje sa webtest vo VS, v ktorom sú zobrazené všetky stránky, akcie a parametre zaznamenané pri nahrávaní. Po skončení nahrávania (tlačidlom STOP) Visual Studio vygeneruje Web Performance Test, ktorý je v editačnom režime a je možné ho ďalej upravovať.



Obrázok 50 Detail Web Performance Testu (Vlastná tvorba)

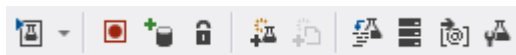
Na obrázku číslo 50 je vidieť, že boli zaznamenané 3 akcie, otvorenie lokálnej adresy, prihlásenie testovacieho užívateľa, vyplnenie formuláru s presmerovaním na stránku s výsledkom. Detaily týchto request-ov (požiadaviek) sú zobrazené v prehľadom strome. Jednotlivé vetvy je možné rozbaľovať a získať tak detailné informácie o zadaných hodnotách (Form Post Parameters).

#### 6.4.2 Web Performance Test Editor

Editor zobrazuje stromovú štruktúru všetkých request-ov (požiadaviek) a response (odpovedí). Zobrazuje všetky vlastnosti (Properties) request-ov a parametre response. Úpravy môžu byť vytvorené v koreňovom uzle – v tomto prípade sa týkajú celého web testu alebo priamo na jednotlivých vetvách – budú sa týkať len individuálnych častí testu.



Editor obsahuje zároveň aj panel nástrojov, ktorý poskytuje rôzne rýchle funkcie ako spustenie testu, pridávanie dátových zdrojov, nastavenie prihlasovacích údajov (credentials).



Obrázok 51 Panel nástrojov Web Test Editor (Vlastná tvorba)

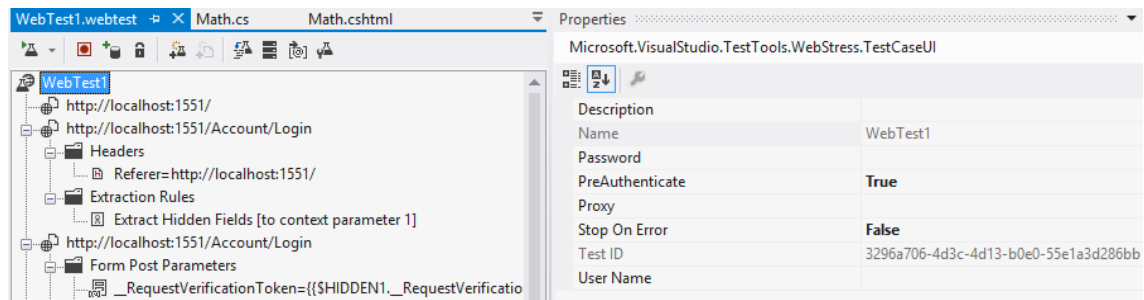
#### 6.4.2.1 Vlastnosti Webtest-u

Tabuľka číslo 3 obsahuje vlastnosti, ktoré možno nastaviť pre webové testy pomocou editora:

Tabuľka 3 Web Test Properties

Vlastnosť	Popis
<b>Description</b>	Popis daného web testu
<b>Name</b>	Pomenovanie webového testu
<b>User Name</b>	Preddefinované meno užívateľa, ak sa v teste vyžaduje prihlásenie alebo iná forma credentials
<b>Password</b>	Heslo daného užívateľa
<b>PreAuthenticate</b>	Špecifikuje či bude autentifikovaný každý request na stránke. Ak je nastavený na TRUE, autentifikačná hlavička (header) sa pošle v každej požiadavke
<b>Proxy</b>	Nastavenie Proxy servera
<b>Test ID</b>	Automaticky generované ID testu

Všetky vlastnosti sa dajú nastavovať v príslušnom Properties okne.



Obrázok 52 Okno Web Test Properties (Vlastná tvorba)

#### 6.4.2.2 Web test request Properties

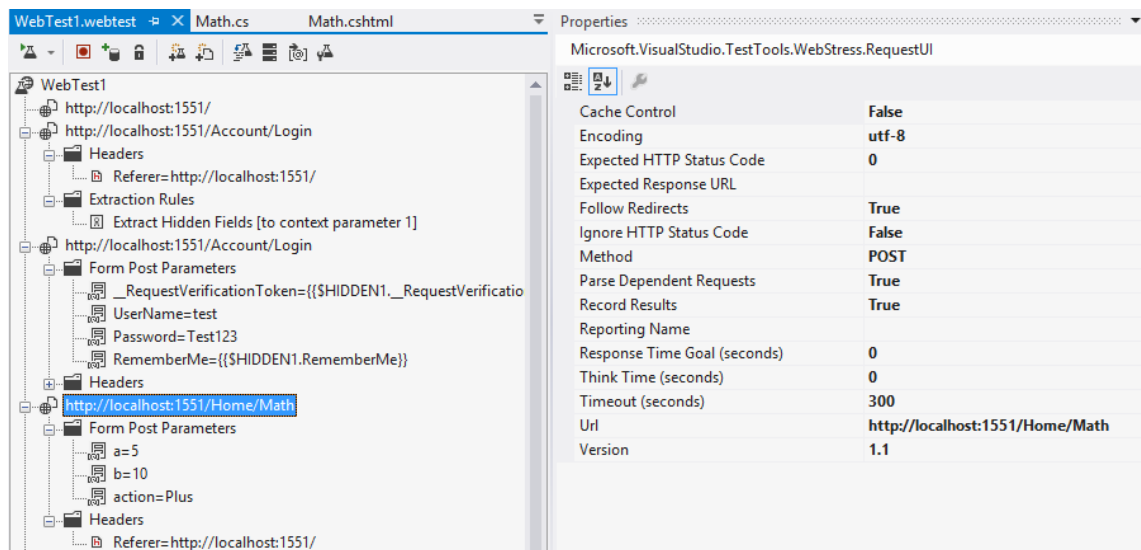
Vlastnosti request-ov je možné pridávať a upravovať na každom uzle zvlášť (na rozdiel od vlastnosti Web Testu)

Tabuľka 4 Web Request Properties

Vlastnosť	Popis
<b>Cache control</b>	Simulovanie cache (medzipamäte) webových stránok. Ak je zapnuté, zdroje webovej stránky (obrázky, CSS a JS súbory atď.) sú načítané len raz a uložia sa do cache pamäte. V prípade vypnutého cache control sa budú zdroje načítavať vždy odznovu (pri modifikácii JavaScript súborov testom).
<b>Encoding</b>	Kódovanie. Default UTF-8
<b>Expected HTTP status code</b>	Nastavuje očakávaný stavový kód pre request. Ak nie je request nájdený, tak server vráti hodnotu 404, pričom by mala byť táto hodnota nastavená v tejto Vlastnosti. Default je nastavená 0, ktorá vracia hodnotu Pass (Vyhovel), ak je návratový kód medzi 200 až 300 a hodnotu Fail, ak je kód medzi 400 až 500.

<b>Expected response URL</b>	Nastaví finálnu URL pre súčasný request. Táto vlastnosť sa používa pri Redirect (Presmerovaní stránok)
<b>Follow redirects</b>	Povoľuje presmerovanie (redirect)
<b>Method</b>	Slúži na nastavenie http metódy (get alebo post)
<b>Parse dependent requests</b>	Používa sa na zapnutie alebo vypnutie zberu závislých požiadaviek, ako je napríklad zaznamenanie stiahnutia obrázkov na požadovanej stránke. Default je táto možnosť vypnutá.
<b>Response time goal (Seconds)</b>	Nastavuje dĺžku odozvy (v sekundách) odpovede serveru. Ak sa nastaví na 0, je možné merať rýchlosť odozvy aplikácie
<b>Think time (Seconds)</b>	Nastavuje „čas na rozmýšľanie“, pretože v reálnom prostredí užívateľovi trvá určitý čas, kým vykoná akciu, resp. čas, ktorý užívateľ potrebuje na premyslenie ďalšieho kroku. Web Test Recorder zaznamenáva časy pri nahrávaní testu, je však možné ich dodatočne zmeniť.
<b>Timeout (Seconds)</b>	Maximálny čas na request, ak sa do uplynutia nevráti odpoveď, test je označený ako Neúspešný.
<b>Version</b>	Nastavenie verzie http protokolu.
<b>URL</b>	URL adresa request-u.

Všetky vlastnosti z tabuľky 4 je možné editovať v Properties okne.

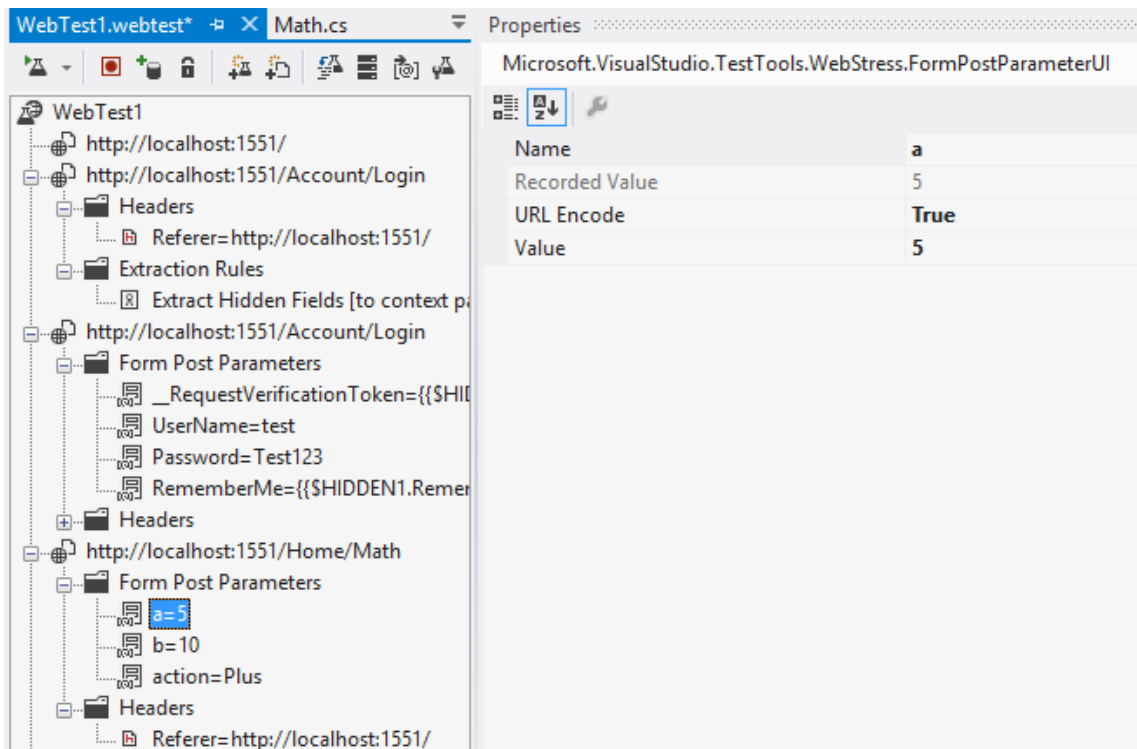


Obrázok 53 Okno Web Request Properties (Vlastná tvorba)

#### 6.4.2.3 Form Post Parametre

Tieto parametre sú zasielané v POST request-e napríklad ako hodnoty formulára odoslané na server. Hodnoty sú zaznamenané počas nahrávania testu a je možné ich editovať pravé v tejto časti web testu.

- Vlastnosť **Name** je generovaná dynamicky počas nahrávania a označuje názov komponenty použitej pre zber dát.
- **Recorded Value** je read-only vlastnosť, v ktorej je uložená pôvodná zaznamenaná hodnota.
- **URL Encode** určuje názov, hodnota parametru má byť kódovaná v URL.
- **Value** je aktuálna hodnota parametru a je nastavená na hodnotu, ktorá bola zaznamenaná počas nahrávania testu, ale môže byť dodatočne zmenená. Túto vlastnosť je možné flexibilne viazať na pole dátového zdroja, akým môže byť XML, Databáza, CSV a i. Používa sa pri tvorbe Data Driven Web Performance Testov



Obrázok 54 Nastavenie Form Post parametrov (Vlastná tvorba)

Táto sada vlastností sa líši v závislosti od ovládacieho prvku, použitého na webovej stránke. Ak by bol vo formulári použitý File Upload, ktorý vyžaduje súbor, je možné pridať túto vlastnosť s URL k danému súboru.

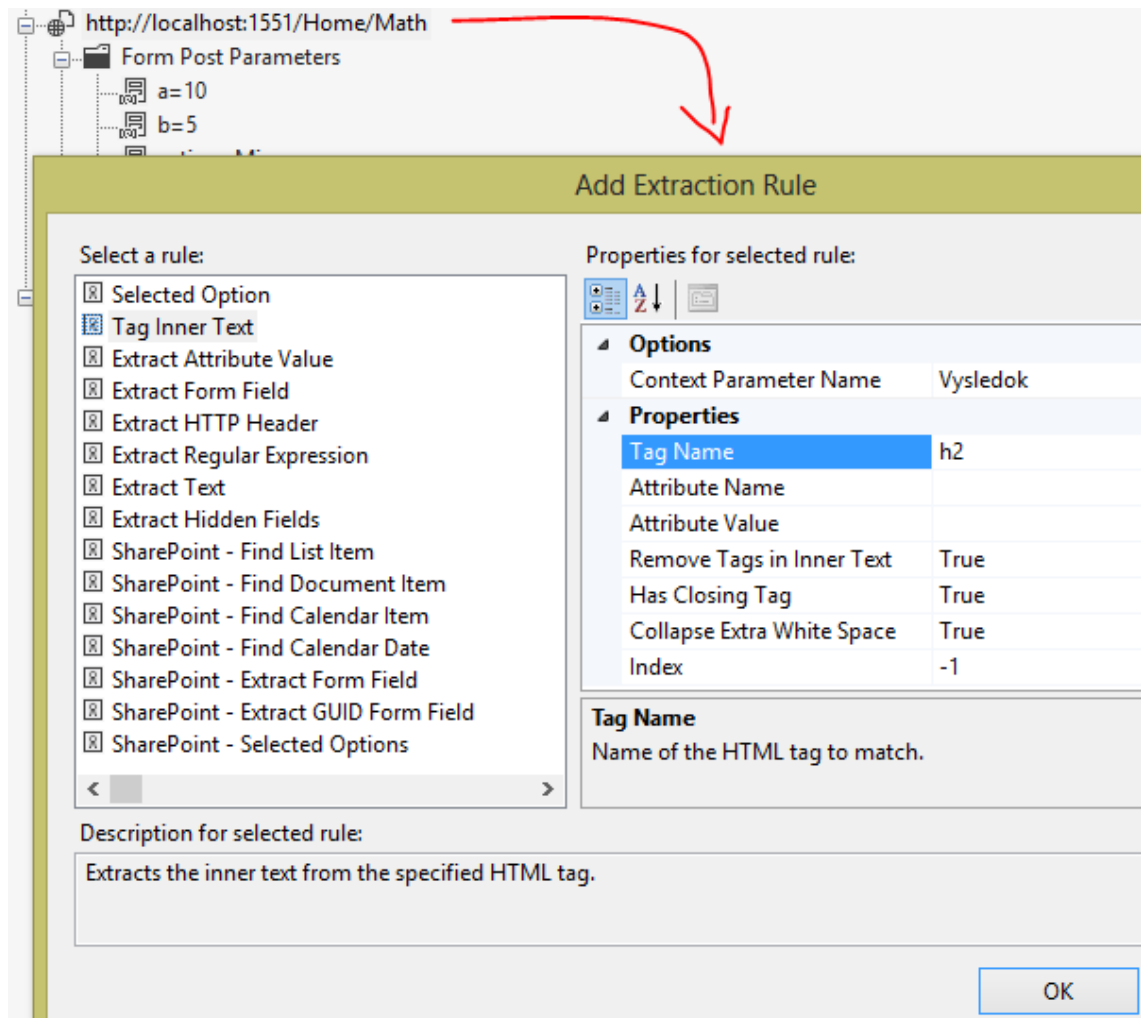
#### 6.4.2.4 Extraction Rules - Extrakčné pravidlá

Extrakčné pravidlá slúžia na výber konkrétnych dát z Odpovedí (Response), ktoré sú uložené v kontextových parametroch a sú prístupné počas celého behu testu. Tieto parametre sa môžu čítať alebo zapisovať tak, aby k nim mali prístup aj následné časti (kroky) testu. Webový test má v sebe zabudované niektoré extrakčné pravidlá, pomocou ktorých sa dá extrahovať akýkoľvek HTTP atribút alebo správa z response.

Visual Studio podporuje niekoľko preddefinovaných extrakčných pravidiel, ktoré sú založené na získavaní údajov z HTML tagov alebo na základe typu poľa webového formuláru. Je možné prirodzene vytvárať aj vlastné Extrakčné pravidlá. Pridanie Extrakčného pravidla do testu je jednoduché – kliknutím na príslušnú Požiadavku a následným vybraním Add Extraction Rule z kontextovej ponuky sa zobrazí okno, v ktorom je možné zvoliť príslušné pravidlo podľa požiadaviek.

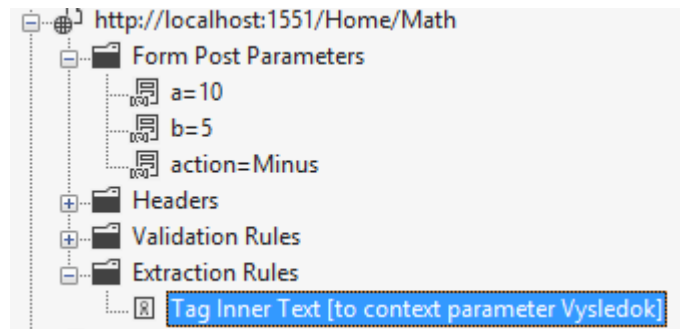
V prípade, že by aplikácia bola multi jazyková a bolo by potrebné kontrolovať hodnotu nadpisu výsledku, teda hodnotu obsiahnutú v tagu h2 (Obrázku 48), zvolilo by

sa pravidlo Tag Inner Text, teda hodnota vnútorného textu daného tagu a kontrolovala by sa hodnota podľa príslušného jazyka. Iná očakávaná hodnota by bola v českom, slovenskom alebo anglickom jazyku. Vo vlastnosti Tag Name sa definuje, ktorý tag sa má na stránke nájsť a aká hodnota má v ňom byť uložená.



Obrázok 55 Pridanie extrakčného pravidla (Vlastná tvorba)

Po potvrdení sa vo Web Test zobrazí nová záložka Extraction rules. Je možné pridať niekoľko extrakčných pravidiel pre jeden request.

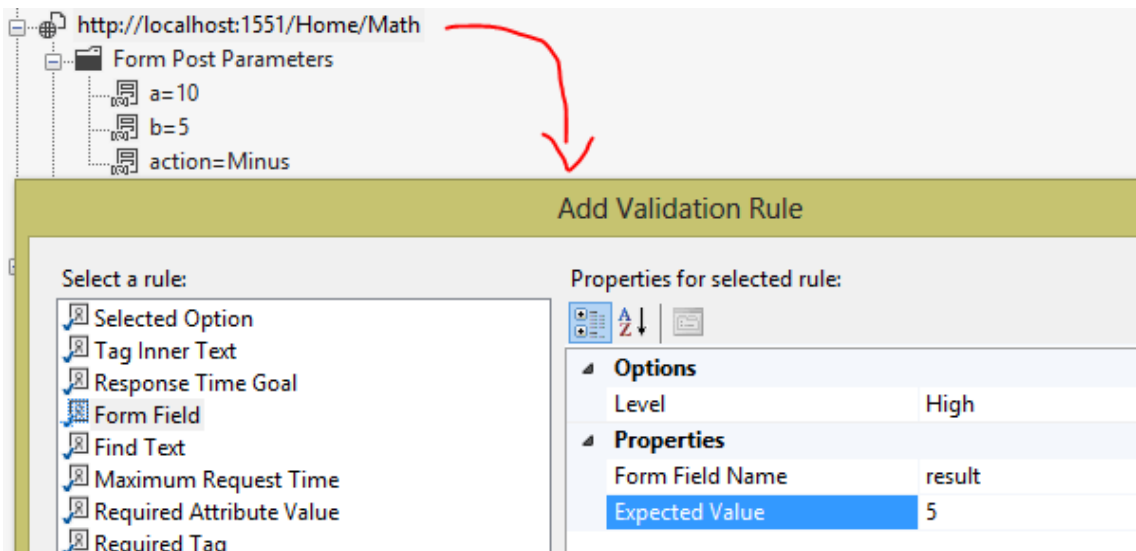


Obrázok 56 Extrakčné pravidlo v Web Test Editore (Vlastná tvorba)

#### 6.4.2.5 Validation rules - Validačné pravidlá

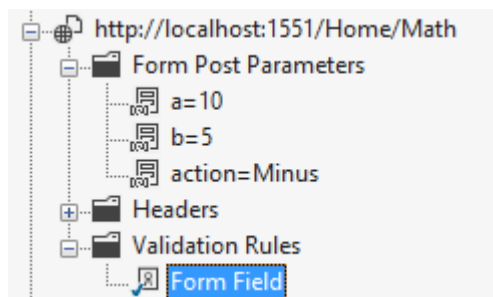
Validačné pravidlá slúžia na overenie, či je obsah alebo chovanie response (odpovedí) korektné. Na overenie správnych hodnôt slúžia práve validačné pravidlá. Validačné pravidlá predstavujú sadu pravidiel, ktoré overujú hodnoty prenášané medzi request-om a response. Všetky dáta z response sú validované voči zadaným pravidlám a test je vyhodnotený ako úspešný len vtedy, ak všetky dáta zodpovedajú vytvoreným pravidlám.

VS poskytuje niekoľko preddefinovaných validačných pravidiel, používaných na overovanie hodnôt vrátených z response. Pre pridanie overovacieho pravidla je nutné kliknúť na príslušný request web testu a v dialógovom okne vybrať Add Validation Rules. Pre overenie výsledku, ktorý je zobrazený v textovom poli s názvom (name) a identifikátorom (id) „result“, sa vyberie z preddefinovaných pravidiel možnosť Form Field. Táto možnosť slúži na overovanie hodnôt formulárových polí. Do form Filed Name sa vyplní názov poľa („result“) a do Expected value očakávaná hodnota. HTML kod je znázornený na Obrázku 48.



Obrázok 57 Pridanie Validačného pravidla (Vlastná tvorba)

Po potvrdení sa vytvorí na danom requeste nová zložka Validation rules, s pridaným pravidlom. Možné je taktiež pridávať niekoľko na sebe nezávislých pravidiel.



Obrázok 58 Validačné pravidlo v Web Test Editore (Vlastná tvorba)

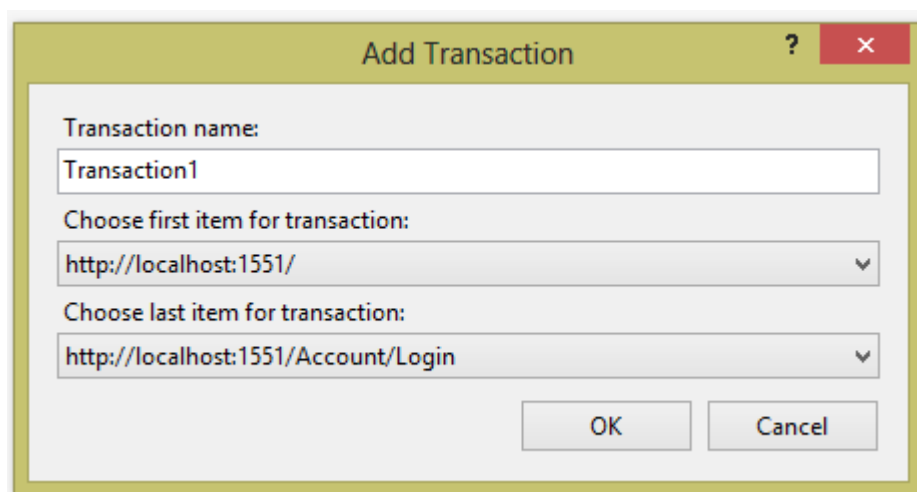
#### 6.4.2.6 Transactions - Transakcie

Transakcie sa používajú pre zoskupenie radu aktivít. Zoskupujú niekoľko request-ov do jednej skupiny, ktorá sa v teste vykonávaná ako jeden request (reálne je to niekoľko request-ov).

Transakcia sa vytvára kliknutím pravého tlačidla myši na príslušný request a výberom možnosti Add Transaction z kontextovej ponuky. Po zobrazení okna s novou transakciou je nutné zvoliť začiatkový a koncový request, všetky request-y medzi nimi budú zaradené do transakčnej skupiny.

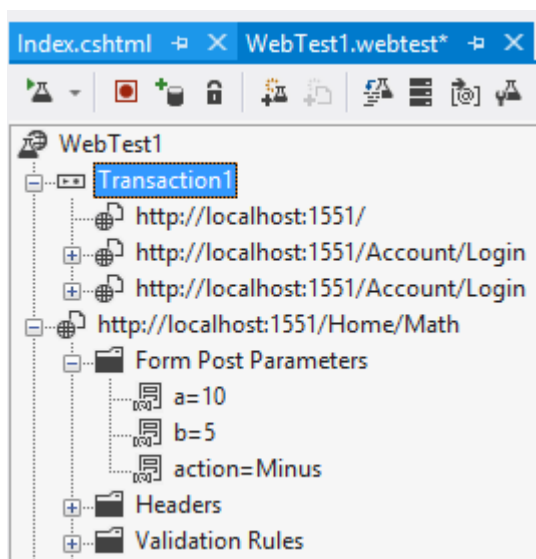


V uvedenom prípade bude obsahom transakcie príchod na úvodnú stránku, prechod na stránku, prihlásenie, vyplnenie prihlasovacích údajov a po úspešnom prihlásení presmerovanie na úvodnú stránku.



Obrázok 59 Pridanie transakcie (Vlastná tvorba)

Po vyplnení povinných detailov a následnom potvrdení sa transakcia pridá do Web Testu, v ktorom sú obsiahnuté všetky príslušné request-y.



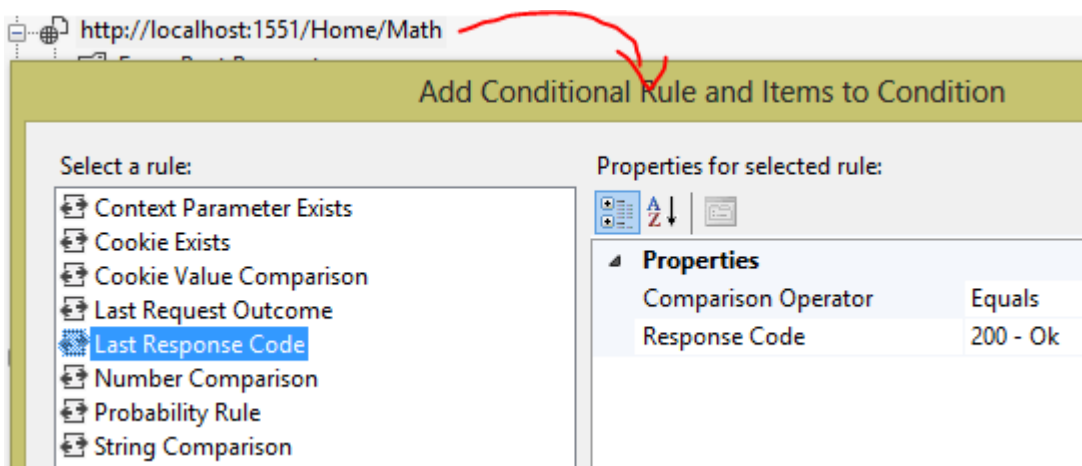
Obrázok 60 Transakcia v Web Test Editore (Vlastná tvorba)

#### 6.4.2.7 Conditional rules - Podmienené pravidlá

Podobne ako extrakčné a validačné pravidlá, môžu byť do Web testu pridané aj Podmienené pravidlá, a to za účelom prídania IF/THEN logiky pri behu testu na základe

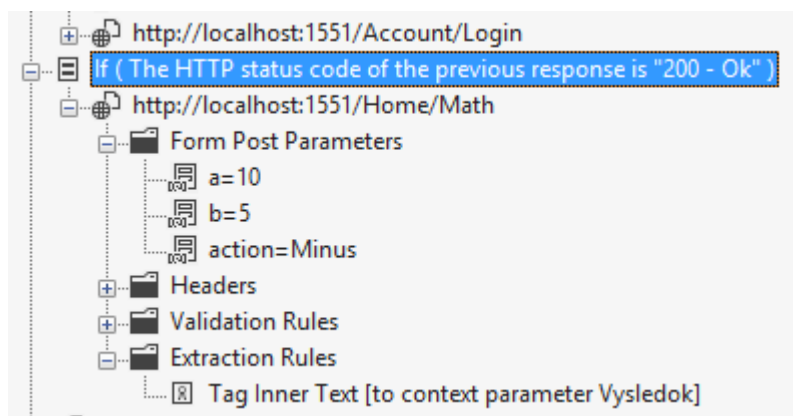
hodnôt parametrov.

Vytvorenie je podobné ako v predchádzajúcich častiach – kliknutím pravým tlačidlom myši na daný request a výberom z kontextovej ponuky Add Conditional Rule. Aj v tomto prípade existuje niekoľko možných už pred pripravených pravidiel. Je možné porovnávať existenciu daného parametru, existenciu cookie súboru alebo jeho hodnotu, porovnanie textového reťazca parametru alebo jeho číselnej hodnoty a taktiež porovnanie hodnoty response kódu predchádzajúceho request-u. Ak prihlásenie užívateľa v transakcii prebehlo v poriadku, môže sa v teste pokračovať ďalej. V opačnom prípade bude test smerovať do druhej „ELSE“ vetvy.



Obrázok 61 Pridanie Podmieneneho pravidla (Vlastná tvorba)

Podmienka overuje hodnotu response kódu, ak je 200 – OK, čo v danom príklade znamená, že transakcia prihlásenia užívateľa prebehlo v poriadku a server vrátil „OK“, čiže je možné pokračovať v teste, ak však server vráti inú hodnotu test môže pokračovať „ELSE“ vetvou alebo môže byť vyhodnotený ako Failed.



Obrázok 62 Podmienene pravidlo v Web Test Editore (Vlastná tvorba)

Request	Status
Transaction1	
http://localhost:1551/	200 OK
http://localhost:1551/Account/Login	200 OK
http://localhost:1551/Account/Login	302 Found
http://localhost:1551/	200 OK
If ( The HTTP status code of the previous response is "200 - Ok" )	Condition Met
http://localhost:1551/Home/Math	200 OK

Obrázok 63 Podmienene pravidlo v Test Result (Vlastná tvorba)

### 6.4.3 Panel nástrojov Web Testu

Web test editor obsahuje aj panel nástrojov pre komfortné a hlavne zrýchlené pridávanie konfigurácií celého testu.

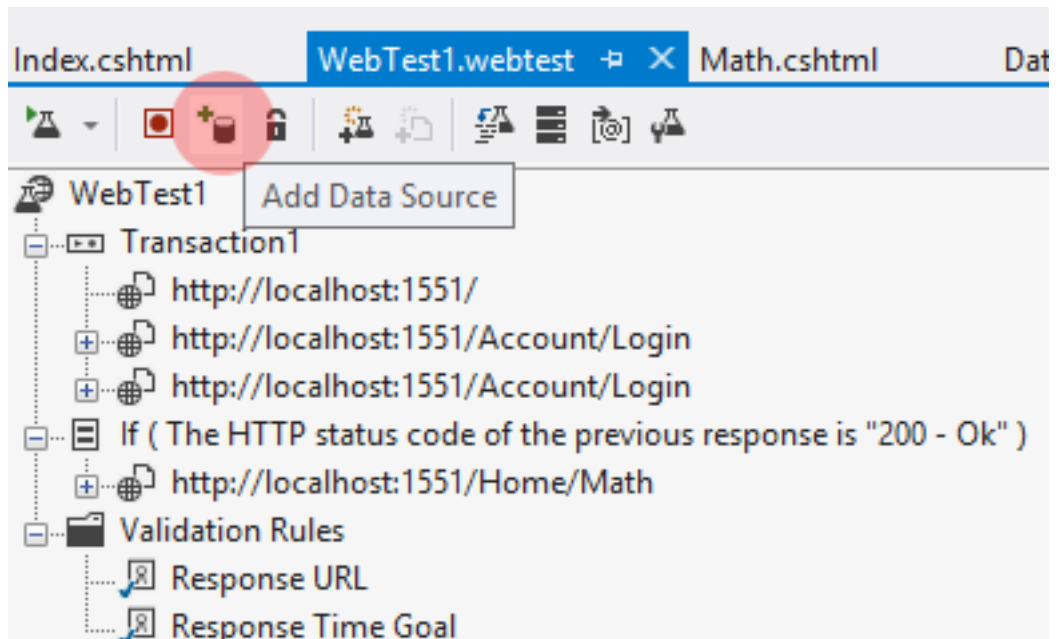


Obrázok 64 Panel nástrojov Web Test Editoru (Vlastná tvorba)

#### 6.4.3.1 Add data source – Tvorba Data driven Web testu

Podobne ako v Unit Testoch je možné aj pri Webových testoch riadiť vstupné dáta pomocou pripojenia na dátový zdroj. Ako dátový zdroj sa môže použiť databáza, CSV súbor, prípadne súbor XML. Ako zdroj dát v tomto prípade poslúži XML súbor, v ktorom budú hodnoty a správne výsledky. Vo webovom teste je dátový zdroj možné viazať na niekoľko typov hodnôt, medzi ktoré patria napríklad parametre formulárov, názvy a hodnoty parametrov URL alebo záhlavie http.

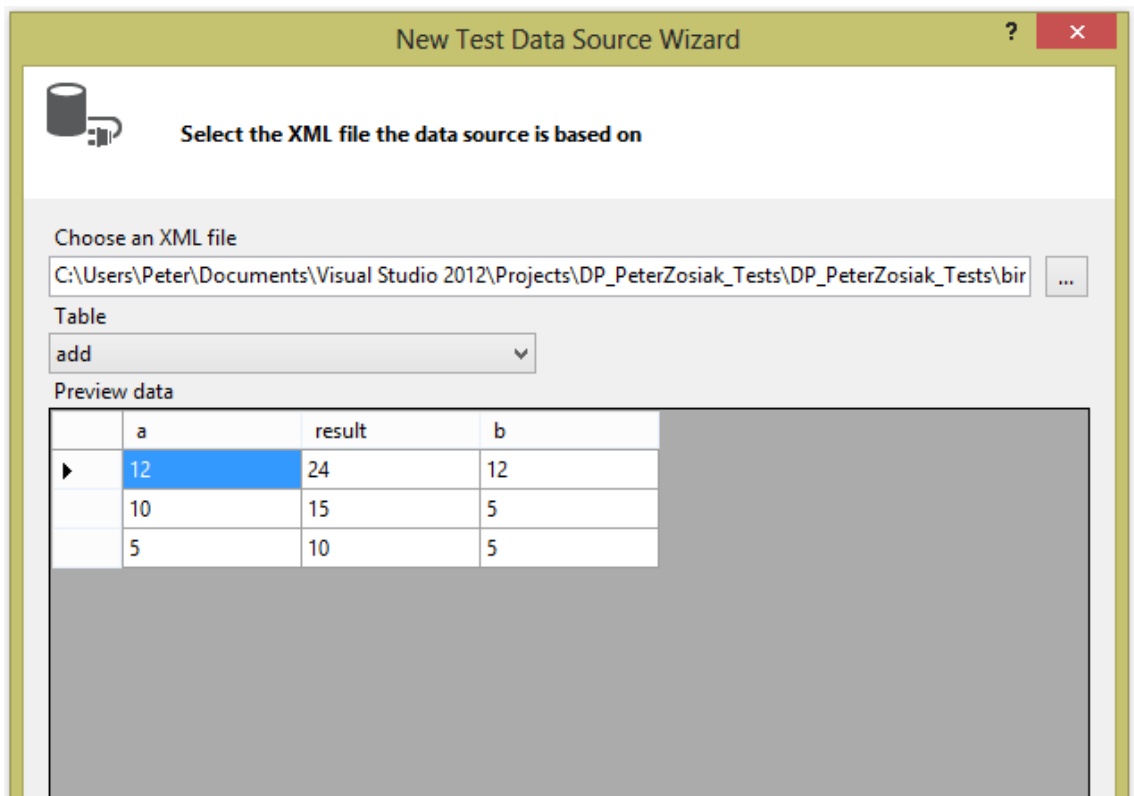
Tak ako je uvedené v kapitolách 6.1.1 a 6.3.4, hlavnou myšlienkou je bindovanie (linkovanie) dátového zdroju na daný parameter a spustenie testu toľkokrát, koľko sa nachádza v dátovom zdroji dátových riadkov. Pri web teste sa dáta budú linkovať na Form POST parametre a validovať sa bude vrátená odpoveď (response).



Obrázok 65 Pridanie Dátového zdroja (Vlastná tvorba)

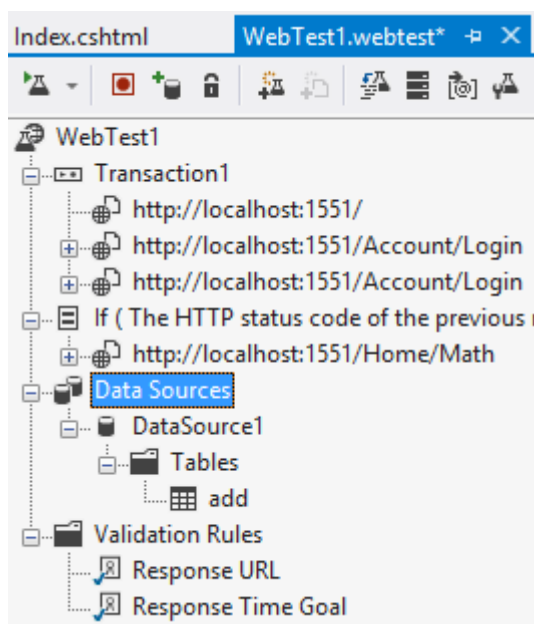
Postup pridania dátového zdroja je nasledovný:

1. Z panelu nástrojov sa vyberie možnosť Add Data Source.
2. V novo otvorenom okne je potrebné zvoliť typ dátového zdroja (Databáza, CSV, XML).



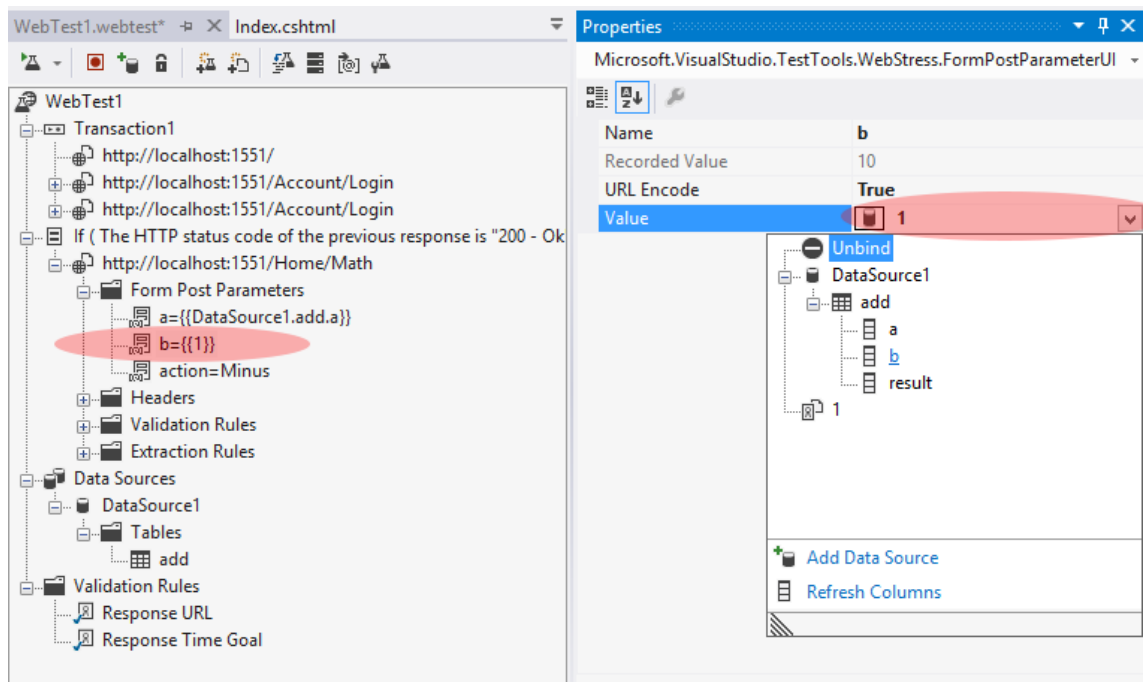
Obrázok 66 Vyber Dátového zdroja a tabuľky (Vlastná tvorba)

3. Po výbere (z už pred pripraveného XML, použitom v kapitolách 6.1.1 a 6.3.4) nasleduje výber tabuľky, v XML prípade ide o voľbu rovnakých vetiev. Keďže bol test vytvorený na funkciu Sčítania, vyberie sa ako Table hodnota add. VS následne načíta všetky dátové riadky a zobrazí ich v časti Preview data.
4. Po potvrdení tlačidlom Finish sa do testu pridá Dátový zdroj.



Obrázok 67 Dátový zdroj v Web Test Editore (Vlastná tvorba)

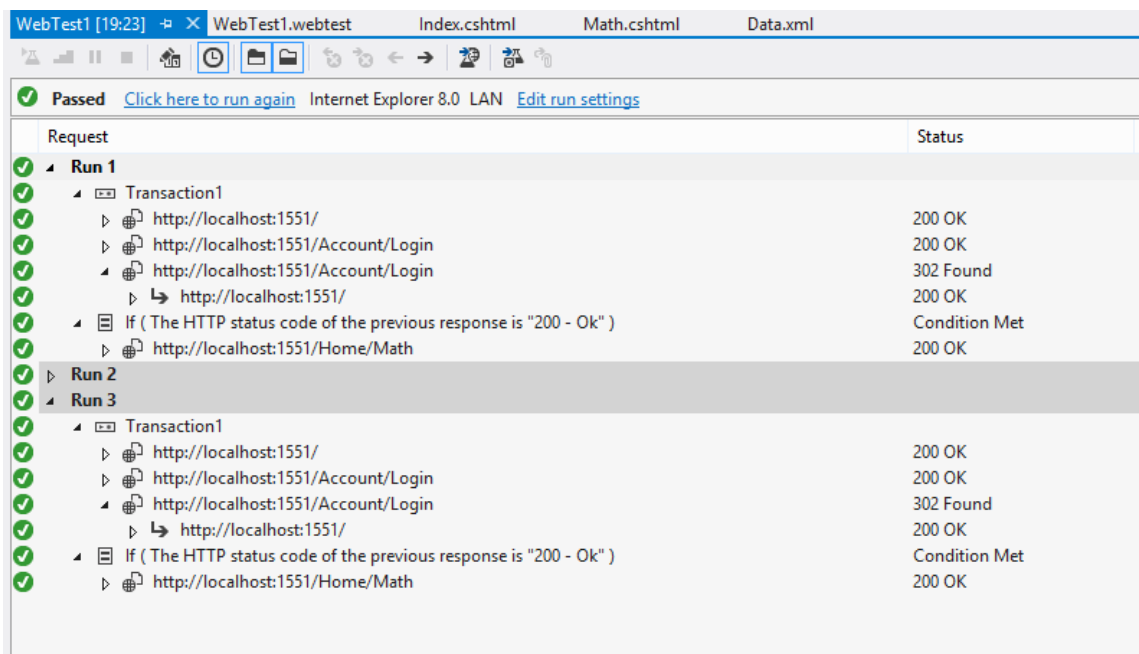
Po pridaní dátového zdroju po potrebné zmeniť zdroj hodnôt parametrov POST formuláru. Po rozkliknutí vetvy Form Post Parameters a následnom kliknutí na daný parameter sa v Properties okne (na riadku Value) zobrazí rozbaľovacia ponuka, z ktorej je nutné vybrať pridaný dátový zdroj (v tomto prípade DataSource1) a zároveň priradiť príslušné stĺpce dátového zdroja k príslušným parametrom.



Obrázok 68 Linkovanie dátového zdroju (Vlastná tvorba)

Na obrázku číslo 68 je znázornený postup aj už priradená hodnota dátového zdroja parametru „a={{DataSource1.add.a}}“.

Po nalinkovaní všetkých hodnôt a spustení testu, sa test bude opakovať presne toľko ráz, koľko existuje riadkov v dátovom zdroji.

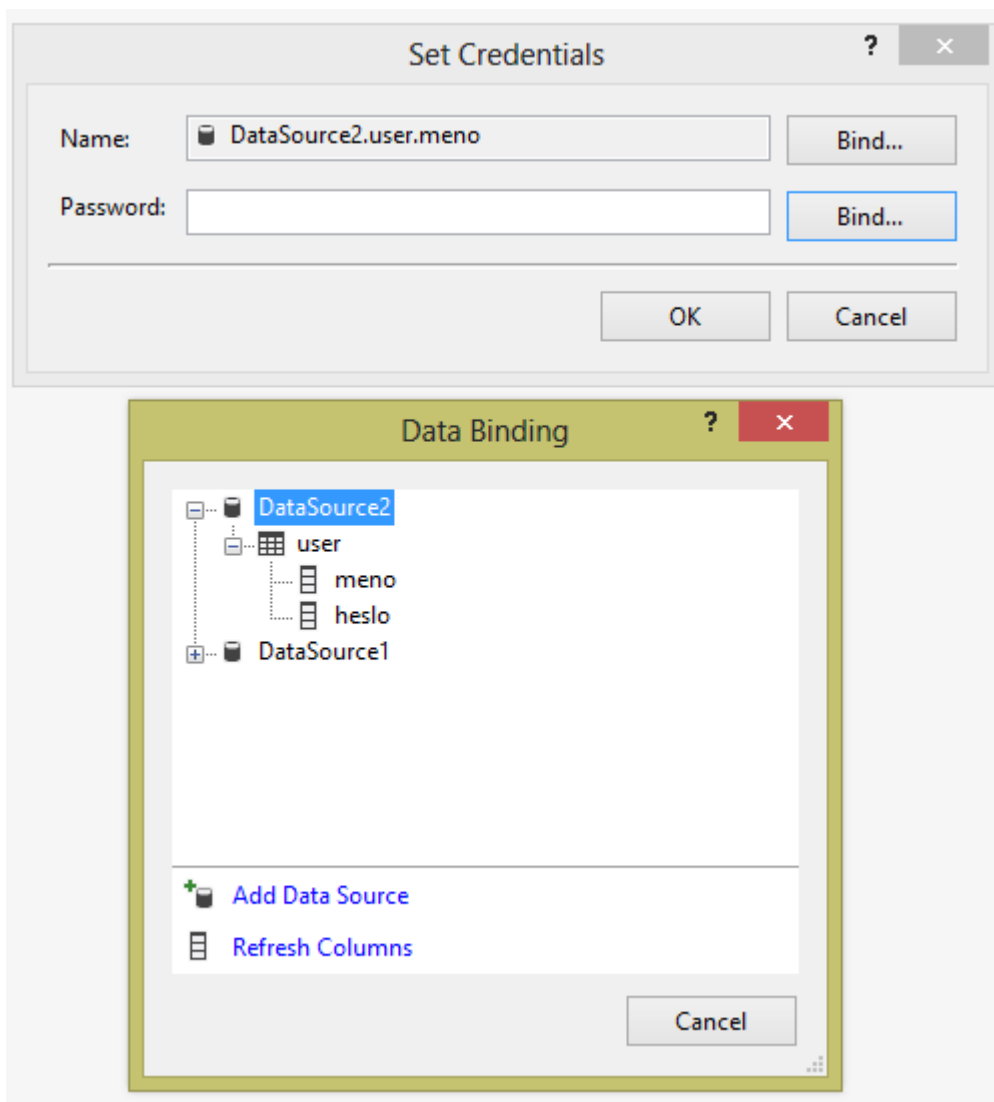


Obrázok 69 Dátový zdroj v Test Result (Vlastná tvorba)

### 6.4.3.2 Setting credentials - Nastavenie prihlasovacích údajov

Používa sa pre nastavenie špecifických užívateľských prihlasovacích údajov, ktoré majú byť použité v teste namiesto údajov súčasného užívateľa. Settings credentials nachádzajú využitie na stránkach, ktoré vyžadujú autentifikáciu. Ak je potrebné danú stránku testovať s viacerými užívateľmi s rôznymi prihlasovacími údajmi, je možné tieto údaje bind-ovať (linkovať) z dátového zdroju ako v kapitole 6.5.1.

Pre pridanie Credentials je potrebné z panelu nástrojov vybrať možnosť Add Credentials a v novootvorenom okne vyplniť prihlasovacie údaje. Ak sa v teste bude používať len jeden užívateľ, vyplní jeho aktuálne meno a heslo, v prípade viacerých užívateľov je potrebné tieto údaje previazať ako je znázornené na obrázku 71.



Obrázok 70 Pridanie Prihlasovacích údajov (Vlastná tvorba)

#### **6.4.3.3 Add recording - Pridanie nahrávania**

Pridanie nového request-u do už vytvoreného testu je vhodné najmä z dôvodu, aby nebolo nutné celý test (pri zmene stránky, request-ov a pod.) nahrávať stále odznova. Po potvrdení tejto voľby sa otvorí nové okno prehliadača s aktívnym Web Test Recorder-om. Po nahraní daného kroku sa request automaticky pridá do testu.

#### **6.4.3.4 Parameterize web server – Parametrizovaný web server**

Nahrávanie testu sa spravidla vykonáva na jednom systéme alebo serveri, vďaka čomu sú request-y zoznamované s menom a s URL adresou daného serveru a jeho portom (napr. <http://localhost:1551>). Ak by sa však zmenil port, na ktorom beží aplikácia na <http://localhost:1660>. Aby nebolo nutné nahrávať všetky testy znovu, pri zmene URL serveru alebo pri zmene portu, VS ponúka možnosť parametrizovať Web server, v ktorom je možné dynamicky meniť názov, URL a port použitého web serveru.

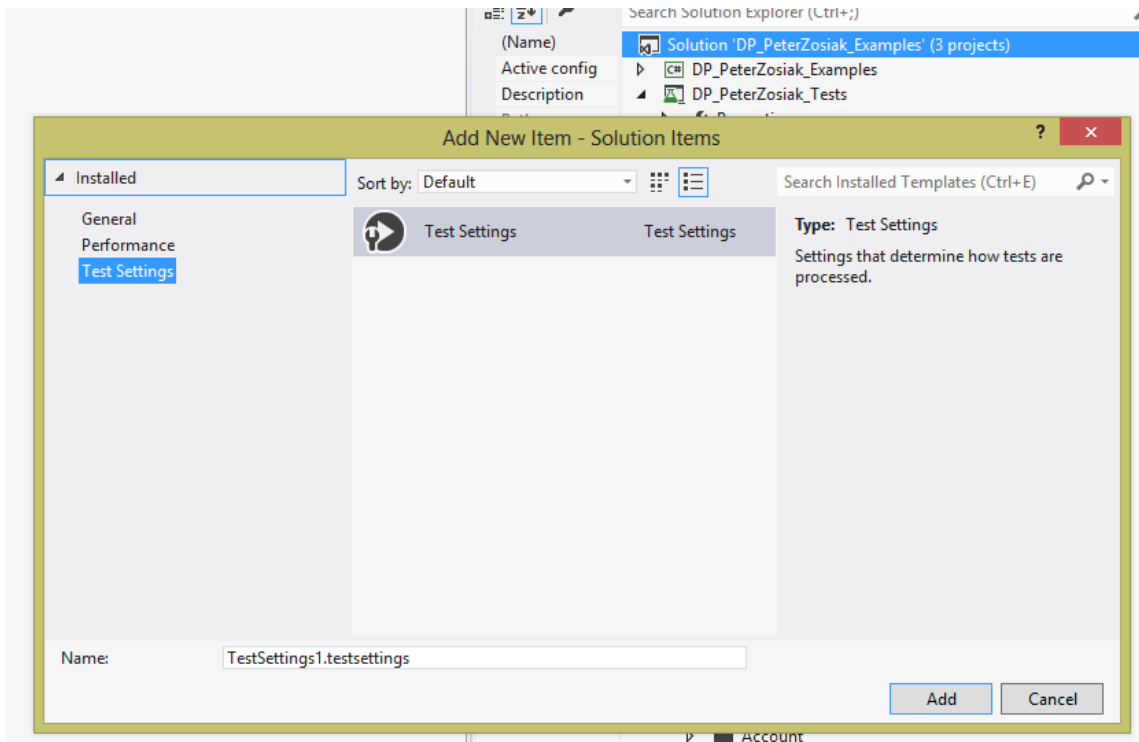
#### **6.4.3.5 Adding a web test plugin – Pridanie webového pluginu**

Plugin je externá knižnica alebo Assembly vytvorená s cieľom rozšíriť funkcionality, ktorá môže byť použitá pri behu testu.

#### **6.4.4 Ladenie a spúšťanie web testov**

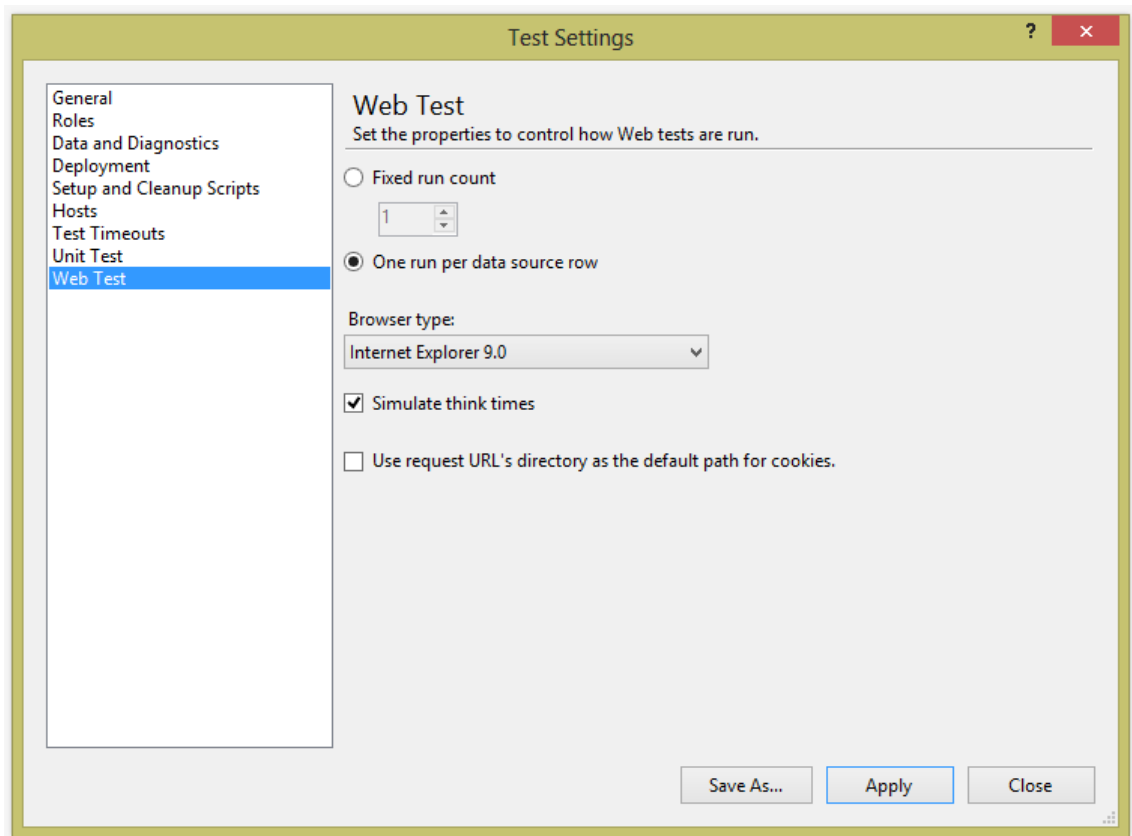
Po dokončení nahrávania testu a overení, že test funguje správne so všetkými request-ami, parametrami a pod., je potrebné zaistiť, že funguje správne aj na iných konfiguráciách (napr. na prehliadači). Pre modifikáciu default nastavení slúži .testsetting súbor. Ak v Testovacom projekte nie je vytvorený automaticky, je ho možné vytvoriť kliknutím na Solutions v Solutions Explorer okne a vyhľadať možnosť Test Settings.





Obrázok 71 Pridanie Test Settings (Vlastná tvorba)

Test Settings súborov môže byť niekoľko, v rôznych konfiguráciách a je možné ich exportovať a importovať do projektu. Preto sa odporúča vyplňať zmysluplné meno Name a Description na záložke General. Pri tvorbe a ladení Webových testov je však podstatná záložka Web test, kde sú popísané všetky nastavenia, a prepíšu default nastavenia.

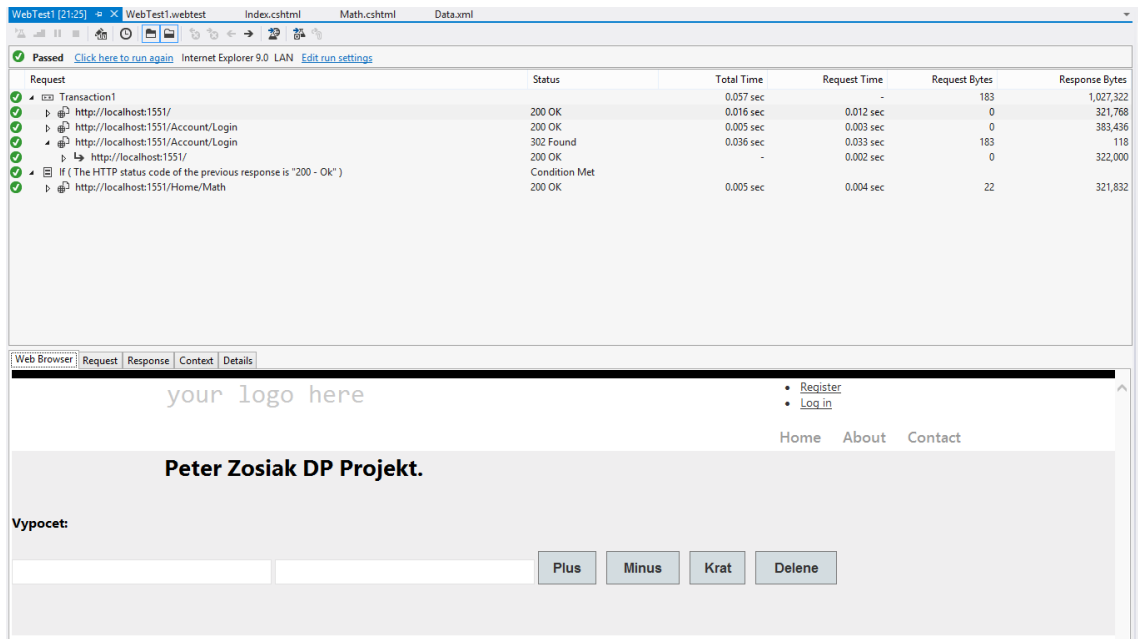


Obrázok 72 Nastavenie Web Testov (Vlastná tvorba)

- **Number of Run Iterations:** Nastavuje koľkokrát sa ma daný webový test spustiť. Je možné nastaviť fixné číslo alebo nastaviť dynamicky podľa počtu riadkov v dátovom zdroji.
- **Browser Type:** Nastavuje typ a verziu prehliadača. Rozbaľovacia ponuka s možnosťou nastavenia najpoužívanejších verzií prehliadačov – desktopových alebo mobilných klientov.
- **Simulate think times:** Simulácia času na rozmyslenie.

#### 6.4.4.1 Spúšťanie testu

Webový test sa spúšťa z panelu nástrojov tlačidlom Run Test. Po pustení je možné sledovať priebeh každého request-u v okne webového prehliadača. Keď sa test dokončí, zobrazí sa okno s výsledkami testu a s detailnými informáciami každého request-u. Ak zlyhá aspoň jeden request alebo jedna validácia, celý test je označený ako Neúspešný. Detaily testu sú rozdelené do záložiek – každá zobrazuje iný typ zozbieraných údajov.



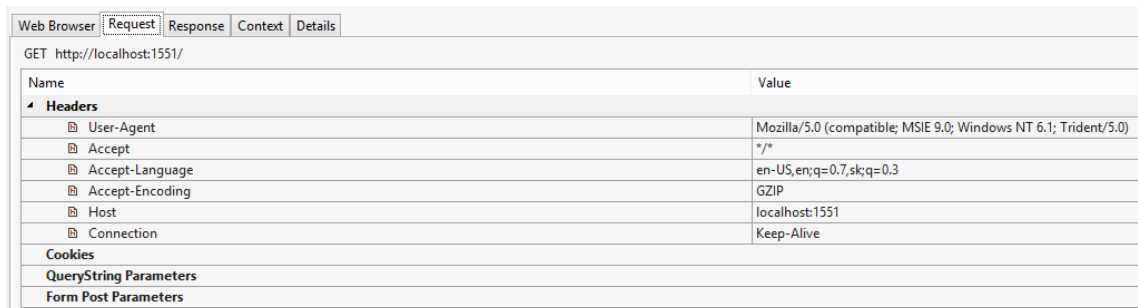
Obrázok 73 Test Result Okno (Vlastná tvorba)

#### 6.4.4.2 Web Browser

Web Browser zobrazuje rovnakú webovú stránku, aká je použitá v request-e. Zobrazuje ju kompletnú v podobe, akoby ju videl užívateľ.

#### 6.4.4.3 Request

Karta Request zobrazuje všetky informácie o request-e ako napr. Headers, Cookies, QueryString parametre a Form Post parametre.



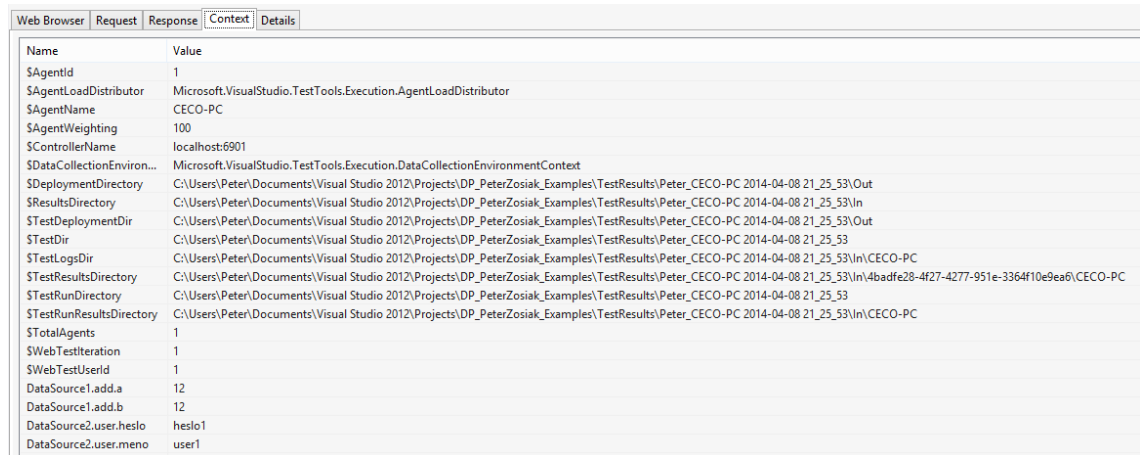
Obrázok 74 Detail záložky Request (Vlastná tvorba)

#### 6.4.4.4 Response

V tejto karte je znázornená odpoveď zo serveru (response) a jej výsledok je zobrazený ako Html kód, ktorý je možné zobrazit' v Html editore.

### 6.4.4.5 Context

Táto časť zobrazuje všetky runtime details, priradené k testu, ako sú napríklad hodnoty z dátového zdroju priradené k parametrom, testovacia zložka, meno Test Agentu a podobne.

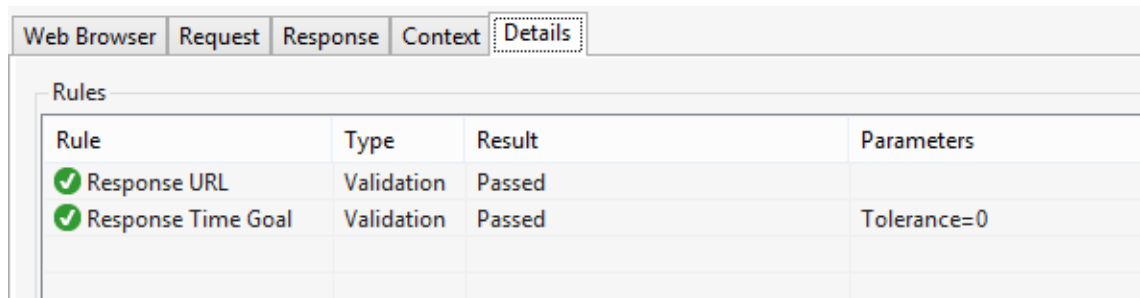


Name	Value
\$AgentId	1
\$AgentLoadDistributor	Microsoft.VisualStudio.TestTools.Execution.AgentLoadDistributor
\$AgentName	CECO-PC
\$AgentWeighting	100
\$ControllerName	localhost:6901
\$DataCollectionEnviron...	Microsoft.VisualStudio.TestTools.Execution.DataCollectionEnvironmentContext
\$DeploymentDirectory	C:\Users\Peter\Documents\Visual Studio 2012\Projects\DP_PeterZosiak_Examples\TestResults\Peter_CECO-PC 2014-04-08 21_25_53\Out
\$ResultsDirectory	C:\Users\Peter\Documents\Visual Studio 2012\Projects\DP_PeterZosiak_Examples\TestResults\Peter_CECO-PC 2014-04-08 21_25_53\In
\$TestDeploymentDir	C:\Users\Peter\Documents\Visual Studio 2012\Projects\DP_PeterZosiak_Examples\TestResults\Peter_CECO-PC 2014-04-08 21_25_53\Out
\$TestDir	C:\Users\Peter\Documents\Visual Studio 2012\Projects\DP_PeterZosiak_Examples\TestResults\Peter_CECO-PC 2014-04-08 21_25_53
\$TestLogsDir	C:\Users\Peter\Documents\Visual Studio 2012\Projects\DP_PeterZosiak_Examples\TestResults\Peter_CECO-PC 2014-04-08 21_25_53\In\CECO-PC
\$TestResultsDirectory	C:\Users\Peter\Documents\Visual Studio 2012\Projects\DP_PeterZosiak_Examples\TestResults\Peter_CECO-PC 2014-04-08 21_25_53\In\4badfe28-4f27-4277-951e-3364f10e9ea6\CECO-PC
\$TestRunDirectory	C:\Users\Peter\Documents\Visual Studio 2012\Projects\DP_PeterZosiak_Examples\TestResults\Peter_CECO-PC 2014-04-08 21_25_53
\$TestRunResultsDirectory	C:\Users\Peter\Documents\Visual Studio 2012\Projects\DP_PeterZosiak_Examples\TestResults\Peter_CECO-PC 2014-04-08 21_25_53\In\CECO-PC
\$TotalAgents	1
\$WebTestIteration	1
\$WebTestUserid	1
DataSource1.add.a	12
DataSource1.add.b	12
DataSource2.user.heslo	heslo1
DataSource2.user.meno	user1

Obrázok 75 Detail záložky Context (Vlastná tvorba)

### 6.4.4.6 Details

Záložka Details predstavuje status pravidiel, ktoré sa vykonávajú v priebehu testov. Jedným z takých pravidiel môže byť Response Time Goal, a teda maximálny čas, za ktorý musí prísť response.



Rule	Type	Result	Parameters
✓ Response URL	Validation	Passed	
✓ Response Time Goal	Validation	Passed	Tolerance=0

Obrázok 76 Detail záložky Details (Vlastná tvorba)

### 6.4.4.7 Generate Code – Tvorba Advanced Web Performance Testov

Visual Studio poskytuje možnosť generovať kód z už vytvoreného Web Testu, a to za účelom rozšírenej modifikácie, kedy už základné úpravy nestačia a je potrebné vytvárať ich programovo. Táto možnosť je natoľko rozsiahla, že ju rozsah práce nedovoľuje spracovať podrobnejšie.

## 6.5 Tvorba Load Testov

Úlohou záťažových testov je overiť, či aplikácia dokáže pracovať podľa očakávaní aj v prípade vyššieho zaťaženia zo strany počtu užívateľov. Pri týchto testoch ide hlavne o monitorovanie testovaného systému (zväčša serveru) s prioritou na jeho výkonnostné parametre.

Záťažové testy môžu byť:

1. Smoke Testy - overujú základnú funkčnosť počas zvýšenej záťaže.
2. Klasické záťažové testy - overujú plnú funkčnosť pri zvýšenej záťaži.
3. Stres testy – overujú robustnosť aplikácie pod veľkou krátkodobou záťažou ( počas špičiek)

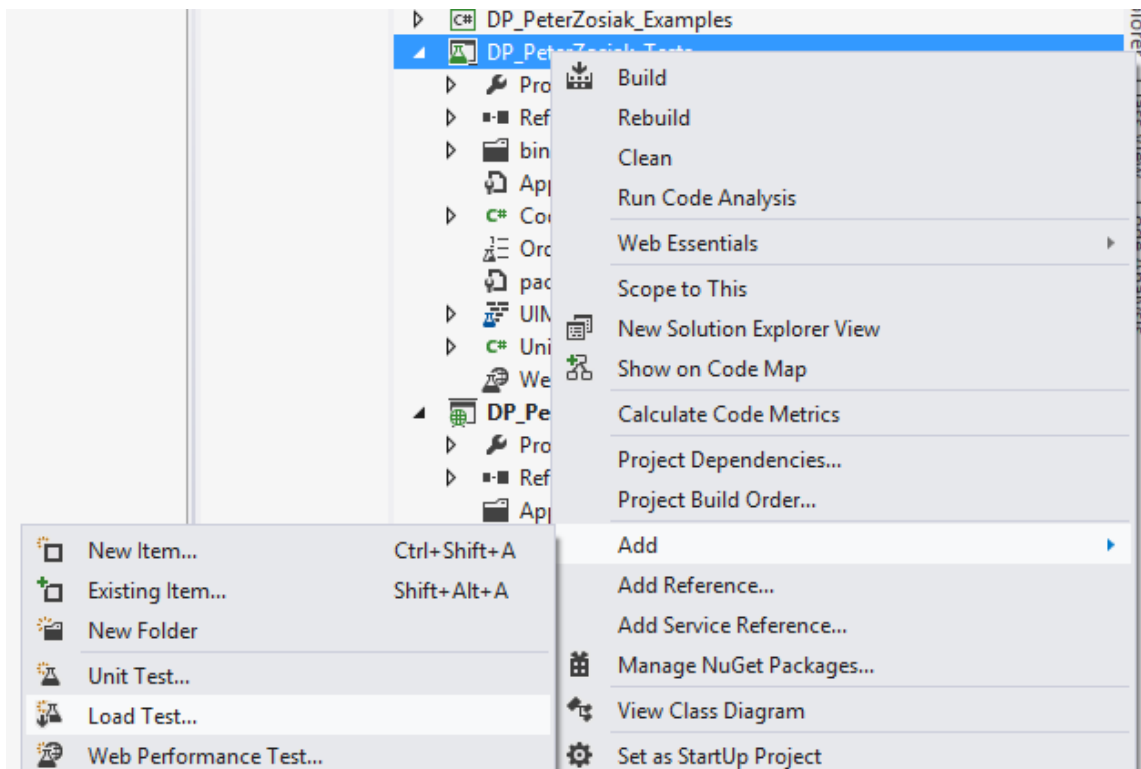
Load (Záťažové) testy v prostredí Visual Studia je možné zostavovať zo všetkých ostatných druhov testov. Je možné ich vytvoriť zo samostatných unit testov, prípadne webových testov, ale aj z viacerých Ordered testov.

Záťažový test simuluje ľubovoľnú kombináciu počtu užívateľov, rýchlosť siete, rôzne verzie webových prehliadačov a ich konfigurácií. V prípade testovania webových aplikácií je nutné testovať s rôznym počtom užívateľov, ktorý generujú veľké množstvo rozličných požiadaviek a sledovať reakciu serveru na zvýšenú záťaž. Je možné ich využiť aj pri testovaní desktopovej aplikácie na rôznych konfiguráciách systému k identifikovaniu kapacít pri ľahkom zaťažení počas krátkej doby až po veľkú záťaž v rôznych časových úsekoch.

Záťažové testy v VS pracujú na princípe Kontroléru (Controller) a User (užívateľských) agentov. Agenti reprezentujú počítače v rozdielnych lokalitách a používajú sa na simulovanie rôznych užívateľských požiadaviek (request-ov). Agenti taktiež nahrávajú a zbierajú všetky testovacie dáta. Kontrolér je centrálny počítač, ktorý riadi agentov.

### 6.5.1 Vytvárania Záťažového testu

Pridanie Load Test do projektu je podobné ako pridávanie iných typov testov, a to pravým tlačidlom myši kliknúť na Testovací projekt v Solutions okne, z kontextovej ponuky vybrať možnosť Add a zvoliť Load Test.

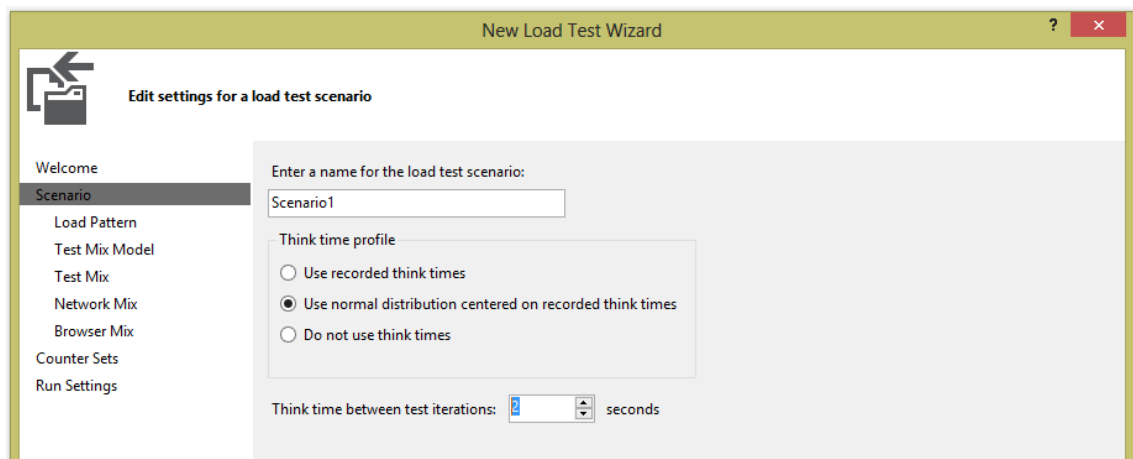


Obrázok 77 Pridanie Zát'azového testu (Vlastná tvorba)

Po potvrdení sa zobrazí nové okno s názvom New Load Test Wizard. Wizard (Sprievodca) obsahuje niekoľko častí pre definovanie potrebných konfigurácii Load testu.

### 6.5.1.1 Definovanie Think Time

Think time je čas, ktorý užívateľ potrebuje k navigácii medzi stránkami alebo k vykonaniu určitej akcie a slúži na modelovanie reálnejšieho prostredia. Používa sa aj v prípadoch, kedy je test nahraný na výkonnejšom PC a spúšťa sa na pomalšom alebo ako možnosť poskytnutia dodatočného času pre spracovanie úloh spustených na pozadí.



Obrázok 78 Definovanie Think Time (Vlastná tvorba)

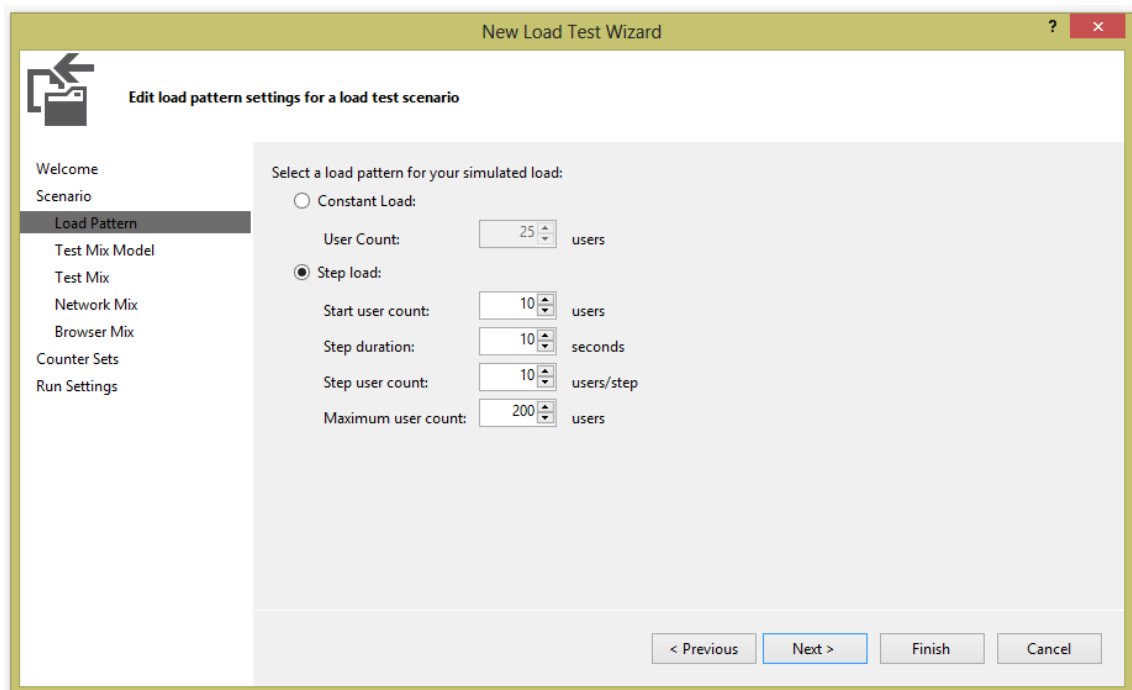
Existujú 3 základné možnosti, ako k tomuto parametru pristupovať:

1. Použiť časové intervaly – rovnaké aké boli zaznamenané pri tvorbe testu.
2. Nastaviť normálovú časovú distribúciu – strednú hodnotu s tým, že časové intervaly budú pre každý request rozdielne.
3. Nepoužívať časy vôbec.

Každý záťažový test pracuje na princípe, že spúšťa určité testy stále dokola. VS ponúka možnosť nastavenia času medzi koncom testu a jeho novým začiatkom, čiže nastavenie času medzi jednotlivými iteráciami testu.

### 6.5.1.2 Load Pattern - Testovacia vzorka

Load pattern sa používa pre nastavenie princípu zvyšovania zaťaženia užívateľov v teste. Pattern závisí na type testu. Pri Intranetových aplikáciách alebo v prípade, že záťažové testy sú zložené len s unit testov, stačí definovať menší počet užívateľov po kratšiu dobu. Pri verejných stránkach sa počet užívateľov v čase mení, preto je vhodné pri tvorbe testu začať s nižším počtom užívateľov a postupne zvyšovať na maximálnu hranicu. Napríklad zvyšovať počet užívateľov po 10 každých 10 sekúnd, pokiaľ sa nedosiahne počet 200.



Obrázok 79 Testovacia vzorka (Vlastná tvorba)

### ➤ Constant Load - Konštantná zát'az

Ak je zvolená táto možnosť, tak je za'azenie počas celého testu konštantné, nemení sa počet užívateľov po celú dobu trvania testu.

**User Count:** Určuje presný počet simulovaných užívateľov.

### ➤ Step load – Postupná zát'az

Test začína so špecifikovaním minimálneho počtu užívateľov a ich počet sa trvale zvyšuje, až kým nedosiahne maximálnej hodnoty uvedenej v teste.

**Start user count:** Počet užívateľov pri štarte testu.

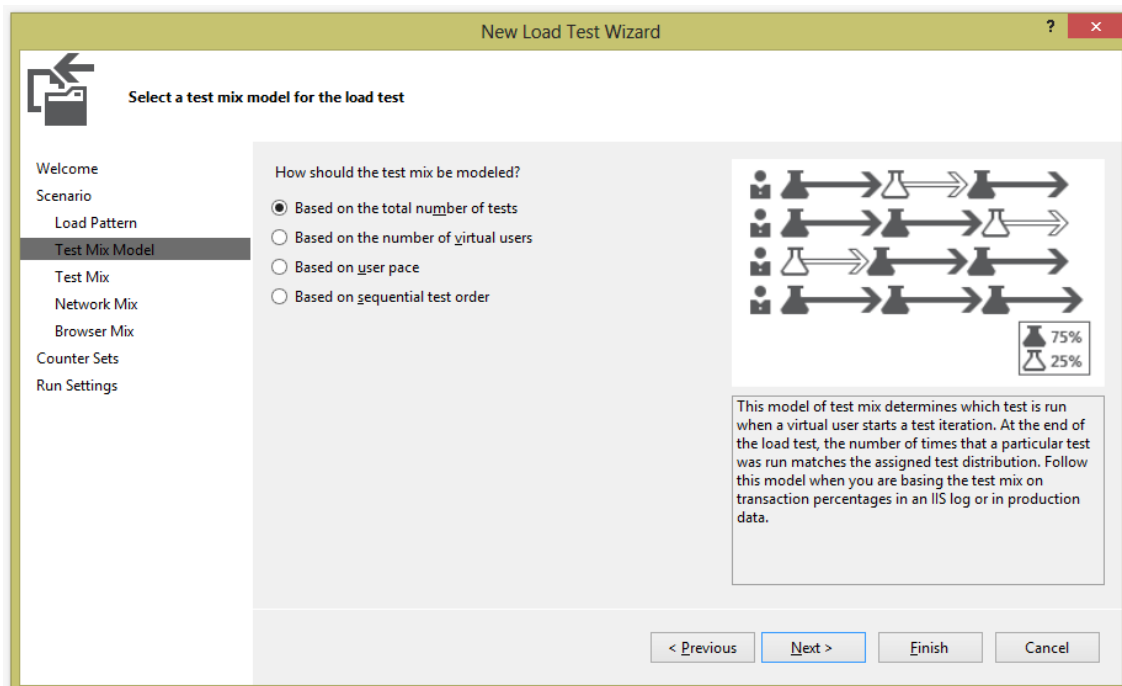
**Step duration:** Doba trvania medzi nárastom užívateľov medzi krokmi.

**Step user count:** Počet užívateľov, ktorý majú byť pridaní medzi krokmi.

**Maximum user count:** Maximálny počet užívateľov.

## 6.5.1.3 Test Mix Model

Test Mix Model predstavuje simuláciu distribúcie koncových užívateľov pre jednotlivé testy vzhľadom na počet testov, počet virtuálnych užívateľov, tempa ich pridávania a poradí testov. Test mix obsahuje rozdielny počet testov, kde má každý z nich rozdielny počet užívateľov na daný test.



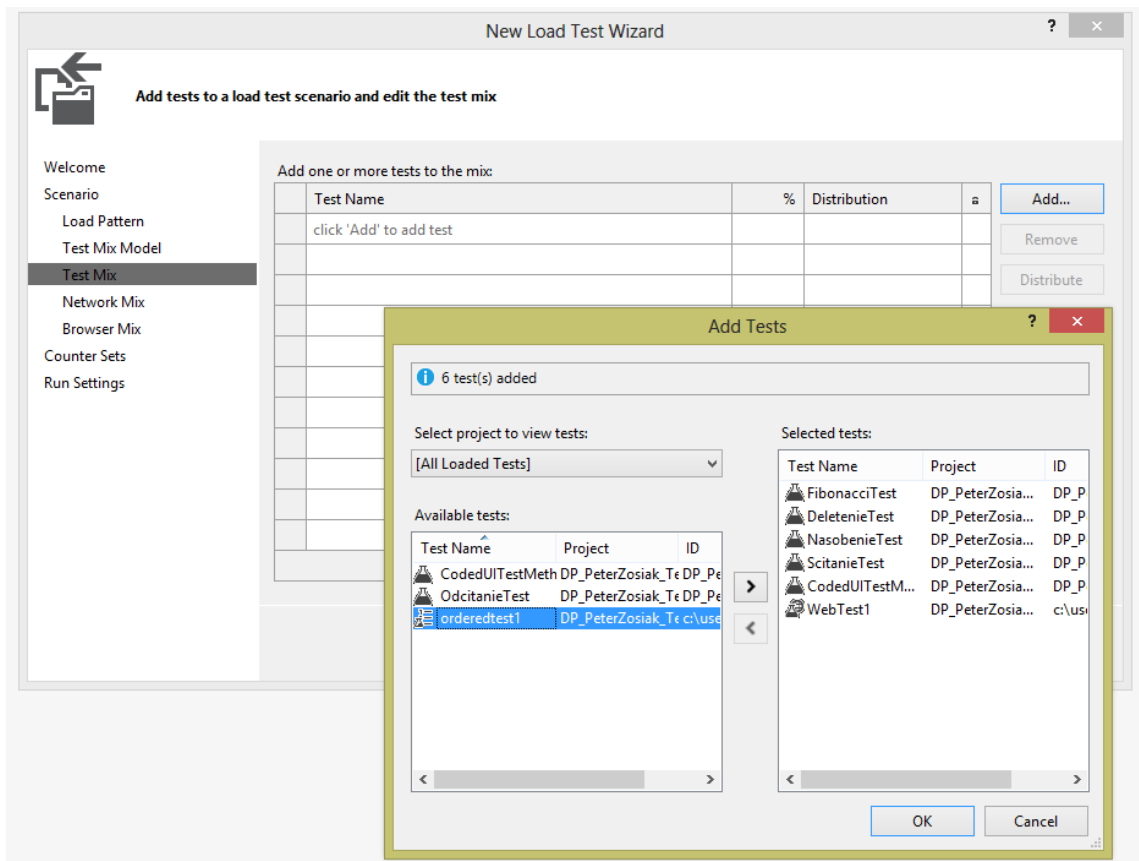
Obrázok 80 Test Mix Model (Vlastná tvorba)



- **Test Mix Model based on number of virtual users**  
Tento model je založený na percente virtuálnych užívateľov. Voľba ďalšieho testu závisí percentuálnom zastúpení virtuálnych užívateľov a taktiež na percentuálnej redistribúcii priority testov.
- **Test Mix Model based on user pace**  
Tento model spustí každý test presne špecifikovaný počet krát za hodinu.
- **Test Mix Model based on sequential test order**  
Model spúšťa testy v poradí, v akom boli nadefinované. Každý virtuálny užívateľ vykonáva všetky testy v cykle a v rovnakom poradí, ako sú nadefinované, až pokiaľ Zát'azový test neskončí.
- **Test Mix Model based on total number of tests**  
V tomto modeli závisí spustenie ďalšieho testu na základe počtu opakovaní daného testu. Počet spustení testu zodpovedá percentuálnej redistribúcii priority testov.

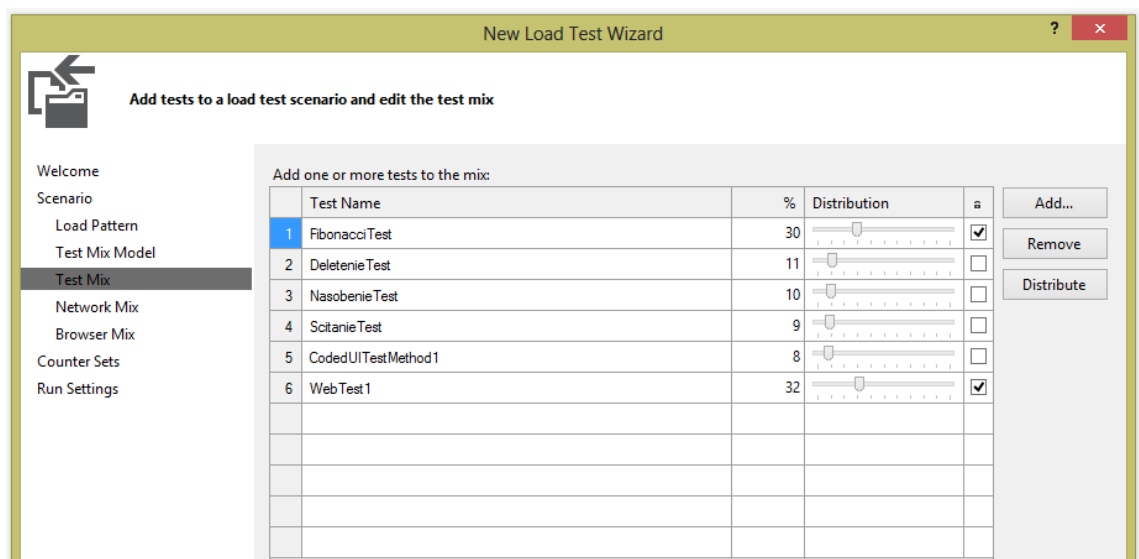
#### **6.5.1.4 Test mix**

V tejto časti sa definujú testy, ktoré majú byť zahrnuté v rámci zát'azového testu. Po kliknutí na tlačidlo Add sa zobrazí nové okno, v ktorom sú zobrazené všetky testy z Testovacieho projektu. Unit testy, Ordered, Coded UI Testy aj Web Performance testy môžu byť súčasťou Load (Zát'azového testu).



Obrázok 81 Test mix – prídanie testov (Vlastná tvorba)

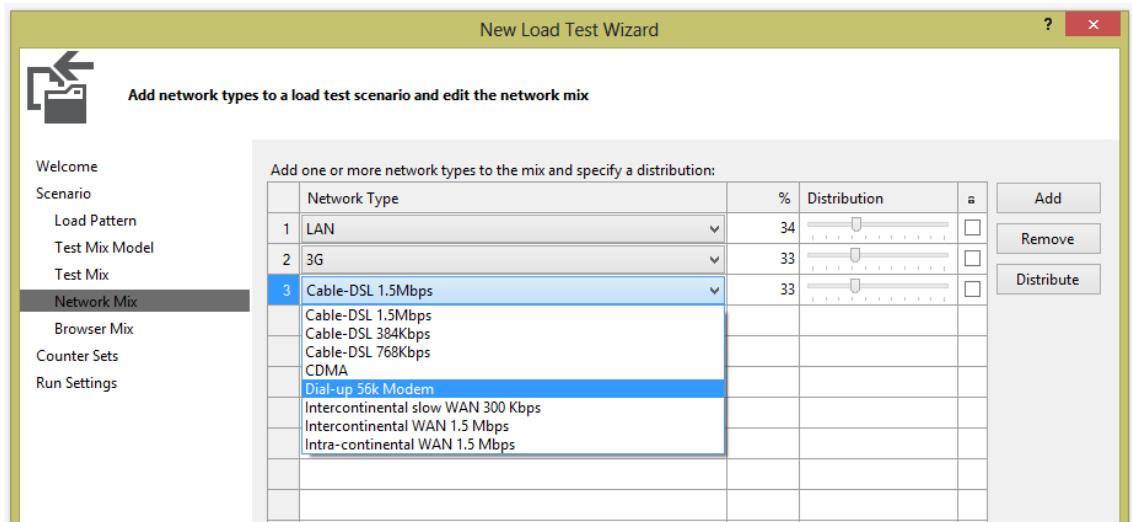
Po výbere testov a potvrdení sú testy pridané do záťažového testu a je im nastavená rovnaká priorita v rámci záťažového testu. Prioritu je možné redistribuovať, a teda zvyšovať počet opakovaní určitého testu na úkor ostatných.



Obrázok 82 Test mix – priorita (Vlastná tvorba)

### 6.5.1.5 Network Mix – Siet'ový mix

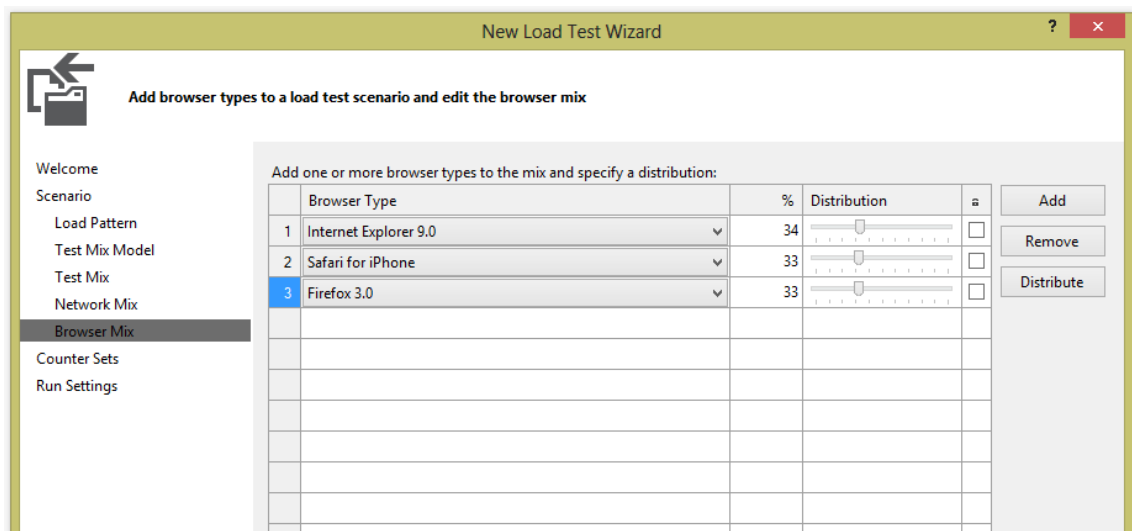
Hodnoty v Network mix slúžia na simuláciu rôznych rýchlostí sietí virtuálnych užívateľov, pretože v reálnom prostredí sa rýchlosť líši v závislosti na polohe a type použitej siete. Na výber je z niekoľkých najpožovanejších typov od LAN až po Dial-Up spojenie. Po pridaní siet'ového mixu je potrebné nastaviť distribúciu priority danej siete.



Obrázok 83 Network Mix (Vlastná tvorba)

### 6.5.1.6 Browser mix – Mix prehliadačov

Nastavenie mixu prehliadačov v rôznych verziách a konfiguráciách. Obdobne ako nastavenie mixu sietí. Vyberá sa typ a verzia prehliadača a nastaví sa jeho priorita.



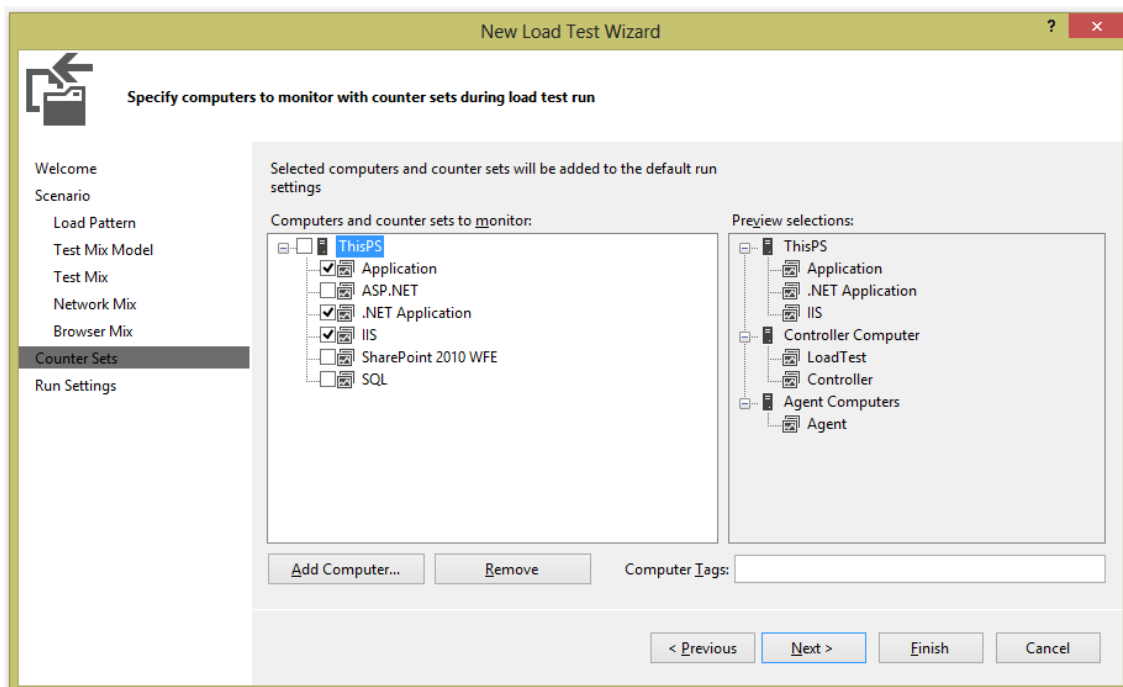
Obrázok 84 Browser mix (Vlastná tvorba)

### 6.5.1.7 Counter sets – Sada meračov

Zaťažové testy v sebe zahŕňajú špecifické merače výkonu služieb, a teda nielen aplikácie, ale aj enviromentu (systém, na ktorom test beží.). Test zhromažďuje všetky

dáta z Kontroléru a agentov a uchováva ich pre neskoršiu analýzu. Zachytené môžu byť aj ostatné systémy, ktoré nemusia byť súčasťou testu, avšak svojim behom (počas testovania) ovplyvňujú výkon. Právé dáta zo sady meračov sú kľúčové pre interpretáciu výkonu systému a aplikácie.

Wizard poskytuje možnosť pridať sady Counter sets (Sady meračov). Default je do testu pridaná sada na meranie aktuálneho systému a spoločná sada meraní pre Controller (Controller Computer) a agentov (Agent Computer).



Obrázok 85 Sada meračov (Vlastná tvorba)

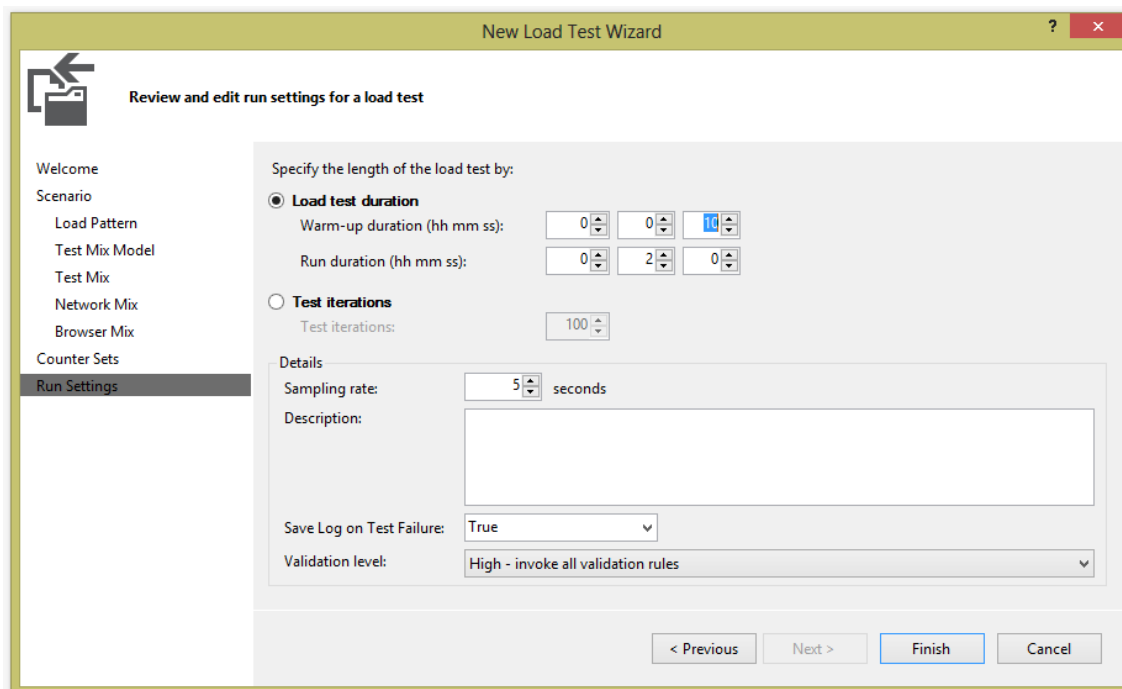
Pridanie vlastnej sady meračov je možné cez tlačidlo Add Computer, kde sa po vyplnení názvu zobrazia oblasti, z ktorých je potrebné zaznamenávať dáta znázornené na obrázku 84.

#### 6.5.1.8 Run Settings – Nastavenia behu testu

Nastavenia behu testu slúžia na nastavenie trvania záťažového testu. Je možné vybrať s dvoch možností:

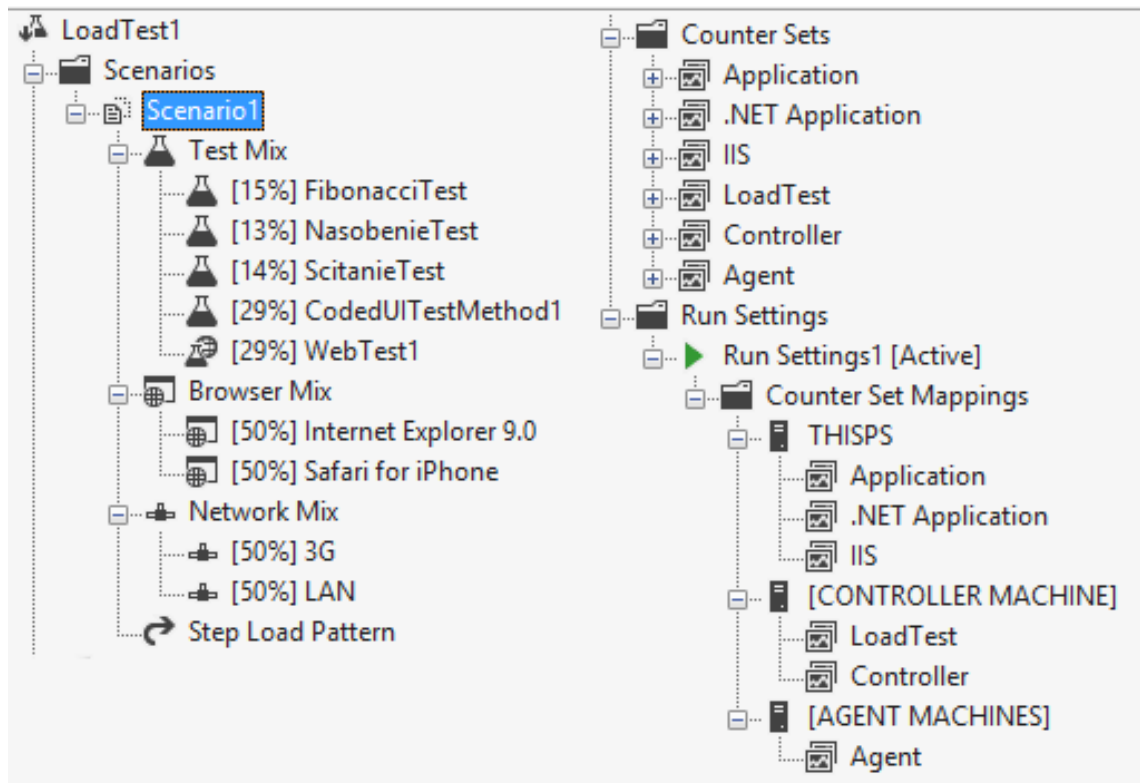
1. *Maximálny časový limit:* Nastaví, ako dlho ma záťažový test bežať. Je možné určiť aj Warm up time, ktorý predstavuje čas na prípravu prostredia na testovanie.
2. *Počet iterácií:* Počet koľkokrát sa jednotlivé testy pustia.

Sekcia Details slúži na nastavenie rýchlosti vzorkovania, a teda nastavenie časového intervalu, v ktorom sa majú zozbierať všetky merané údaje. Napríklad každých 5 sekúnd, ako je zobrazené na obrázku 85.



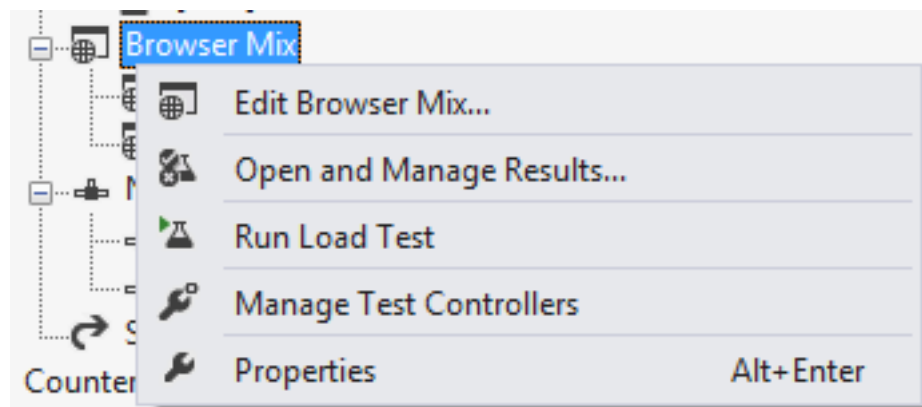
Obrázok 86 Run Settings – Nastavenia behu testu (Vlastná tvorba)

Po kliknutí na tlačidlo Finish sa vytvorí LoadTest1 a automaticky sa otvorí v editačnom režime.



Obrázok 87 Load Test (Vlastná tvorba)

Všetky nastavené vlastnosti (Test mix, Network mix a pod.) je možné dodatočne meniť, a to buď v Properties okne alebo kliknutím pravého tlačidla myši na príslušnú sekciu (napr. kliknutím na sekciu Browser Mix a následným výberom z kontextovej ponuky Edit Browser Mix). Po potvrdení sa opäť zobrazí Wizard (Sprievodca), v ktorom je možné nastavenia meniť presne tak, ako pri ich vytváraní.



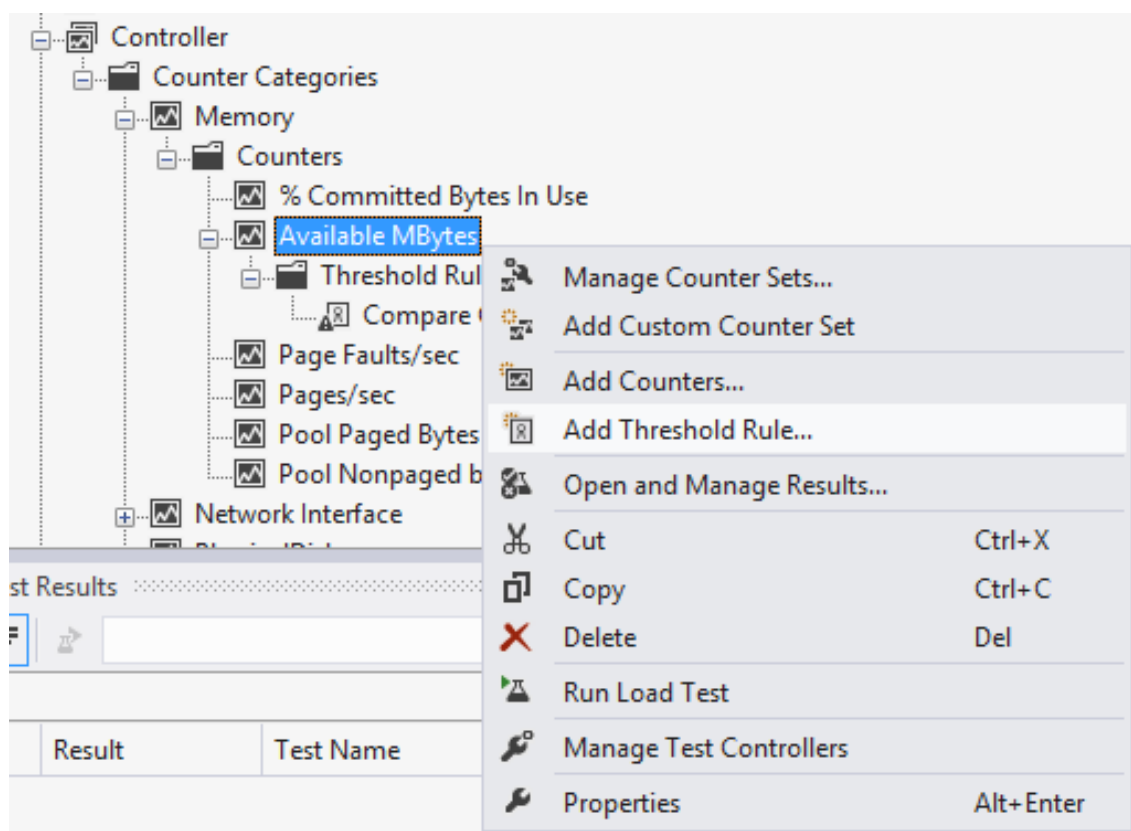
Obrázok 88 Editovanie Zát'azového testu (Vlastná tvorba)

### 6.5.1.9 Threshold rules – Pravidlá rozsahu

Hlavným účelom meračov a sád meračov je zistiť aktuálny výkon aplikácie v rámci testu, využitie pamäte, procesora a podobne. Ak aplikácia beží v poriadku a namerané hodnoty sú v rozsahu (určenom v špecifikácii) počas celého testu, je možné

tento test vyhodnotiť ako Vyhovel (Pass). Pre nastavenie rozsahu (Threshold) slúžia Threshold rules. Stupeň ich prekročenia bude v teste zaznamenaný ako warning alebo error. Tieto hranice rozsahu je možné nastaviť pre každý jeden merač zvlášť.

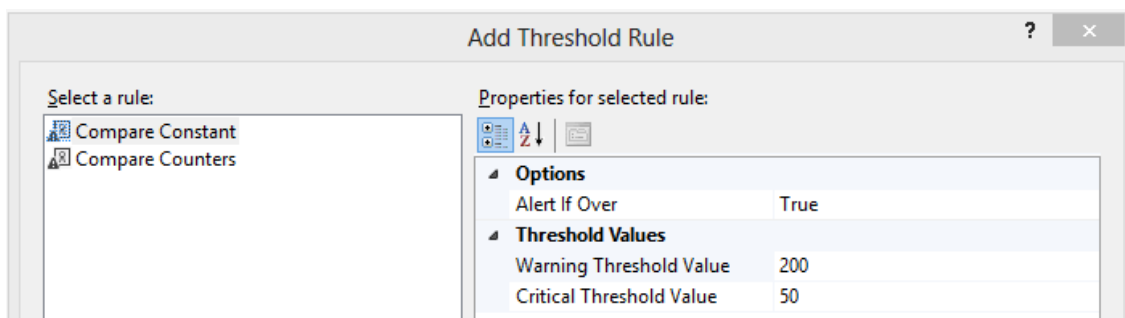
Pre meranie voľnej pamäti v systéme je potrebné rozklenúť príslušný merač na vetve Memory a z kontextovej ponuky na označenej vetve Available MBytes zvoliť možnosť Add Threshold Rule.



Obrázok 89 Pridanie Pravidla rozsahu (Vlastná tvorba)

Po potvrdení sa zobrazí dialógové okno s nastaveniami.

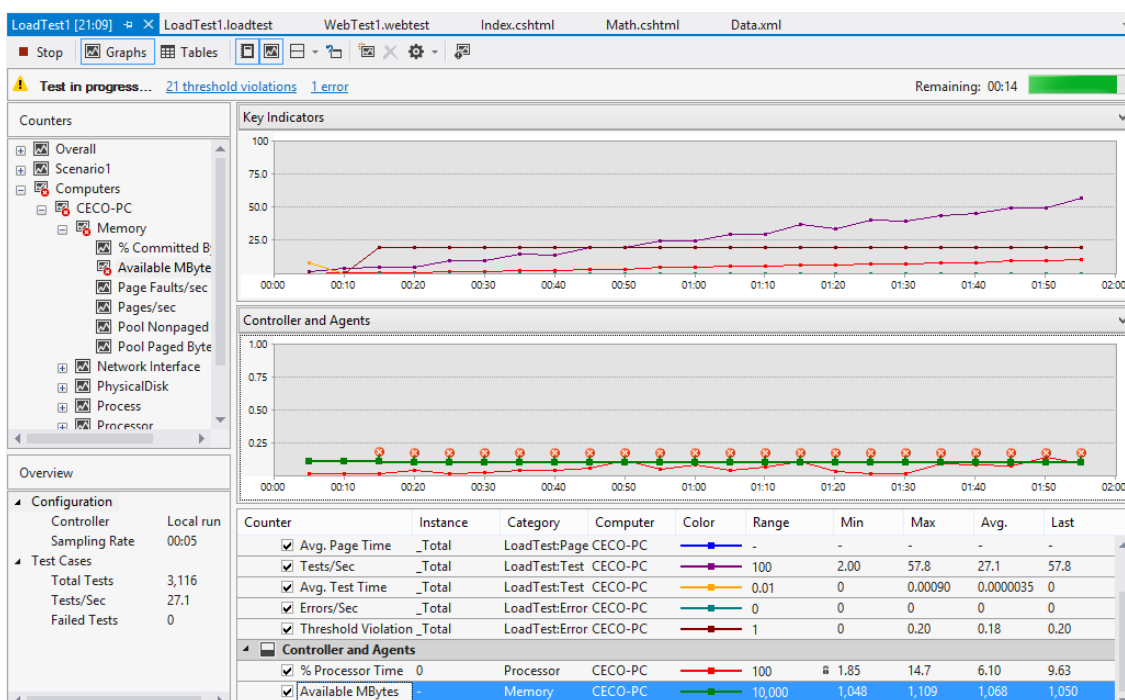
Existujú dva rôzne typy pravidiel, ktoré môžu byť pridané. Jedným z nich je porovnanie konštantnej hodnoty a druhým je porovnanie hodnoty s hodnotou odôvodnenou iného merača výkonu. Je možné voliť medzi dvoma úrovňami varovaní, závislých od veľkosti prekročenia hodnoty: Warning a Critical.



Obrázok 90 Nastavenie Pravidla rozsahu (Vlastná tvorba)

## 6.5.2 Spustenie testu a analýza výsledkov

Záťažový test sa spúšťa rovnako ako všetky iné testy v VS, a to buď z okna Test Explorer možnosťou Run Selected Test alebo z panelu nástrojov záťažového testu Run Test. Po spustení sa zobrazí okno s aktuálnymi hodnotami zobrazenými v prehľadných grafoch s legendou. Nové osy je možné do grafov pridávať cez okno Counters. Ak je prekročený rozsah nameraných hodnôt nastavených v Threshold, sú tieto hodnoty zaznačené v grafe príslušnou ikonou s červeným výkričníkom.



Obrázok 91 Priebeh záťažového testu (Vlastná tvorba)

Zobrazenie grafov je možné meniť a k dispozícii sú 4 základné grafy:

- *Key Indicators*: Pre zobrazenie nameraných údajov ako priemerný čas odozvy, JIT, chyby za sekundu, počet virtuálnych užívateľov atď.



- *Page Response Time*: Graf zobrazuje koľko trvá odpoveď (response) jedného request-u (požiadavky).
- *System Under Test*: Dáta z rôznych počítačov alebo agentov použitých v testoch. Dáta zahŕňajú údaje ako vyťaženie procesoru a využitie pamäte.
- *Controller and Agents*: Zobrazuje detailné informácie o systéme alebo strojoch zapojených do záťažového testu. Taktiež zobrazujú údaje ako vyťaženie procesoru a využitie pamäte, avšak konkrétnych systémov.

Grafy je možné pridávať aj vlastné, a to kliknutím na ľubovoľný už pridaný graf a zvolením možnosti Add Graph.

#### 6.5.2.1 Summary view – Sumárny prehľad

Sumárny prehľad, ktorý slúži na detailnejšie zobrazenie celkových informácií, je možné zobraziť z panelu nástrojov Záťažového testu. Najdôležitejšími informáciami sú zobrazenia piatich najpomalších stránok a piatich najpomalších testov. Zvyšok testov je radený podľa priemerného času. Sumárny prehľad obsahuje:

- ☑ **Test Run Information**: Sekcia zobrazuje celkové údaje záťažového testu ako je dátum a čas začiatku a ukončenia testu, počet agentov použitých na testovanie a celkové nastavenia testu.
- ☑ **Overall Results**: Poskytuje informácie ako napríklad maximálne zaťaženie užívateľmi, počet testov za sekundu, počet request-ov, počet zobrazených stránok za sekundu, priemerný čas odozvy a podobne.
- ☑ **Test Results**: Sekcia zobrazuje informácie o stave testu ako napríklad počet testov pre každý zvolený záťažový test, počet testov, ktoré vyhovelí (Pass) alebo nevyhovelí (Failed).
- ☑ **Page Results**: Táto časť uvádza informácie o rôznych URL adresách použitých v záťažovom teste. Taktiež udáva koľkokrát bola stránka zobrazená, jej priemerný čas odozvy a taktiež čas potrebný pre každú jednu požadovanú stránku.
- ☑ **Transaction Results**: Transakcia je sada úloh v teste. Sekcia zobrazuje informácie ako názov Scenáru, meno testu, čas potrebný pre každú sadu transakcií a počet opakovaní.

- ☑ **System under Test Resources:** Zobrazuje informácie o systémoch zapojených do záťažového testu, vyťaženie procesoru počas testovania, množstvo použitej pamäte atď. Ide o dáta namerané na hostiteľskom PC, teda na PC, na ktorom bol test spustený.
- ☑ **Controller and Agents Resources:** Sekcia zobrazuje informácie o zariadeniach používaných ako Controller (Kontrolér) a zariadeniach použitých ako Agents (Agenti). Zobrazuje informácie o procesore, pamäti a všetky údaje, ktoré boli nastavené ako Counter (merače).
- ☑ **Errors:** Sekcia zobrazuje podrobný zoznam chýb, ku ktorým došlo počas záťažového testu. Zobrazuje typ chyby, počet výskytov, kedy došlo k rovnakej chybe a chybovú správu.

The screenshot displays the 'Load Test Summary' section of a LoadRunner report. It includes a navigation bar at the top with tabs for Summary, Graphs, Tables, and Detail. The main content is organized into several sections:

- Test Run Information:** A table with fields like Load test name (LoadTest1), Start time (11/04/2014 21:09:20), End time (11/04/2014 21:11:20), Duration (00:02:00), and Controller (Local run).
- Key Statistic: Top 5 Slowest Tests:** A table listing tests such as FibonacciTest, ScotanieTest, and NasobenieTest with their 95% Test Time in seconds.
- Overall Results:** A table showing performance metrics like Max User Load (120), Tests/Sec (32.9), Avg. Test Time (sec) (0.00000051), and Avg. Response Time (sec).
- Test Results:** A table summarizing test scenarios (Scenario 1) and their results (Total Tests, Failed Tests, Avg. Test Time).
- Transaction Results:** A table with columns for Name, Scenario, Test, Response Time, Elapsed Time, and Count.
- System Under Test Resources:** A table showing resource usage for machine 'THISPS', including % Processor Time and Available Memory.
- Controller and Agents Resources:** A table showing resource usage for machine 'CECO-PC', including % Processor Time and Available Memory.
- Errors:** A table listing error details, such as 'Exception: LoadTestCounterCategoryNotFound' with a count of 1 and a descriptive message.

Obrázok 92 Detail sumárneho prehľadu (Vlastná tvorba)

Z obrázku je evidentné, že záťažový test LoadTest1 bol spustený 11/04/2014 o 21:09:20 a skončil o 21:11:20. Boli použité nastavenie Run Settings 1 a zároveň bol použitý len 1 testovací agent. V teste boli použité unit testy NasobenieTest, FibonacciTest a ScitanieTest, spustené 1291,1363 a 1297 krát a najpomalší bol FibonacciTest, ktorý trval 54 tisíciny sekundy. Z obrázku 92 sa dá vyčítať, že test zabral 3,96% percenta procesorového času a taktiež zabral 1,020 MB operačnej pamäte.

### 6.5.2.2 Tables view - Tabuľkový prehľad

Tabuľkový prehľad poskytuje súhrnné informácie o výsledku testov v tabuľkovom formáte. V default nastavení existujú dve tabuľky. Prvá zobrazuje zoznam testov použitých v záťažovom teste, koľkokrát boli v rámci testu spustené, koľko ich vyhovelo a koľko nie a koľko testov prebehlo za sekundu. Druhá, spodná tabuľka zobrazuje výhradne chyby, ktoré vznikli počas testu. Zaznamenáva, ktorého testu sa chyba týka a jej detailné chybové hlásenie.

Test	Scenario	Total	Passed	Failed	Tests/Sec	Test Time	95% Test Time
FibonacciTest	Scenario1	1,363	1,363	0	11.4	0	0
NasobenieTest	Scenario1	1,291	1,291	0	10.8	0.0000015	0.0000377
ScitanieTest	Scenario1	1,297	1,297	0	10.8	0	0

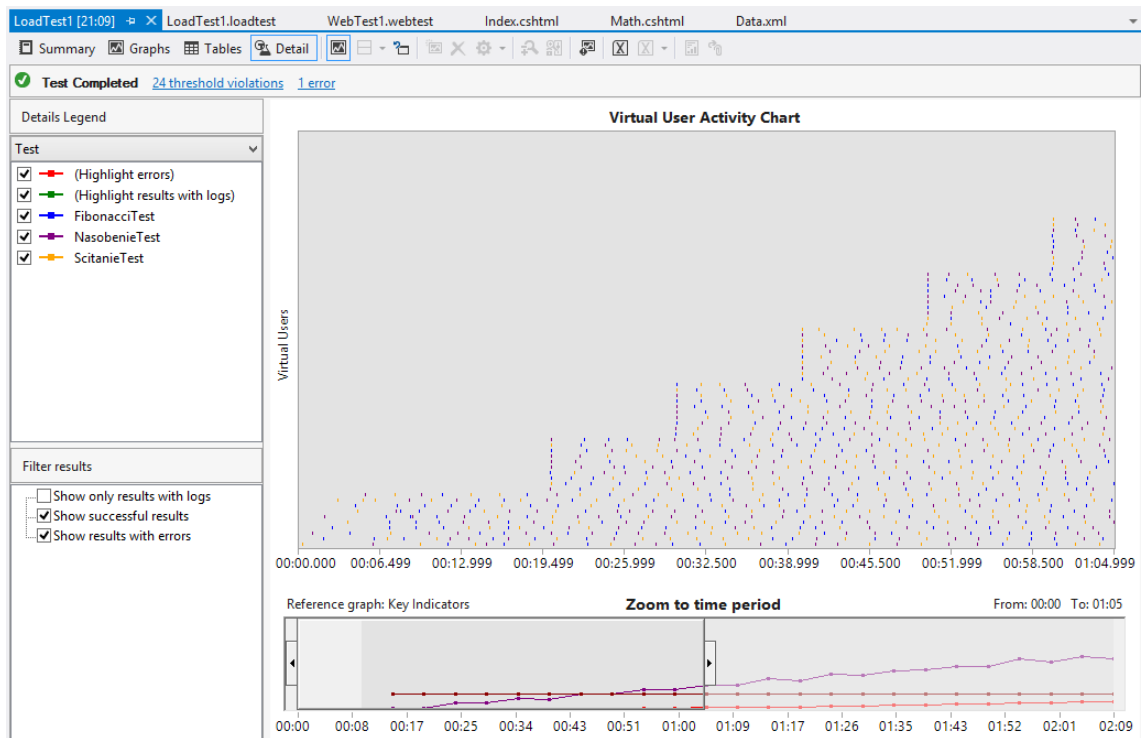
  

Type	Subtype	Count	Last Message
Total		1	
Exception	LoadTestCoun...	1	The performance counter category 'Memory' cannot be accessed on computer 'THISPS' (The net...

Obrázok 93 Detail tabuľkového prehľadu (Vlastná tvorba)

### 6.5.2.3 Detail view - Detailný pohľad

Detailný pohľad zobrazuje aktivity virtuálnych užívateľov v rámci grafu záťažového testu. Graf zobrazuje zaťaženie užívateľmi (User Load), neúspešne alebo prerušené testy a najpomalšie testy. V pravej časti pohľadu je možné zapínanie a vypínanie zobrazenia daného testu v grafe. Spodná časť testu obsahuje časovú os, v prípade selekcie možnosti zobrazenia len určitého časového úseku záťažového testu.



Obrázok 94 Detailný prehľad (Vlastná tvorba)

## 7. Záver

Diplomová práca, postavená na teoretickom základe i vlastnom výskume, sa zameriava na explikáciu problematiky testovania a tvorby automatizovaných testov v nástroji Visual Studio 2012 a optimalizáciu tohto procesu pre zníženie nákladov potrebných na vývoj. Potreba kvalitných a nechybových aplikácií v súčasnosti narastá, a preto sa veľký význam kladie na testovanie a zvyšovanie kvality softvéru. Zavedenie automatizovaných testov do procesu vývoja výrazne znižuje čas identifikovania chýb a náklady na ich odstránenie. Čas vynaložený na tvorbu automatizovaných testov je v komparácii s opakovaným manuálnym testovaním výrazne kratší.

Prvá časť práce je zameraná na definíciu pojmov potrebných na prezentáciu základných princípov testovania. Súčasťou tejto štúdie je objasnenie testovacích techník, ktoré by mali byť využité v rôznych vývojových fázach za účelom maximalizácie efektivity detegovania chýb. Teoretickú základňu uzatvára predstavenie testovacej dokumentácie, ktorá predstavuje nevyhnutnú dokumentáciu pre manažment, plánovanie a riadenie procesu testovania.

Druhá časť práce poskytuje detailný návod ako vytvárať automatizované testy v prostredí Microsoft Visual Studia verzie 2012. Postupne objasňuje vytváranie unit testov v novo vytvorenom Testovacom projekte, tvorbu dátovo riadených unit testov, špecifikuje špeciálne typy unit testov a vysvetľuje problematiku pokrytia kódu. V práci sa nachádza aj interpretácia tvorby Ordered testov v kontexte rozšírenia funkcionality unit testov pridaním podpory radenia a možnej vzájomnej nadväznosti. Druhá časť práce ozrejmuje tvorbu výrazne špecifických testov užívateľského rozhrania, objasňuje problematiku záznamu, a teda tvorby a editácie testu a vysvetľuje štruktúru takéhoto testu.

Majoritná časť práce je venovaná tvorbe webových testov, a to najmä z dôvodu narastajúceho počtu webových aplikácií a presunu klasických aplikácií do Webovej alebo Cloudovej platformy. Práca prezentuje tvorbu webového testu na localhost aplikácii a zároveň popisuje všetky vlastnosti, ktoré je možné už pri vytvorení teste editovať. Záver druhej časti prináša výklad tvorby záťažových testov a interpretáciu ich výsledkov. Hlavným prínosom tejto práce je poskytnutie dôkladného návodu, na základe ktorého je možné zorientovať sa v komplexnej problematike testovania a určiť správnu stratégiu alebo postup pri návrhu testovania pre vyvíjanú aplikáciu či softvér a vytvárať automatizované testy pre všetky oblasti a časti vývojového procesu. Vzhľadom k tomu, že práca poskytuje praktický návod ako vytvárať automatizované

testy a rieši väčšinu potrieb a problémov, ktoré pri tejto tvorbe vznikajú, jej inštruktívny charakter možno využiť v reálnom testovacom prostredí.

V porovnaní s konkurenčnými produktmi, prevažne open source charakteru, prináša Visual Studio niekoľko výrazných pozitív. Ako jediné poskytuje ucelenú platformu pre tvorbu všetkých typov automatizovaných testov, najčastejšie používaných vo fáze vývoja. V súčasnosti sa na trhu nevyskytuje rovnako komplexný open source nástroj, ktorý by poskytoval ucelenú alternatívu k tvorbe unit testov spolu webovými, GUI alebo záťažovými testami, a preto je potrebné používať niekoľko medzi sebou nekompatibilných nástrojov, za dosiahnutím podobného výsledku ako pri použití Visual Studia. Ďalším podstatným pozitívom tohto nástroja je finančný faktor. Nekomerčné nástroje sa kvalitou spracovania, ergonómie, komplexnosti a použiteľnosti nevyrovnajú kvalitám Visual Studia, kvôli čomu je čas na vytvorenie testu dlhší ako čas na vytvorenie testu vo Visual Studiu. V konečnom dôsledku to aj napriek vyššej zriaďovacej cene znižuje náklady na vývoj.

Väčšina nekomorných nástrojov má nedostatočnú, často až absentujúcu dokumentáciu, ktorá je vo väčšine prípadov udržiavaná len malou komunitou bez pravidelnej aktualizácie. Visual Studio poskytuje plnú dokumentáciu, viacero odborných a profesionálnych webových poradní, na ktorých je možné nájsť odpoveď na väčšinu problémov. Microsoft zároveň poskytuje platenú podporu a vzhľadom na bohatú históriu (od roku 1995) je nepravdepodobné, že by vývoj Visual Studia skončil pre nezáujem, ako sa neraz stalo pri nekomerčných open source projektoch. V závere diplomová práca dedukuje, že konkrétny subjekt (podnik, jednotlivec) musí zvážiť, čo všetko očakáva od vývojového nástroja.

## 8. Literatúra

GOUSSET, Mickey. *Řízení životního cyklu aplikací ve Visual Studiu 2010*. Vyd. 1. Překlad Jan Pokorný. Brno: Zoner Press, 2010, 671 s. ISBN 978-80-7413-102-8.

PAGE, Alan, Ken JOHNSTON a Bj ROLLISON. *How we test software at Microsoft*. Redmond, Wash.: Microsoft, c2009, xx, 423 p. Best practices (Redmond, Wash.). ISBN 07-356-2425-9.

BRADER, Larry, Howie HILLIKER a Alan Cameron WILLS. MICROSOFT. *Testing for continuous delivery with Visual Studio 2012*. Vyd. 1. Redmond: Microsoft patterns & practices, 2013. ISBN 978-1-62114-018-4.

Manual testing. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001-, 2014-02-14 [cit. 2014-02-22]. Dostupné z: [http://en.wikipedia.org/wiki/Manual\\_testing](http://en.wikipedia.org/wiki/Manual_testing)

Test automation. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001-, 2011-02-20 [cit. 2014-02-22]. Dostupné z: [http://en.wikipedia.org/wiki/Test\\_automation](http://en.wikipedia.org/wiki/Test_automation)

PATTON, Ron. COMPUTER PRESS. *Testování softwaru*. Vyd. 1. Praha, 2002, xiv, 313 s. Programování. ISBN 80-722-6636-5.

Bertolino, A.: *Software Testing Research: Achievements, Challenges, Dreamse. Future of Software Engineering*, 2007. ISBN 0-7695-2829-5.

PERRY, William E. *Effective methods for software testing*. 3rd ed. Indianapolis, IN: Wiley, c2006, xxvii, 973 p. ISBN 978-076-4598-371.

ZOŠIAK, Peter. *Hodnotenie Kvality Produktu*. Praha, 2013. Seminární práce. Vysoká škola Finanční a správní.

ISO/IEC 9126. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2014-02-23]. Dostupné z: [http://en.wikipedia.org/wiki/ISO/IEC\\_9126](http://en.wikipedia.org/wiki/ISO/IEC_9126)

Software Testing Tutorial. *Software Testing Tutorial* [online]. č. 2013, s. 23 [cit. 2014-02-25]. Dostupné z: [http://www.tutorialspoint.com/software\\_testing/software\\_testing.pdf](http://www.tutorialspoint.com/software_testing/software_testing.pdf)

ISO, Software Product Evaluation - Quality Characteristics and Guidelines for their Use. ISO/IEC IS 9126, Geneva, Switzerland: International Organization for Standardization, 1991.

Systémové a integrační testy. *SW Testování* [online]. [cit. 2014-02-25]. Dostupné z: [http://www.swtestovani.cz/index.php?option=com\\_content&view=article&id=29:systemove-a-integrani-testy&catid=3:zaklady&Itemid=11](http://www.swtestovani.cz/index.php?option=com_content&view=article&id=29:systemove-a-integrani-testy&catid=3:zaklady&Itemid=11)

Fáze a úrovně provádění testů. In: *Testování softwaru: Portál zabývající se problematikou ověřování kvality software. Manuální i automatizované testování.* [online]. 2010 [cit. 2014-02-25]. Dostupné z: <http://testovanisoftwaru.cz/tag/systemove-testovani/>

Integration testing. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001-, 2013-10- [cit. 2014-02-25]. Dostupné z: [http://en.wikipedia.org/wiki/Integration\\_testing](http://en.wikipedia.org/wiki/Integration_testing)

HLAVA, Tomáš. Progresní a regresní testy. In: *Testování softwaru: Portál zabývající se problematikou ověřování kvality software. Manuální i automatizované testování.* [online]. [cit. 2014-02-25]. Dostupné z: <http://testovanisoftwaru.cz/druhy-typy-a-kategorie-testu/progresni-a-regresni-testy/>

HLAVA, Tomáš. Progresní a regresní testy. In: *Testování softwaru: Portál zabývající se problematikou ověřování kvality software. Manuální i automatizované testování.* [online]. 2010 [cit. 2014-02-25]. Dostupné z: <http://testovanisoftwaru.cz/tag/funkcni-testy/>

JANDORA, J. *Řízení SW projektů*. Praha, 2011. Semestrální práce. České učení vysoké technické. Fakulta elektrotechnická.

Druhy testování v procesu vývoje SW. *SW Testování* [online]. [cit. 2014-02-25]. Dostupné z: [http://www.swtestovani.cz/index.php?option=com\\_content&view=article&id=18:druhy](http://www.swtestovani.cz/index.php?option=com_content&view=article&id=18:druhy)

IEEE Standard Glossary of Software Engineering Terminology [online]. 1990 [cit. 2012-11-29]. Dostupné z: <http://www.idi.ntnu.no/grupper/su/publ/ese/ieee-se-glossary-610.12-1990.pdf>



HLAVA, Tomáš. Smoke testy. In: *Testování softwaru: Portál zabývající se problematikou ověřování kvality software. Manuální i automatizované testování*. [online]. 2011. vyd. [cit. 2014-02-26]. Dostupné z: <http://testovanisoftwaru.cz/druhy-typy-a-kategorie-testu/smoke-testy/>

Smoke test. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2014-02-26]. Dostupné z: [http://cs.wikipedia.org/wiki/Smoke\\_test](http://cs.wikipedia.org/wiki/Smoke_test)

Exploratory testing. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2014-02-26]. Dostupné z: [http://en.wikipedia.org/wiki/Exploratory\\_testing](http://en.wikipedia.org/wiki/Exploratory_testing)

Software testing. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001-, 2014-02-26 [cit. 2014-02-26]. Dostupné z: [http://en.wikipedia.org/wiki/Software\\_testing](http://en.wikipedia.org/wiki/Software_testing)

Funkční vs nefunkční testování. *SW Testování* [online]. [cit. 2014-02-25]. Dostupné z: [http://www.swtestovani.cz/index.php?option=com\\_content&view=article&id=22:funkni-vs-nefunkni-testovani&catid=3:zaklady&Itemid=11](http://www.swtestovani.cz/index.php?option=com_content&view=article&id=22:funkni-vs-nefunkni-testovani&catid=3:zaklady&Itemid=11)

PERRY, William E. Effective methods for software testing. 3rd ed. Indianapolis, IN: Wiley, c2006, xxvii, 973 p. ISBN 978-076-4598-371.  
Manual Testing. *Manual Testing* [online]. 2008 [cit. 2014-03-04]. Dostupné z: <http://www.manualtesting.info/>

ČERMÁK, Miroslav. White box test. In: *CleverAndSmart* [online]. 2008-12-13 [cit. 2014-03-04]. Dostupné z: <http://www.cleverandsmart.cz/white-box-test/>

Unit testing. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2014-03-04]. Dostupné z: [http://en.wikipedia.org/wiki/Unit\\_testing](http://en.wikipedia.org/wiki/Unit_testing)

Load testing. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001-, 2014-03-03 [cit. 2014-03-05]. Dostupné z: [http://en.wikipedia.org/wiki/Load\\_testing](http://en.wikipedia.org/wiki/Load_testing)

Stress Testing. JANALTA INTERACTIVE INC. Techopedia [online]. 2010 [cit. 2014-03-05]. Dostupné z: <http://www.techopedia.com/definition/15310/stress-testing>

Volume testing. In: Wikipedia: the free encyclopedia [online]. San Francisco (CA): Wikimedia Foundation, 2001-, 2013-3-16 [cit. 2014-03-05]. Dostupné z: [http://en.wikipedia.org/wiki/Volume\\_testing](http://en.wikipedia.org/wiki/Volume_testing)

Network Sensitivity Tests. *Load Testing by RPM Solutions Pty Ltd: Load Testing and Performance Testing of ANY technology* [online]. 2004 [cit. 2014-03-05]. Dostupné z: [http://www.loadtest.com.au/types\\_of\\_tests/network\\_sensitivity\\_tests.htm](http://www.loadtest.com.au/types_of_tests/network_sensitivity_tests.htm)

Failover Tests. *Load Testing by RPM Solutions Pty Ltd: Load Testing and Performance Testing of ANY technology* [online]. 2004 [cit. 2014-03-05]. Dostupné z: [http://www.loadtest.com.au/types\\_of\\_tests/failover\\_tests.htm](http://www.loadtest.com.au/types_of_tests/failover_tests.htm)

ČERMÁK, Miroslav. Plán testování. In: *CleverAndSmart* [online]. 2009-02-09 [cit. 2014-03-05]. Dostupné z: <http://www.cleverandsmart.cz/plan-testovani/>

ČERMÁK, Miroslav. Testovací případ. In: *CleverAndSmart* [online]. 2009-02-14 [cit. 2014-03-05]. Dostupné z: <http://www.cleverandsmart.cz/testovaci-pripad/>

Microsoft Visual Studio. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001-, 2014-04-03 [cit. 2014-04-15]. Dostupné z: [https://en.wikipedia.org/wiki/Visual\\_studio](https://en.wikipedia.org/wiki/Visual_studio)