**UNIVERZITA PARDUBICE**
**ÚSTAV ELEKTROTECHNIKY A INFORMATIKY**

# VYUŽITÍ TECHNIK EXTRÉMNÍHO PROGRAMOVÁNÍ PRO AUTOMATICKÉ TESTY A TDD

**BAKALÁŘSKÁ PRÁCE**

**2007** **Radim Dubový**

**UNIVERZITA PARDUBICE**
**ÚSTAV ELEKTROTECHNIKY A INFORMATIKY**

# VYUŽITÍ TECHNIK EXTRÉMNÍHO PROGRAMOVÁNÍ PRO AUTOMATICKÉ TESTY A TDD

**BAKALÁŘSKÁ PRÁCE**

**AUTOR PRÁCE: Radim Dubový**
**VEDOUCÍ PRÁCE: Ing. Lukáš Čegan**

**2007**

**UNIVERSITY OF PARDUBICE**
**INSTITUTE OF ELECTRICAL ENGINEERING**
**AND INFORMATICS**

# APPLICATION OF EXTREME PROGRAMMING TECHNOLOGY FOR AUTOMATIC TESTING AND TDD

**BACHELOR WORK**

**AUTHOR: Radim Dubový**
**SUPERVISOR: Ing. Lukáš Čegan**

**2007**


**Vysokoškolský ústav:** Ústav elektrotechniky a informatiky
**Katedra/Ústav:** Ústav elektrotechniky a informatiky
**Akademický rok:** 2006/2007

# ZADÁNÍ BAKALÁŘSKÉ PRÁCE

**Pro: Dubový Radim**
**Studijní program:** Informační technologie
**Studijní obor:** Informační technologie

**Název tématu:** Využití technik extrémního programování pro automatické testy zdrojového kódu

**Zásady pro zpracování:**
Teoretická část bude obsahovat vysvětlení pojmů extrémní programování, automatické testy, Test-Driven Development a pojmů dalších souvisejících témat. Dále bude popsána teorie psaní automatických testů a programové nástroje k tomuto testování vyvinuté a také stručný popis jejich funkcionality. Praktická část bude obsahovat naprogramování webové aplikace pro firmu Baader Computer s implementací automatických testů pomocí nástroje NUnit nebo NUnitAsp (nebo podobného), to znamená vytvoření automatického testu na vytvořené třídy a ASP.NET stránky.

**Seznam odborné literatury:**


- Jančen D., Saiedian H., *Test-Driven Development: Concepts, Taxonomy and Future direction*.
- Amber W. S., *Introductin to Test-Driven Development (TDD).*
- Beck K., *Programování řízené testy*. Grada, 2004.


**Rozsah:** 30 stran
**Vedoucí práce:** Ing. Čegan Lukáš
**Vedoucí katedry (ústavu):** prof. Ing. Pavel Bezoušek, CSc.
**Datum zadání práce:** 30. 11. 2006
**Termín odevzdání práce:** 18. 5. 2007

Prohlašuji:

Tuto práci jsem vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorský zákon, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Souhlasím s prezenčním zpřístupněním své práce v Univerzitní knihovně Univerzity Pardubice.

V Pardubicích dne 3. 5. 2007

Radim Dubový

(vlastnoruční podpis)

# ABSTRAKT

Tato práce se zabývá problematickou automatických testů jako metodiky při psaní programového kódu, která zefektivňuje využití času nejen při dalších úpravách kódu, ale hlavně při nalézání chyb a při dalších aktivitách s programováním spojených.

Techniky extrémního programování se stávají ve světě čím dál známější a tato práce popisuje výhody a nevýhody při zavedení automatických testů. Dále na konkrétním příkladu částečně aplikuje tuto metodiku, používá k tomu vybrané nástroje a na základě těchto skutečností vyvozuje závěry o tom, kdy se vyplatí a kdy ne.

# Obsah

## Seznam obrázků

# 1. Úvod

Při psaní programového kódu často vzniká mnoho chyb a jejich odhalení bývá obtížné, těžko simulovatelné nebo dokonce tak specifické, že je výsledkem shody náhod. Při použití mnoha na sobě závislých souborů nebo souborů používajících stejnou třídu objektů, může zas při jedné změně této třídy za jedním účelem být pozměněna funkcionalita v jiné části kódu a program potom může vracet nesmyslné výstupy. Pro částečnou kontrolu mohou posloužit automatické testy, což je jeden z principů extrémního programování jako metodiky tvorby programového kódu.

Automatické testy tedy slouží pro kontrolu správně vytvořeného programového kódu. Pokud test nedopadne úspěšně, znamená to pro programátora okamžitou informaci, že kód je vytvořen s chybou a je nutné ji odstranit. Automatický test by měl chybu přibližně lokalizovat a ušetřit tak čas strávený při jejím hledání.

Mým úkolem je ve firmě Baader Computer aplikovat techniku extrémního programování, nazvanou Test-Driven Development (Programování řízené testy), pro snadnou a spolehlivou informaci, zda je v rámci webové aplikace SQL dotaz napsán správně, respektive zdali vybírá správné záznamy z databáze. To rychle a téměř bez práce poskytne programátorovi dostatečné informace o tom, zda je někde problém (a chyba musí být napravena) nebo je vše v pořádku a tím pádem se může program nasadit do ostrého provozu nebo se může pokračovat v jeho dalším vývoji.

Vedlejším a nepovinným cílem mé práce je otestovat správnou funkčnost vytvořených tříd, například serverových ovládacích prvků, klientských skriptů atd.

Má bakalářská práce by měla vyhodnotit aspekty psaní testů k programovému kódu a dát podnět k zájmu/nezájmu o tuto metodiku programování.

# 2. Extrémní programování

## 2.1. Metody vývoje software

Od samotného počátku, co lidé vytvářejí počítačové programy, existují snahy o ovládnutí vývojářského chaosu s cílem učinit z vývoje software předvídatelný proces. S rostoucím využitím informačních technologií ve všech oblastech lidské činnosti neustále rostly nároky na funkčnost a kvalitu software, který v důsledku nabýval na velikosti a složitosti, a softwarové projekty přestaly být zvladatelné.

Snaha o nastolení řádu vedla k formulování různých metodik vývoje software, což si lze představit jako souhrn pravidel a postupů, kterými bychom se měli řídit, pokud chceme vývoj programu dotáhnout do úspěšného konce. Existují různé metodiky vývoje software, lišící se složitostí a způsobem řízení vývoje, od neformálních (agilních) až po podrobně strukturované (normativní).

Nejpopulárnějšími metodami vývoje software posledních let jsou bezesporu agilní metody vývoje software. Asi nejznámější z nich je extrémní programování. Klíčovou součástí extrémního programování je i technika vývoje řízeného testy (TDD). Ačkoli obliba extrémního programování jako celku postupně upadá, princip vývoje řízeného testy má šanci udržet se na výsluní delší dobu.

## 2.2. Co je extrémní programování

Extrémní programování (XP) je metodologie vývoje softwaru a programování, která velmi ovlivnila celé softwarové inženýrství. Zavedl ji Kent Beck v knize Extreme Programming Explained. Jedná se o souhrn jednoduchých praktik, které mají dohromady velký společný účinek.

Extrémní programování používá osvědčené a známé principy a postupy vývoje software, dotahuje však jejich použití do extrémů. Extrémní programování ctí tyto hodnoty: jednoduchost, komunikace, zpětná vazba, odvaha.

- základem vývoje je neustálá revize zdrojového textu programů, aby nedocházelo k tzv. „profesionální slepotě“, pracují na jednom zdrojovém kódu vždy dva programátoři (párové programování)
- testování je důležité – proto se program neustále testuje. Testovací rutiny jsou součástí kódu (tento testovací kód někdy přesahuje svým rozsahem vlastní výkonný kód) a program prakticky při každém spuštění testuje, zda se v průběhu vývoje nepoškodil.
- neustále se provádí refaktorování – ověřování, zda návrh programu je správný a snaha o jeho zjednodušení a odstranění duplikací v kódu
- program se udržuje na co nejmenší úrovni složitosti – vždy se programuje jen to, co je v danou chvíli nezbytné
- neustále se testuje integrace jednotlivých komponent – konečný program se i několikrát denně sestavuje a testuje se, zda všechny komponenty spolupracují tak, jak mají
- vývoj probíhá v krátkých iteracích – vždy se vyřeší jedna konkrétní změna programu a okamžitě se ověří, zda všechno pracuje jak má

Klasický životní cyklus vývoje software je rozdělen do těchto fází:

- Zjištění a formulace požadavků (plánování), což představuje:
  - plánování vydání tvoří časový harmonogram
  - časté vydávání malých změn
  - měří se aktuální rychlost vývoje
  - projekt je rozdělen do iterací
  - každá iterace začíná plánováním
  - rychlé schůze, nejlépe ve stoje
  - "sprav to, když se to rozbije"

- Návrh architektury systému (design):
  - ceněná je jednoduchost
  - pro systém musí existovat metafora
  - pro design se používají CRC kartičky
  - funkčnost není přidávána předčasně
  - časté refaktorování (kdykoliv a kdekoliv)
- Programování (kódování):
  - zákazník vždy spolupracuje
  - zdrojový kód musí odpovídat firemní kultuře
  - nejdřív se píší testy jednotek (unit testy)
  - všechen kód programují dva programátoři
  - integraci provádí v jednu chvíli pouze jediný pár programátorů
  - integrace probíhá často
  - zdrojové kódy vlastní všichni programátoři
  - optimalizace se provádí až nakonec
  - žádné pracovní přesčasy
- Testování:
  - všechen kód má testy jednotek (unit testy)
  - všechen kód musí projít testy jednotek, před tím než je vydán
  - když se najde chyba, vytvoří se nové testy
  - akcepční testy se spouští často a výsledky se zaznamenávají
- Předání zákazníkovi
- Provoz a údržba
- Stažení z provozu

V mnoha případech se jedná o zdlouhavý a nákladný proces a nalezení defektů během pozdních fází vývoje může mít fatální následky pro

celý projekt. Aby se vývojáři vyvarovali takových chyb, je nutné, aby měli možnost průběžné kontroly, zda se neodchýlili od požadavků na výsledný produkt.

## 2.3. Programování řízené testy

Programování řízené testy (Test-Driven Development, TDD, neformálně známo jako tvorba automatických či zautomatizovaných testů) je programovací technika, která zahrnuje nejprve napsání testu podle toho, co si od výstupů slibujeme a až potom implementaci takového kódu, který test splní. Toto testování má velmi rychlou zpětnou vazbu. TDD se začalo rozvíjet okolo roku 2000 jako součást extrémního programování, ale v současnosti se uplatňuje i jako samostatná cesta. Programátoři často upozorňují, že se jedná o celou metodu návrhu software, ne jen o metodu testovací.

Společně s ostatními technikami lze tento koncept též aplikovat na odstranění softwarových chyb a vylepšení již existujících aplikací, které nebyly vyvíjeny tímto způsobem.

Test si lze představit jako sadu podnětů a očekávaných odezev systému. Při testování ověřujeme, že program se chová korektně – pro dané vstupy vrací správné výstupy. Testy obsahují pravdivé/nepravdivé tvrzení (tzv. **assertion**), které musí být splněno/nesplněno. Spuštění testu dává programátorovi okamžitou informaci o korektním či nekorektním chování programu během jeho vývoje a refaktorizace kódu. Po proběhnutí testu bychom tedy měli obdržet jednoznačnou odpověď (úspěch/neúspěch).

Testovací frameworky založené na konceptu xUnit poskytují mechanismy pro tvorbu a spouštění automatických testů.

## 2.4. Základní druhy testů

Jako první měřítko rozdělení je způsob, jakým se testy provádějí. Podle tohoto měřítka lze testy rozdělit na automatizované a manuální. Popularita testování vzrostla hlavně díky vývoji nástrojů, které umožňují jednoduchý vývoj automatizovaných testů a jejich spouštění ve velkém

měřítku. Vývojář tak má možnost si během chvíle a často stiskem jednoho tlačítka opakovaně ověřit, že aplikace splňuje požadavky dané testy.

Druhé hledisko rozdělení je podle části systému, kterou testujeme. Prvním a nejjemnějším druhem testu je test jednotky (unit test). Jednotka je nejmenší částí systému. Různí se názory na to, co lze považovat za jednotku systému. Pro objektově orientované programy to může být buď třída, nebo metoda. Další druh testu se nazývá funkční test. Funkční test se používá pro testování větších částí systému (modulů) a pohlíží na systém zvenku – jeho zájmem není vnitřní implementace testovaných operací (black box – tzv. testování černé skříňky). Posledním druhem testu je test integrační, který testuje součinnost systému jako celku.

Další rozdělení existuje podle toho, kdo test provádí. Tak můžeme rozdělit testy na:

- Programátorské testy – automatizovaný test napsaný programátorem zajišťuje, že program je funkční. Funkční kód je napsán, aby pouze prošel testem, nic víc, nic míň. Tyto testy jsou známy jako již zmiňované unit tests (jednotkové testy).
- Zákazníkovy testy – test napsaný inženýry k validaci (ověření) funkce celé aplikace z pohledu zákazníka. Tyto testy jsou známy jako testy přijatelnosti (acceptance tests).

Pro účel mé práce jsem se zabýval pouze programátorskými testy, protože funkce webových aplikací firmy Baader Computer bývají jednoduché a navíc soustavná komunikace se zákazníkem riziko nepochopení problému omezuje na minimum. Zákaznické testy se provádějí až těsně před uvedením aplikace do provozu (a to manuálně) a případné chyby jsou z valné většiny včas odhaleny a opraveny.

## 2.5. Test-last versus test-first přístup

Tradiční přístupy vývoje předpokládají existenci kódu programu před tím, než bude možné testovat a vytvořené testy využívají pouze pro

účely kontroly, zda se projekt neodchýlil od zadání. V odborné literatuře se tento způsob označuje jako **test-last**.

Naopak při programování řízeném testy (TDD), vytváříme testy před tím, než máme dostatek kódu, který by testy prošel (**test-first** přístup). Účel vytváření testů v TDD není kontrola, zda kód souhlasí se specifikací, ale testy zde představují nástroj pro návrh systému. Ačkoliv se to nezdá, kořeny test-first přístupu, lze nalézt i, z pohledu IT, v dnes již vzdálené minulosti. Některé modely životního cyklu software z 80. let 20. století zahrnovaly testování v počáteční fázi projektu a některé zdroje tvrdí, že prvky TDD použila americká Národní agentura pro letectví a kosmonautiku (NASA) v projektu Mercury již v 50. letech 20. století.

Mnoho vývojářů používá test-first přístup aniž by si to uvědomovali. Kent Beck, popularizátor extrémního programování a TDD, sám tvrdí, že test-first přístup se naučil již v mládí, při čtení knihy o programování. Kniha radila stanovit si nejdříve vstup programu a k němu určit očekávaný výstup. Potom bylo třeba programovat tak dlouho, dokud program neposkytoval očekávané výstupy.

## 2.6. Refaktorování

Refaktorování je další významnou technikou, používanou při agilních formách vývoje. Cílem refaktorování je čistý kód, který funguje. Refaktorování je úprava zdrojového kódu systému po malých izolovaných změnách, které nemají za následek změnu funkčnosti. Zachování funkčnosti lze během refaktorování kontrolovat opakovaným spouštěním testů. Refaktorování může výrazně zlepšit čitelnost kódu, odstraní duplicity a usnadní tak budoucí údržbu a rozšiřování systému. Protože se jedná o malé definované a kontrolované změny, lze je provádět i automatizovaně pomocí nástrojů pro refaktorování, dnes již integrovaných do mnoha populárních vývojových prostředí, což výrazně zvyšuje produktivitu vývojáře.

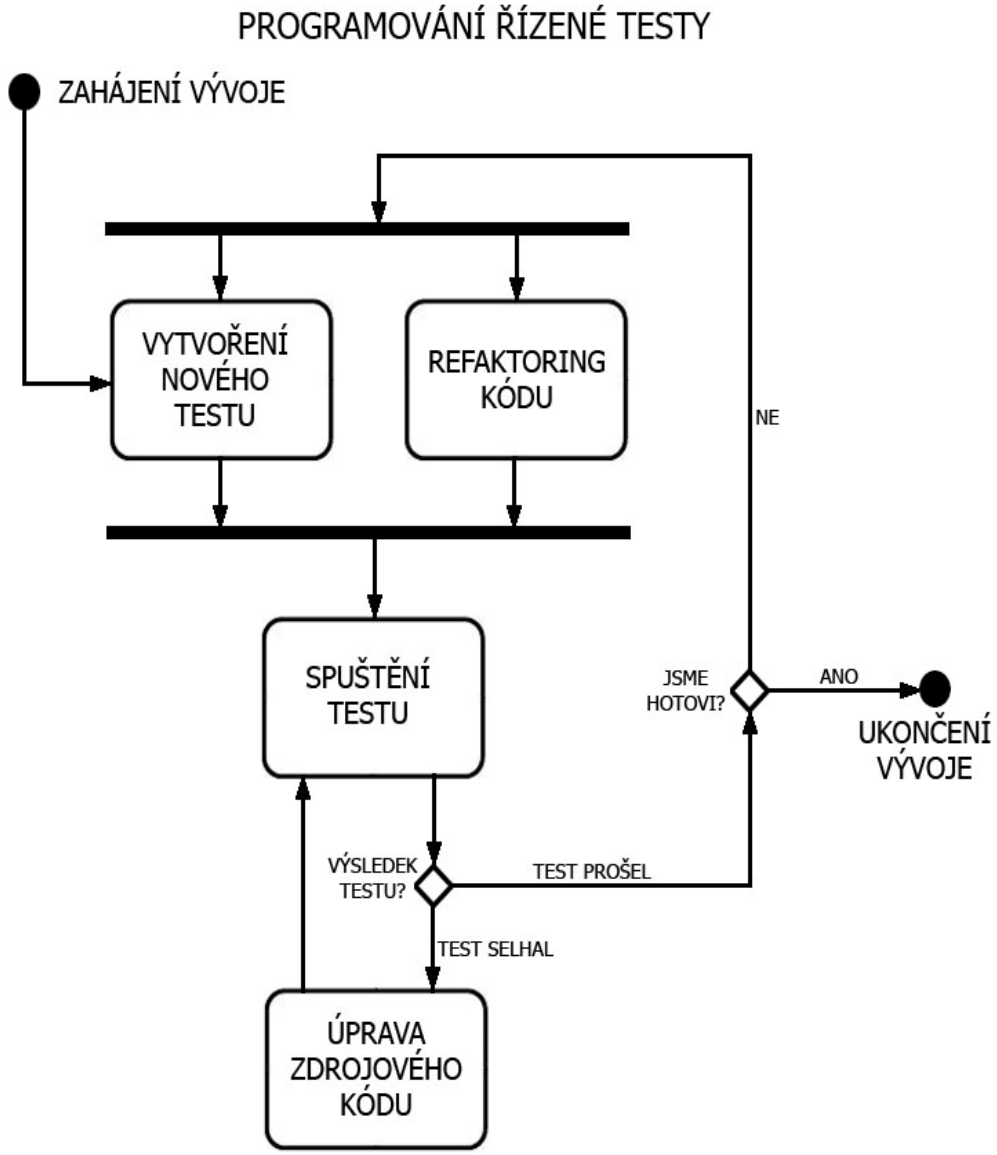
## 2.7. Cyklus při psaní automatických testů

Podle Kenta se TDD definuje dvěma jednoduchými pravidly:

- Nikdy nenapište ani jeden řádek kódu, aniž byste neměli automatizovaný test, který neprojde.
- Eliminujte duplikace.

Ve většině vývojových cyklech vlastník produktu nebo projektový manažer poskytne sadu požadavků. Typicky jsou to požadavky v dokumentu napsaném ve Wordu a jsou pojmenovány jako specifikace funkčnosti. Dokument ve Wordu je vynikající pro popsání příběhu, ale to, co potřebujeme my, je popsání příběhu projektového manažera do programovacího jazyka – například nové požadavky jsou napsány v C# (nebo ve Visual Basicu). Požadavky jsou napsány v kódu jako test a každý takový test reprezentuje jeden požadavek. Dále, když pak chceme dopsat později funkčnost, musíme se ujistit, že jsme eliminovali jakýkoliv duplicitní kód (kód určité funkčnosti se nachází pouze na jednom místě).

Princip práce při programování řízeném testy, vyobrazený na obrázku č. 1, se dá shrnout do tohoto dobře zapamatovatelného sousloví: **červená, zelená, refaktorování**. Názvy barev v tomto sousloví mají souvislost s uživatelským rozhraním běžně používaným v nástrojích pro automatizované spouštění testů (například NUnit, JsUnit atd.), kde červený proužek znamená, že test selhal a naopak zelený svědčí o úspěšnosti provedení testu.




**Obrázek 1: Vývoj řízený testy**

Následující sekvence je založena na knize Programování řízené testy [1] od Kenta Becka, která je považována za prvotní text související s konceptem automatických testů v jeho moderní formě. Popišme si tedy základní kroky podrobněji:

6) Napsání testu

Aby mohl být test napsán, musí vývojář jasně porozumět specifikaci a požadavkům programované součásti (podle dokumentace k zadání, komunikace se zadavatelem nebo s vedením firmy).

7) Spuštění všech testů

Napoprvé je jasné, že nový test určitě selže nebo nepůjde vůbec zkompilovat. Tím pádem se ujistíme, že test náhodou neprošel, aniž bychom napsali jediný řádek kódu.

8) Implementace testované logiky – psaní kódu

Napíšeme kód, o kterém předpokládáme, že úspěšně projde testem. Nový kód pravděpodobně nebude v této etapě dokonalý, i když testem možná projde. To je přijatelné, protože v dalších krocích ho budeme zdokonalovat a vylepšovat. Je důležité vědět, že napsaný kód je navržen hlavně k tomu, aby prošel testem – žádná další funkcionalita by neměla být již dopředu přidávána.

9) Spuštění všech testů

Pokud všechny testy projdou, programátor může být přesvědčený o tom, že kód splňuje všechny testované požadavky a je připraven k refaktorizaci. Pokud ne, musí být kód odpovídajícím způsobem změněn a vracíme se ke kroku 3.

10) Refaktorování

Pomocí refaktorování a opakovaného spouštění testu se kód upraví do přijatelné podoby (samozřejmě, že se neupraví sám).

Programování řízené testy se dá vyjádřit jako rovnice:

**programování řízené testy = test-first přístup + refaktorování.**

## 2.8. Výhody a nevýhody TDD

TDD je, spíše než způsob testování, programovací technika s vedlejším efektem, který zaručuje, že veškerý kód je řádně pokryt testy jednotek (unit testy). Pro kompletní testování je třeba vzít v úvahu ještě funkční a integrační testy.

Tradiční testování, pokud je úspěšné, vede k odhalení defektů. V TDD je to stejné, pokud test selže, jedná se o pokrok, protože víme, kde je třeba vyřešit problém. Důležitější je, že můžeme brát za jasný znak úspěchu, pokud test po opravení defektu již nikdy neselže. TDD zvyšuje

důvěru v to, že systém funguje správně a odpovídá požadavkům, čímž dodává vývojáři odvahu vrhnout se do dalších změn bez obav, že nové změny způsobí selhání systému. Pokud nové změny způsobí defekt, bude odhalen brzy díky testům.

Programátoři, používající pravé TDD u nových projektů, udávají, že měli jen velmi zřídka pocit nutnosti spouštět debugger. TDD pomáhá sestavovat software rychleji a lépe. Nabízí více než pouhou validaci správnosti – současně také kontroluje návrh programu. Protože programátor se zaměřuje na testy, musí si umět představit, jak by měl program používat uživatel, a proto se zabývá hlavně uživatelským rozhraním a méně implementací.

I když napíšeme více otestovaného kódu než neotestovaného, celková implementace trvá obvykle kratší dobu, než použitím ostatních metodik. Velké množství testů limituje počet chyb v kódu. Včasné a časté používání testů napomáhá zachytit vady již v začátcích vývojového cyklu a chrání projekt před tím, aby se stal nezvladatelným a zbytečně drahým a vyhneme se tím únavnému debuggování v pozdějších fázích vývoje, kdy už je množství programového kódu nepřehledné.

Avšak stejně jako lze pomocí TDD vyprodukovat kvalitní a vhodný kód, může se stát, že vytvoříme nekvalitní a zbytečný kód. Tým programátorů by měl používat také jiné techniky, jako jsou projektové diskuze a revize kódu, aby se přesvědčili o tom, že jdou po správné cestě.

TDD ověřuje správnost jen takových případů, na které jsou napsány testy. Nesprávný test, který špatně reprezentuje funkcionalitu, vyprodukuje chybný kód. Výsledkem je, že TDD je jen natolik účinné, nakolik jsou dobré jeho testy, a proto by mělo být použito vždy v kombinaci s ostatními technikami. Automatické testování by se nemělo realizovat v odvětvích typu kryptografie, umělá inteligence, bezpečnost atd. a je též obtížné ho použít ho v určitých situacích, například pro testování grafických uživatelských rozhraní a relačních databází, kdy systémy s komplexním vstupem a výstupem nebyly navrženy pro oddělené testování jednotek nebo refaktoring.

## 2.9. Mock objekty

Testování pomocí mock objektů znamená vlastně techniku psaní určitého druhu automatických testů. V podstatě se jedná o nahrazení reálného objektu testovací fasádou, která neprovádí žádnou funkcionalitu nahrazovaného objektu - jen se jako tento objekt tváří. Místo původní logiky objektu je vloženo chování, které ve svém testu potřebujete.

Nejlepší je ukázat si to na příkladu. Při psaní automatických testů programátor často narazí na situaci, kdy k napsání jednoduchého testu je třeba velmi složitě a pracně připravovat okolní podmínky. Např. testuje se metoda v objektu, který se dotazuje interně DAO objektů na data v databázi (DAO = Data Access Objects – objekty pro přístup do databáze). To ale znamená, že před tím, než se začne testovat logika tohoto objektu, musí se připravit správná data v databázi. Také se musí zajistit, že se tam omylem nedostanou další data, která by test zhatila (např. v SELECT dotazu vrátila více řádků) a tudíž je obvykle třeba zase po sobě uklízet. Ošetření okolních podmínek spuštění testu pak nakonec může stát několikrát více času, než je potřeba k napsaní vlastní testovací logiky.

Existují různé techniky, jak toto zajistit (např. s použitím HSQL databáze, která byla celá v paměti a po každém testu se kompletně zrušila a před novým opět vytvořila a dosadila se příslušná testovací data – tím, že se vše odehrávalo pouze v paměti byly testy velmi rychlé). Ovšem lehčí je právě použít techniku mock objektů.

Při použití této techniky se programátor soustředí na testování pouze logiky objektu a předpokládá, že okolní objekty fungují správně tak, jak mají. Potom se v našem příkladě za DAO objekty dosadí pouze "prázdné" ulity, které mají stejné rozhraní, ale místo vlastní logiky obsahují logiku takovou, kterou potřebujeme pro vlastní test. Tzn. ve chvíli, kdy se objekt zeptá DAO na konkrétní data, neproběhne dotaz do databáze, ale jsou mu rovnou vrácena data, která v daném testu potřebujeme.

### 2.10. Zhodnocení

Vývoj řízený testy je technika, která je v současné době hojně využívaná a její obliba roste. Důvod její obliby tkví ve faktu, že úsilí nutné k jejímu osvojení není tak veliké, jako u extrémního programování, jehož je TDD součástí. Vývojář sice musí mít z počátku disciplínu k tomu, aby testy opravdu psal a spouštěl, ale bez toho by se nejednalo o programování řízené testy. Díky jejímu charakteru, je možné její použití v kombinaci s různými technikami vývoje na nižší úrovni (např. v rámci jedné iterace) a i ve větším spektru projektů.

Je důležité si uvědomit, že se jedná o obecnou techniku a je tedy možné ji aplikovat při programování v jakémkoliv programovacím jazyce. Omezení může nastat pouze v případě, že pro nový programovací jazyk ještě neexistuje příslušný testovací software, ale pokud se technologie hojně rozšíří, tak vždy řádově v měsících se takový software objeví (jak se tomu stalo poměrně záhy po nástupu technologie .NET, kdy vznikla možnost testovat webové stránky). V současné době existuje pro všechny důležité programovací jazyky nějaký software, který umožňuje provádění automatických testů. Proto nikomu nic nebrání v tom, aby se do principu TDD ponořil.

## 3. Projekt

### 3.1. Firma Baader Computer

Téma mojí bakalářské práce mi bylo zadáno firmou Baader Computer sídlící v Praze, pobočkou v Šumperku, která poskytuje svým zákazníkům služby v oblasti informačních technologií, a to od vytvoření konceptu, příp. provedení analýzy společně se zákazníkem až po implementaci informačního systému včetně následných služeb ve formě správy systému nebo podpory koncových uživatelů. Webové aplikace a databázové projekty jsou vyvíjeny hlavně pro automobilové firmy a jejich prodejce jak v naší republice, tak v zahraničí (ze seznamu zákazníků lze jmenovat na příklad firmy Škoda, Peugeot, Seat, GE Money Bank).

## 3.2. Zadání projektu

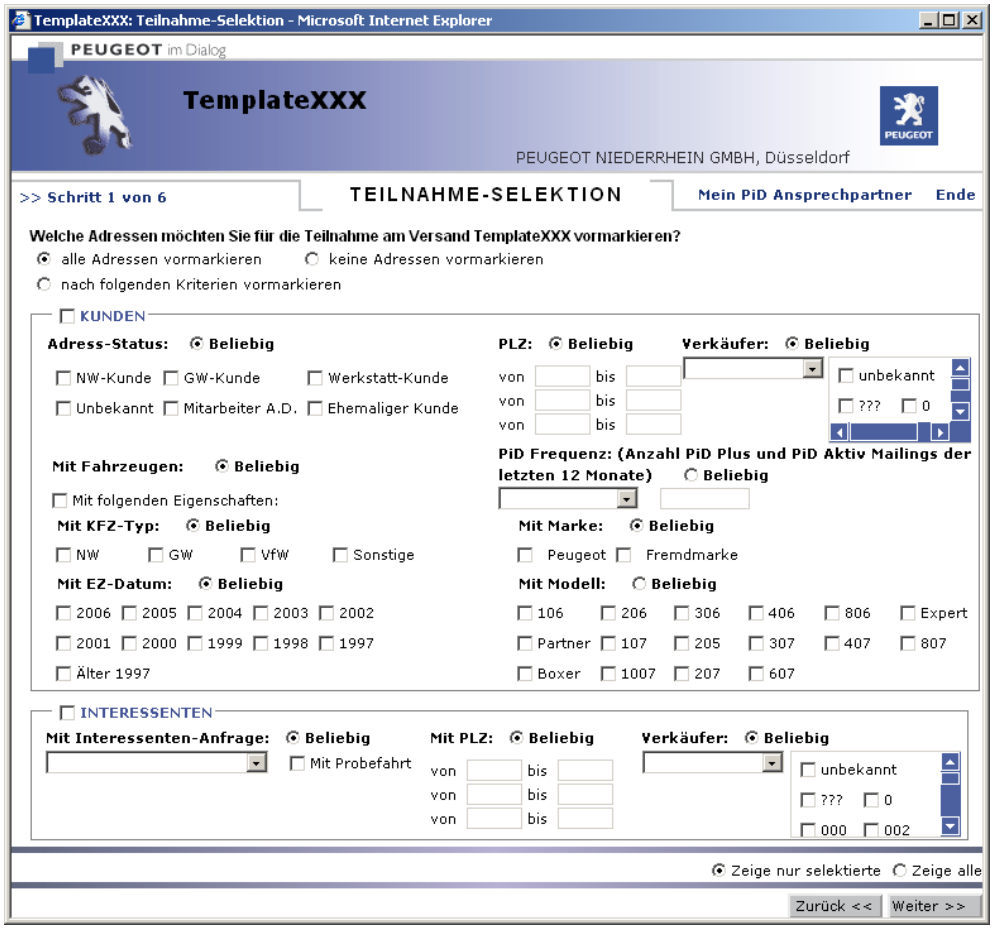
Protože většina nově vytvářených aplikací firmy Baader Computer vychází z podobných požadavků, jakožto i grafické úpravy, vždy se využívá již hotového programového kódu a detaily jsou upravovány dle konkrétních specifikací dané zakázky. Při procesu modifikace však nezřídka vzniknou mnohé chyby, které nejsou na první pohled zjevné. Jednou takovou chybou byl například výběr nežádoucích záznamů z databáze.

Cílem mého projektu je vytvořit webovou aplikaci, která bude sloužit jako šablona pro další vyvíjené aplikace, přičemž bude aplikováno zautomatizování za účelem zrychlení vývoje a zjednodušení odhalování chyb (kdy v budoucích aplikacích se stejně jako některé části zdrojového kódu změní i automatické testy). Webová aplikace využívá technologie ASP.NET, takže je potřeba najít vhodný nástroj pro implementaci testů.

Obecné zadání konkrétní webové aplikace se vždy řídí stejným schématem a má sloužit pro pověřené zástupce automobilek, kteří jejím prostřednictvím vybírají adresy svých koncových nebo potencionálních koncových zákazníků z databáze podle vybraných kritérií a poté s tímto výběrem provádí požadované operace. Protože struktura databáze koncových zákazníků, ze které jsou vybíráni, je poměrně složitá, tak jedním z problémů je výběr jen takových zákaznických adres, které jsou skutečně požadovány (dle uživatelských voleb, které jsou odesílány z webové stránky na server) – to v podstatě znamená, že při tvorbě složitého dotazu v jazyce SQL může někde nastat chyba.

Pro další pokračování v projektu by bylo vhodné zhruba vysvětlit, k čemu vlastně webová aplikace má sloužit a jak má pracovat. Jedná se o klasickou architekturu klient-server, kdy klient se pomocí přihlášení připojí na zabezpečené webové stránky a zde pomocí webového uživatelského rozhraní provádí různé volby (uživatelské rozhraní filtrovací stránky by mělo vypadat jako na obrázku č. 2). Jedná se vlastně o jakéhosi průvodce, jehož některé kroky bývají v mnoha aplikacích shodné a jiné

se zásadně mění, ale vždy se pracuje s velkým množstvím záznamů v databázi. Jsou generovány automatické skripty a SQL dotazy, které změní v databázi příslušné hodnoty. Konkrétně se jedná o zadávání kritérií pro výběr určitých záznamů. A zde nastává hlavní problém. Složitý SQL dotaz by někdy mohl vydat na desítky až stovky řádků a je tedy těžko kontrolovatelný.




**Obrázek 2: GUI webové aplikace - stránka s filtry**

Účelem práce není napsat testy pro všechny použité třídy (mezi kterými jsou i sdílené soubory, o kterých se předpokládá, že jsou správné, protože jsou již dlouhou dobu používány), ale pouze zkontrolovat výběr správných záznamů z databáze. Nesoustředíme se tedy na kompletní kontrolu a přísné používání principů extrémního programování, ale jen na jednu část programu. Je to sice poněkud netypické použití automatických testů, protože jazyk SQL není sám o sobě programovací jazyk nýbrž pomůcka pro přístup k databázi, ale kontrolou kódu psaného

v jazyku C# můžeme ovlivnit vytvářený SQL dotaz, který po spuštění změní určitý počet záznamů.

## 3.3. Zdůvodnění TDD pro tento projekt

Technika automatických testů je pro firmu nová, takže není dost dobře známo, co lze od tohoto postupu očekávat. Výsledkem mojí práce v této oblasti má být rovněž zjistit, zda se čas strávený tvorbou automatických testů vyplatí a jestli má tato metodika efektivní přínos pro firmu (a bude i nadále využívána), nebo je to ztráta času, neboť s každou modifikací kódu pro jinou aplikaci se musí i náležitě změnit jejich automatické testy. Časová náročnost těchto změn může mít tedy i negativní vliv na náklady spojené s vývojem, hledáním a opravou chyb a servisem.

To, že píšeme/modifikujeme vlastní test, i když už je mnoho kódu napsáno, má své opodstatnění pro další vývoj. Ze zadaného vytvořeného kódu se bude vycházet i v budoucnu při tvorbě obdobných aplikací, a proto nikdy nenastane situace, že se bude psát vše od začátku. Pokud však firma chce začít testy používat, musí někdy začít. Proto se do této jakoby šablony, ze které budou následné webové aplikace čerpat, vloží testy, které se při dalším vývoji mírně upraví, stejně jako se bude mírně upravovat i vlastní kód aplikace.

Testy se zaměřují schválně na kód, který ovlivňuje citlivá data v databázi (tedy SQL dotazy) a tvoří tak nejdůležitější část systému. Kontrola probíhá zvláštní metodou, která vytvoří takový SQL dotaz, který by po spuštění neměl vrátit žádné záznamy – vytváříme vlastně dotaz, který je opakem původního. Pokud nějaký vrátí, je to špatně, protože se jedná o záznam, který neměl být změněn a přesto změněn byl. Jiná kontrola ohledně změn v databázi není známa. Automatika testování je však zachována.

## 3.4. Nástroje pro automatické testy

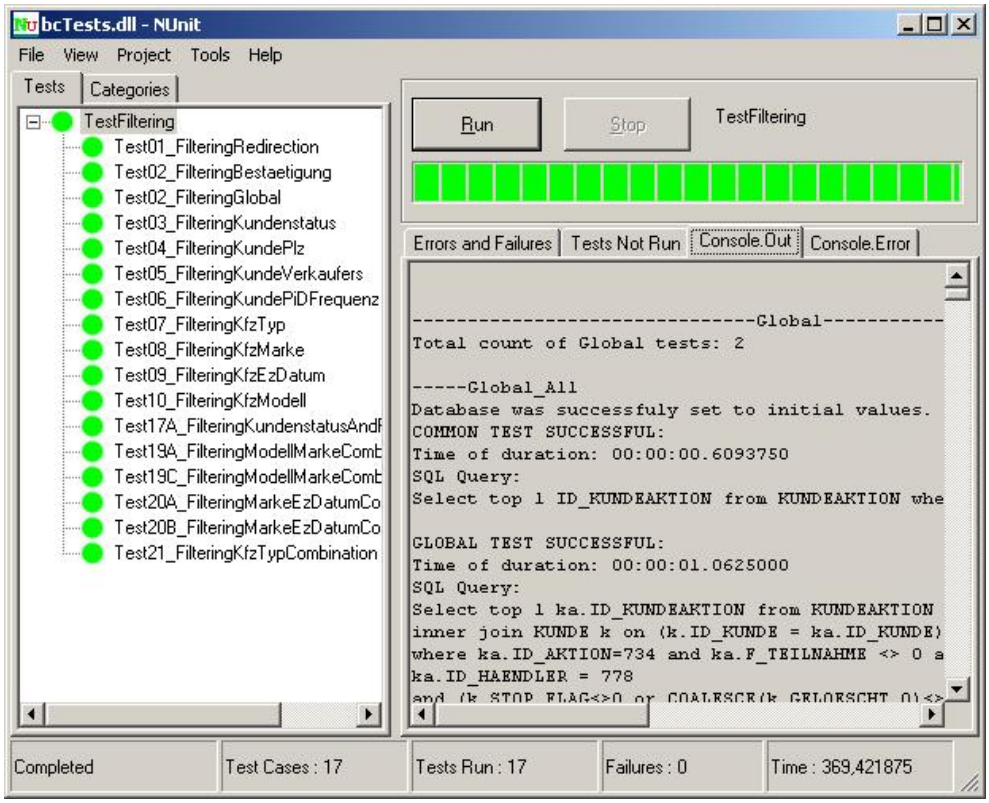
Nástrojů pro automatické testy existuje celá řada, avšak pro různé technologie existuje jiný. Například pro testování jednotek v jazyku Java

byl vytvořen JUnit, pro testování kódu ve Visual Basicu VBUnit atd. Pro testování javascriptu se bude používat systém JsUnit.

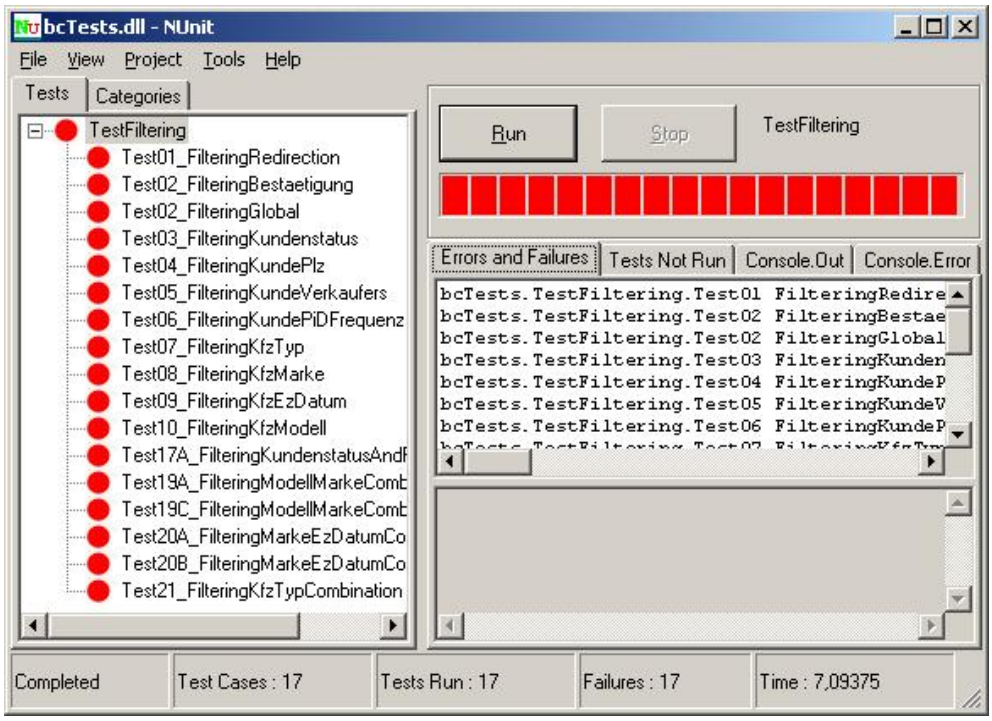
Technologii ASP.NET vyhovují následující dva nástroje:

- Microsoft Visual Studio 2005 Team System se zabudovanými nástroji pro podporu testování softwaru – tato varianta nepřichází v úvahu, protože firma Baader Computer má již zakoupenou jinou verzi Visual Studia a do dalších verzí již nehodlá investovat.
- NUnit s rozšířením NUnitAsp splňuje podobné požadavky, jako produkt od Microsoftu a přitom je zcela zdarma.

Na obrázcích 3 a 4 je k vidění grafické uživatelské rozhraní testovacího nástroje NUnit, který lze rozšířit o objekty třídy NUnitAsp a poskytuje tak možnosti k testování jak samotných tříd (dll souborů), tak i webových stránek. Na prvním obrázku je znázorněno, jak to vypadá, když test proběhne úspěšně a na druhém neúspěšný test.




**Obrázek 3: Rozhraní programu NUnit (úspěšný test – pozitivní zelená barva)**




**Obrázek 4: Rozhraní programu NUnit (neúspěšný test – negativní červená barva)**

# 4. Postup řešení

## 4.1. Samotný postup

Jak jsem již zmínil výše, jako testovací software jsem vybral software NUnit s rozšířením NUnitAsp.

Vesměs všechny frameworky sloužící pro automatické testy, obsahují obdobnou sadu nástrojů a používají se v nich podobné principy.

Kód spouštěný před každým testem lze napsat do **inicializační funkce**, která je spuštěna vždy před každým jednotlivým testem (obvykle nazvaná SetUp nebo podobně). Naproti tomu většinou existuje **destrukční funkce**, která je vykonána vždy až po každém testu (obvykle nazvaná TearDown).

Pro zjištění chyb se používají tzv. **assertions**, jejichž účelem je potvrdit či vyvrátit naše tvrzení či předpoklad a zjistit samotný výsledek testu. Obecně se dá říci, že všude existují tyto základní druhy assertions funkcí:

- AssertTrue(boolean) – pokud je vyhodnocení parametru true, test je úspěšný
- AssertFalse(boolean) – pokud je vyhodnocení parametru false, test je úspěšný
- AssertEquals(object1, object2) – pokud se object1 rovná object2, test je úspěšný
- AssertNotEquals(object1, object2) – pokud se object1 nerovná object2, test je úspěšný
- AssertNull(object) – pokud je parametr null, test je úspěšný
- AssertNotNull(object) – pokud parametr není null, test je úspěšný

Tento výčet je pouze orientační a neváže se k žádnému konkrétnímu testovacímu frameworku a funkce jsou také pouze pseudosimulované, takže ve skutečnosti jsou jejich názvy jiné.

Obvykle existuje v testovacích nástrojích také možnost testy různě seskupovat a pak je hromadně spouštět, aby se to nemuselo dělat po jednom. Těmto kolekcím se říká **suites**.

## 4.2. Testování javascriptu

Pro testování funkčnosti skriptů napsaných v jazyce Javascript jsem použil sadu testovacích html stránek, které jsou dohromady známy jako testovací framework JsUnit.

Postup testování Javascriptu:

Instalace frameworku není zapotřebí, protože se vlastně jedná pouze o sadu souborů (html stránek a přidružených souborů) a funguje v obyčejném webovém prohlížeči. Framework funguje tak, že předpokládá buď hotovou webovou stránku nebo dynamickou stránku ze serveru, která se pak načítá v testovacím rozhraní. Toto rozhraní je představováno html stránkou nazvanou testRunner.htm, kterou najdeme v kořenovém adresáři JsUnit frameworku.

Samotný soubor, který má testovat naši stránku, obsahuje funkce, jejichž název musí začínat řetězcem „test“ – tímto způsobem frameworku označíme ty funkce, které má spouštět (například „testFunction“).

Pro většinu kombinací platformy a browseru funguje autodetekce testovacích funkcí, což znamená, že JsUnit skenuje testovací stránku a zjistí si, které funkce tam existují. Ale například pro následující kombinace není autodetekce podporována: Mac 0S9 a IE5, Mac OSX a IE5, KDE a Netscape 6.x/Mozilla 0.9, KDE a Konqueror. Zde je potřeba před použitím implementovat funkci nazvanou exposeTestFunctionNames(), která bude vracet pole stringů (názvy všech testovacích funkcí). Pokud budeme používat externí soubory, což budeme, tak musíme také použít tuto funkci (jen v prohlížeči Mozilla Firefox ne).

Teď už jen stačí spustit vybraný prohlížeč a otevřít stránku testRunner.htm, kde načteme cestu k testovanému html souboru a tlačítkem Run testy spustíme. Na rozdíl od ostatních frameworků, kde je obyčejně vždy vytvářena nová instance testovací třídy, jsou zde mezi jednotlivými testy uchovávány hodnoty proměnných. Z tohoto důvodu zde existuje funkce setUpPage(), která je spuštěna na každé testovací stránce pouze jednou na začátku před spuštěním jakéhokoliv testu (na konci této funkce se musí nastavit proměnná setUpPageStatus na hodnotu "complete").

Kromě základních assetrions můžeme kvůli nejednoznačnosti hodnoty null, která je v Javascriptu vyjádřena jak hodnotou null, tak hodnotou "undefined", najít obdobné funkce assertUndefined a assertNotUndefined (kdy undefined znamená, že jsme proměnné zatím nepřiřadili žádnou hodnotu a null, že jsme jí přiřadili hodnotu null) a s ohledem na javascriptovskou funkci isNaN jsou zde implementovány i assetrions nazvané assertNaN a assertNotNaN, které vrací úspěšný výsledek testu, pokud parametr není/je převoditelný na číslo.

Debugging v Javascriptu je složitý a většinou je k tomu potřeba speciální software nebo plugin nebo se používá funkce alert. JsUnit tento problém řeší tak, že používá 3 funkce (warn, inform a debug), které se

můžou použít kdekoliv v testovací funkci. Každá funkce představuje úroveň vypisování informací při spuštění testu (úroveň se dá zvolit před spuštěním testů).

Funkce tedy může vypadat například takto (všimněte si, že v ní volám funkci Uncheck, která představuje samotnou testovanou funkci, s mnoha různými parametry):

```
function testUncheck()
{
   Uncheck(null);
   var pom = 4;
   Uncheck(pom);
   assertNotNull(pom.checked);

   AssignArrays();
   for(var i=0; i<arrayOfKFZ_ART.length; i++)
      arrayOfKFZ_ART[i].checked=true;

   Uncheck(arrayOfKFZ_ART);
   for(var i=0; i<arrayOfKFZ_ART.length; i++)
      assertFalse(arrayOfKFZ_ART[i].checked);

   mainform.KFZ_FILTER_NEXT.checked = true;
   Uncheck(mainform.KFZ_FILTER_NEXT);
      assertTrue(mainform.KFZ_FILTER_NEXT.checked)
}
```

## 4.3. Problémy při testování javascriptu

Problém při testování javascriptovských funkcí nastává v případě, když obsahuje kód, který provádí přesměrování na jinou stránku (například pomocí window.location.href) nebo použití funkce submit(). V takovém případě se sice testovací funkce provedou, ale taktéž se provede ono přesměrování pryč z testovacího rozhraní testRunner.html a my tak nemáme možnost vidět výsledky testů na stránce.

Dalším problémem je případ, kdy se očekává interakce od uživatele, jako v případě dialogových oken zobrazovaných funkcí confirm() DOM objektu window. Pak je nemožné zjistit, zda bylo kliknuto na tlačítko OK nebo Cancel a další postup ve vykonávání testované funkce je tím ovlivněn, a proto naše tvrzení nemá záchytný bod.

Z těchto důvodů je třeba zabudovat princip automatického testování do testovací funkce samotné, jako jsem to udělal já. Vytvořil jsem globální booleovskou proměnnou, která říká, zda se jedná o test nebo o skutečnou práci a pokud jde o test, zohlední se to v podmiňovacím pří-

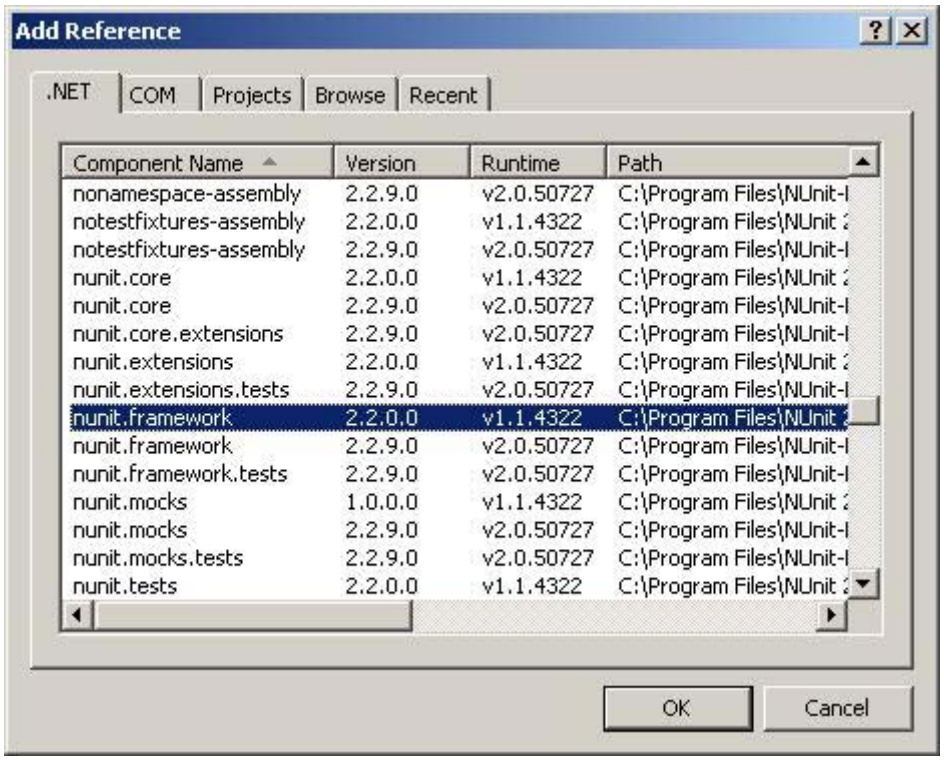
kazu a přesměrování nebo submit se neprovede a uživatelská interakce se nasimuluje inicializací nějaké pomocné proměnné.

## 4.4. Testování aspx stránek

Princip testování aspx stránek spočívá v tom, že pomocí objektů z NUnitAsp navštívíme danou stránku, tím získáme její zdrojový HTML kód a ten je potom převeden z tagů do objektů (tzv. je rozparsrován), které nabízí NUnitAsp. Těmto objektům se říká **testery** a vážou se vždy na konkrétní třídu z frameworku .NET. Například serverový ovládací prvek TextBox, který se do HTML kódu vyrenderuje jako tag input s atributem type="text", je takto rozeznán a tudíž je možné pro něj použít tester TextBoxTester, pomocí něhož můžeme vyčíst jeho hodnotu (atribut value) nebo i hodnoty dalších atributů. Obdobně i pro další ovládací prvky existují jejich ekvivalenty na straně testerů, například pro Button je to ButtonTester, pro DropDownList DropDownListTester nebo LabelTester pro třídu Label a podobně. Za pomoci metod těchto testerů můžeme uměle spouštět události, jejichž obslužný kód má být testován.

Při testování aspx stránek si musíme ve Visual Studiu založit nový projekt typu Class Library, který představuje skupinu tříd programovacího jazyka C#.

V části References musíme přidat odkazy na NUnit framework (soubor nunit.framework.dll jedině verze 2.2.0.0 – viz kapitola o problémech) a jeho rozšíření NUnitAsp (soubor NUnitAsp.dll) – viz obrázek č. 5:




**Obrázek 5: Přidání reference na knihovnu objektů**

Třída, kterou vytvoříme a která má sloužit jako testovací kód, musí být nějakým způsobem označena, aby NUnit věděl, že se jedná o třídu, kterou má zpracovat. To se provede přidáním meta-atributu **Test-Fixture** před název třídy. Musíme podědit z třídy WebFormTestCase, která představuje bázovou třídu pro třídy testované pomocí NUnitAsp – důležité jsou zejména tyto chráněné atributy a metody:

- HttpClient **Browser** – objekt na procházení webových stránek,
- WebForm **CurrentWebForm** – webová stránka aktuálně načtená Browserem,
- virtual void **SetUp()** – funkce spouštěná před každým testem,
- virtual void **TearDown()** – funkce spuštěná po každém testu.

Hlavička testovací třídy pak může vypadat například takto:

```
[TestFixture]
public class TestFiltering : WebFormTestPage
```

NUnit pak takovou třídu prochází a hledá metody, které mají něco testovat. Aby je mohl identifikovat, musí tyto metody splňovat několik zásad. Za prvé musí být veřejné (public), za druhé nesmí vracet žádný

jiný návratový typ než void, za třetí nesmí mít žádný parametr a za čtvrté musí být označeny meta-atributem **Test**. Hlavička takové funkce pak může vypadat takto:

```
[Test]
public void Test01_FilteringRedirection()
```

Třída WebFormTestCase je potomkem třídy **WebAssertion**, která obsahuje užitečnou metodu AssertVisibility, která zjišťuje hodnotu atributu Visibility daného serverového ovládacího prvku.

Nyní si musíme vybrat, jakým způsobem chceme aspx stránky testovat. Jsou dva způsoby. První a jednodušší způsob je zkompilovat celou webovou aplikaci pomocí příkazu Publish Web Site, ale nevýhodou je, že při jakékoliv změně v jejím zdrojovém kódu musíme celou aplikaci zkompilovat znovu, což je dosti neefektivní proces, protože trvá poměrně dlouho a časté testování pak zdržuje (a při dodržení principu TDD, že testování se má provádět tak často, jak to jde, je to opravdu hodně času). Druhým způsobem je provádění testů přímo na nepublikované aplikaci, kdy změna kódu v souborech .aspx.cs má okamžitý vliv na test, ale pokud chceme využít některé třídy ze sekce AppCode, je třeba je k ní přilinkovat jako referenci na soubor AppCode.dll (takže před tím také musíme provést příkaz Publish Web Site a při změně nějakého souboru v této sekci tuto činnost opakovat).

Oba způsoby neřeší problém inicializace "konstant" (tedy spíše statických proměnných), které se načítají z konfiguračního souboru web.config. Ten je totiž z NUnit projektu nedostupný, i když je nalinkován soubor AppCode.dll, a tyto proměnné tedy musíme získat jinak. Já si vytvořil pomocnou aspx stránku, která obsahuje ovládací prvky typu Label, které jsou inicializovány hodnotami ze souboru web.config. Já pak pomocí testerů vytáhnu jejich hodnoty a můžu je pak použít i v testech. Avšak tato pomocná stránka musí být kvůli inicializaci statických proměnných navštívena před každým testem (je tedy výhodné takový kód umístit do funkce SetUp, která se provede vždy před každým testem).

Ve funkci SetUp je také dobré inicializovat jednotlivé testery a propojit je se serverovými ovládacími prvky na aspx stránce. Toto přiřa-

zení se děje pomocí atributu ID, který mají všechny controly. Testery používáme jako globální proměnné v celé konkrétní testovací třídě.

Příklad přiřazení:

```
private TextBoxTester KUNDE_PLZvon1 = new
TextBoxTester("KUNDE_PLZvon1", CurrentWebForm);
```

Příklad testovací funkce:

```
[Test]
public void Test02_FilteringBestaetigung()
{
   bcUtilities.RedirectToPage(Browser,
strHaendlerNr,"filtering.aspx", "doaction=test");

   string sqlcmd = "Select * from HAENDLER_IN_AKTION where
ID_AKTION=" + strActionID + " and ID_HAENDLER=" + strID;

   SqlDataReader Reader = Database.DatabaseExecuteReader(sqlcmd);
   if (Reader.Read())
   {
      AssertEquals("0",Convert.ToString(Reader["BEST"]));
      Assert(Convert.ToString(Reader["BEST"]) != "1");
   }

   Reader.Close();
}
```

## 4.5. Úvaha o použití mock objektů

Přede mnou stál úkol kontrolovat správnost vytvoření SQL dotazu, jehož předpokládaným výsledkem by měl být nulový počet vrácených záznamů. Nejedná se zde tedy o kontrolu funkcionality jednotky, ale vše se odehrává v oblasti datové vrstvy.

Mock objekty se typicky nehodí na testování datové vrstvy, to znamená, že pokud potřebuji testovat, zda jsem správně složil SQL dotaz, v takovém případě mi nezbývá nic jiného, než se skutečně dotázat databáze a počkat si na reálné výsledky. S tím souvisí i nutná příprava testovacích dat. Pomocí mock objektů mohu např. zjistit, kolikrát se provedl příkaz Update, ale jednoduše už nezjistím, které záznamy byly ovlivněny.

Mock objekty se hodí na testování aplikační vrstvy. Vrstva obsahující aplikační logiku obvykle pracuje s datovou vrstvou, ze které získává data pro své operace. Za předpokladu, že máme již datovou vrstvu otestovanou a funkční, můžeme si ušetřit práci při testování aplikační vrstvy použitím mocků.

- původní přístup:

Pokud chci testovat aplikační objekt, ten se dotazuje datové vrstvy – před vlastním testem musím naplnit data v databázi tak, aby, když se aplikační objekt zeptá datového, vrátila správná data. Tím testuji vlastně najednou několik věcí (integrační test): vlastní aplikační objekt, objekty datové vrstvy, spolupráci aplikačního objektu s datovou vrstvou.

- přístup s mock objekty:

Pokud chci testovat aplikační objekt, ten se dotazuje datové vrstvy – v testu si vytvořím virtuální mock objekty datové vrstvy a nainstruuji jim jejich chování – tzn. řeknu jim, že až se jich aplikační objekt zeptá, mají vrátit konkrétní testová data. Žádné dotazy do databáze neproběhnou. Výsledkem je daleko méně práce při psaní testů a izolovaný (unit) test, při kterém testuji jen a jen aplikační objekt. Samozřejmě musím někde jinde otestovat zvlášť datovou vrstvu, abych zamezil chybám na tomto místě. Myšlenkou je, že i když budu izolovaně psát testy na více objektů (zvláště aplikační a zvláště datová vrstva), zabere mi to méně práce a bude to lépe otestované, než kdybych se pokoušel psát pouze integrační (složité) testy. V unit testech mám totiž větší možnosti jak otestovat i různé nuance daného volání, což bych při volání ob jeden objekt obtížně testoval.

Co z toho vyplývá? Mým zájmem není testovat aplikační vrstvu, nýbrž tu datovou, takže pro mé testování je použití mock objektů nevhodné a tudíž jsem tuto techniku ani nepoužil. V mém konkrétním případě je tedy lepší původní přístup.

## 4.6. Testování SQL dotazů

Jak jsem tedy postupoval: Předpokládám, že pokud je testovaný dotaz správný a já vytvořím takový dotaz, který je opakem původního, nebudou vráceny žádné záznamy. Vytvořil jsem si pomocnou funkci, která napodobuje chování assert funkce, nazval jsem ji AssertQueryHasNoRows, jejíž nejdůležitější parametr je stringový parametr sqlcmd, jenž představuje inverzní SQL dotaz. Účel ostatních parametrů je spíše infor-

mační – pomocí nich vypisuji do konzole programu NUnit zprávy o právě probíhajícím testu. Pokud výsledek dotazu sqlcmd nevrátí žádný záznam, tato funkce vrací true, v opačném případě false.

```
protected void AssertQueryHasNoRows(string name, string message,
DateTime startTime, string sqlcmd)
{
   if (sqlcmd != String.Empty)
   {
      SqlDataReader Reader =
Database.DatabaseExecuteReader(sqlcmd);

      TimeSpan t = System.DateTime.Now.Subtract(startTime);

      string successMessage = "\nTime of duration: " +
t.ToString() + "\nSQL Query:\n" + sqlcmd + "\n";

      string wholeMessage = message + successMessage;

      AssertEquals(name.ToUpper() + " TEST UNSUCCESSFUL: " +
wholeMessage, false, Reader.Read());

      Reader.Close();

      Console.WriteLine(name.ToUpper() + " TEST SUCCESSFUL: " +
successMessage);
   }
   else
      Console.WriteLine("Empty Sql query.");
}
```

Stručně popíšu strukturu databáze. Nejdůležitější tabulkou je tabulka KUNDEAKTION, která obsahuje adresy jednotlivých zákazníků. Tato tabulka je pomocí skládaného dotazu modifikována, a proto mě zajímá, kolik záznamů bylo dotazem ovlivněno. Tato změna je indikována hodnotou políčka F_TEILNAHME (datový typ bit), kde 0 znamená, že záznam vybraný není, a 1, že vybraný je. Ke každému záznamu se váže políčko ID_HAENDLER, které odkazuje na obchodníka, který z daných záznamů může vybírat, a ID_AKTION omezující výběr jen na určitou akci, pro niž bude šablona této aplikace v budoucnu využita.

Nejprve je potřeba navolit pomocí testerů hodnoty ovládacích prvků, takže vybereme (neboli programově nastavíme) kombinaci, kterou chceme testovat, tzn. zaškrtneme potřebné checkboxy, vyplníme požadovaná textová políčka nebo vybereme hodnotu v rozbalovacím menu. Při tom musíme dbát na nastavení defaultních hodnot controlů změněných v předchozím testu, o čemž se zmiňuji v poznámce o view state v následující kapitole o problémech. Změny se projeví až po submitu těchto hod-

not, což provedeme později simulací kliku na submitovací tlačítko (děje se ve funkci SubmitAndCommonTests – viz níže):

```
private ButtonTester DeleteAndAddSelection;
DeleteAndAddSelection=new ButtonTester("DeleteAndAddButton",
CurrentWebForm);
...
DeleteAndAddSelection.Click();
```

Jak jsem již naznačil, všechny testy jsou prováděny pomocí vytvořené pomocné funkce SubmitAndCommonTests, která má několik úkolů (provádí se v tomto pořadí):

- Nejprve se nachystají data v databázi, tzn. všem adresám z tabulky KUNDEAKTION je nastavena hodnota F_TEILNAHME na 0 (nejen pro testovaného obchodníka, ale i pro ostatní – později se kontroluje, zda náhodou nebyly dotazem ovlivněny i cizí záznamy). Na takto připravené databázi mohu nyní aplikovat testovací SQL dotaz a posléze jeho inverzní dotaz, který by neměl nic vrátit, aby byl test úspěšný. Tímto dotazem dosáhneme vynulování případných předchozích výběrů:

  ```
  Update KUNDEAKTION SET F_TEILNAHME=0 WHERE F_TEILNAHME<>0
  and ID_AKTION=222
  ```

- V závislosti na simulované činnosti uživatele (klik vyvolávající událost měnící databázi) se vykoná testovaný SQL dotaz zmíněný níže. V prvním příkazu UPDATE se vymažou předchozí volby a v druhém jsou hodnoty znovu nastaveny podle aktuálních voleb. Řekněme, že konkrétní SQL dotaz by mohl vypadat například takto (zkrácená zjednodušená forma – vynechány některé matoucí podmínky):

  ```
  Update KUNDEAKTION
  set F_TEILNAHME=0
  from KUNDEAKTION ka
  where
  ka.ID_AKTION=222 and ka.ID_HAENDLER=333 and
  ka.F_TEILNAHME=1

  Update KUNDEAKTION
  set F_TEILNAHME=1
  from KUNDEAKTION ka
  inner join KUNDE k on (k.ID_KUNDE=ka.ID_KUNDE)
  where
  ka.ID_AKTION=222 and
  ka.ID_HAENDLER=333 and
  k.KUNDENSTATUS in (1)
  ```

ebyly ovlivněny i záznamy patřící jinému obchodníkovi – následující dotaz nesmí vrátit žádný záznam, protože by to značilo, že byl vybrán záznam nepatřící obchodníkovi s identifikací 333:

```
Select top 1 ID_KUNDEAKTION from KUNDEAKTION where
ID_AKTION=222 and F_TEILNAHME<>0 and ID_HAENDLER<>333
```

- Funkce SubmitAndCommonTests má kontrolní parametr assertList typu AssertPairClass[], kde třída AssertPairClass zapouzdřuje dva objekty – tester a jeho hodnotu či hodnoty, které u daného testeru předpokládáme. Pokud si tyto a skutečné hodnoty neodpovídají, znamená to, že některé pravděpodobně byly načteny z view state, a test selže (zde se jedná o testování aplikační vrstvy).

- Nyní již nezbývá nic jiného, než vytvořit inverzní SQL dotaz. Ten je dynamicky skládán podle zadaných hodnot serverových controlů a odpovídající dotaz k příkladu uvedenému výše by vypadal následovně. Vidíte, že vybíráme takové záznamy, které mají F_TEILNAHME změněné na 1 a přitom nesplňují podmínku k.KUNDENSTATUS in (1). Neměl by tedy vrátit nic.

```
Select top 1 ka.ID_KUNDEAKTION
from KUNDEAKTION ka
inner join KUNDE k on (k.ID_KUNDE = ka.ID_KUNDE)
where
ka.ID_AKTION=222 and
ka.ID_HAENDLER=333 and
ka.F_TEILNAHME<>0 and
k.KUNDENSTATUS not in (1)
```

Kámen úrazu nastává právě při tvoření samotného inverzního dotazu. Vychází totiž ze stejných možností, jako původní dotaz. Zatímco testovaný dotaz byl tvořen z hodnot serverových ovládacích prvků, jeho protějšek byl generován z hodnot testerů, což je skoro jedno a totéž, alespoň co se složitosti týká.

## 4.7. Problémy při testování aspx stránek

První problém nastane již při instalaci NUnitAsp. Podporuje totiž jen NUnit verze 2.2.0.0 a vyšší verze již ne. Při načtení testu v novější

verzi pak vždy nastane chyba, kterou odstraníme tím, že nainstalujeme i verzi 2.2.0.0 a pak ji přidáme jako referenci do projektu.

Framework NUnitAsp sice obsahuje mnoho druhů testerů, ale nepokryjí kompletní sadu všech standardních serverových ovládacích prvků. Problém tak nastává, když takové prvky používáme nebo když používáme vlastní serverové ovládací prvky. V takovém případě je třeba napsat si svůj vlastní tester, který se bude na takový prvek vázat. V mém případě se tento problém také vyskytl a pro firemní ovládací prvek bcLabelControl a bcLabelControlList jsem musel naprogramovat testery, které budou umět dané tagy a atributy rozeznat a zpracovat (bcLabelControlTester a bcLabelControlListTester).

Použití přiřazení nestandardních testerů je stejné jako těch standardních:

```
private bcLabelControlTester KUNDE_FILTER_NEXT = new
bcLabelControlTester("KUNDE_FILTER_NEXT", CurrentWebForm);
```

Všechny stránky v mojí aplikaci prochází bezpečnostní kontrolou kvůli hodnotám v session, které se tam ukládají při přihlášení a pokud je výsledek negativní, je přesměrován zpět k přihlašovacímu dialogu. Není tedy možné najet na konkrétní stránku přímo, ale jen přes proklikání všemi stránkami. Na štěstí ještě existuje pomocná stránka startexternal.aspx, která v querystringu přenáší potřebné parametry, pomocí kterých se spustí autorizace a zápis do session. Tato stránka je potřeba také proto, že moje aplikace je součástí mnohem komplexnější aplikace, ve které se již uživatel jednou připojil a přes tuto stránku se do této aplikace připojí bez dalšího loginu (komplexní aplikace předá automaticky potřebné querystring parametry). Proto také při testování pomocí NUnitAsp používám pro navštívení jednotlivých stránek tuto metodu přístupu přes startexternal.aspx, kdy querystring parametry dodávám jako konstantní hodnoty dle potřeby (hodnoty identifikující vytipovaného uživatele).

Při opakovaném přístupu k jedné stránce si musíme dát pozor na tzv. **view state**. Zjednodušeně řečeno, view state udržuje v zakryptovaném hidden poli různé hodnoty a nastavení serverových ovládacích prvků i po několika requestech na stejnou stránku. To znamená, že pokud pro-

gramovým kódem, například klikem na tlačítko, změním barvu pozadí textového pole a pak kliknu na jiné submitovací tlačítko, barva textového pole zůstane i nadále změněna. Tato informace se načetla ze zmiňovaného view state představovaného objektem ViewState, což je objekt slovníkového typu identifikující své prvky klíčem, ke kterému se váže jeho hodnota. Jindy výhodný view state však může být pro sérii několika testů nebezpečný či otravný, protože ovládací prvky mohou mít na začátku každého testu hodnoty nastavené předchozím testem. Proto je nutné všechny hodnoty uvést do původního stavu.

## 4.8. Demonstrace výsledků

Během programování aplikace bylo zjištěno, že automatické testování zní jako dobrý nápad, ale hned z několika důvodů se nevyplatí jeho komplexní implementace pro celý program. Tyto důvody jsou následující:

- Některé části programu jsou natolik jednoduché, že psaní testů je celkem zbytečné a zabere často více času než psaní samotného programového kódu.
- Mnou vytvořená webová aplikace, která má sloužit jako šablona pro další vývoj, má sice stejný základ, ale v mnoha ohledech je pak pro konkrétní aplikaci změněna. Tím pádem se musí změnit i její automatické testy. Protože tyto konkrétní aplikace jsou obvykle tvořeny v časové tísni, nezbývá čas na psaní automatických testů.
- Protože webová aplikace má životnost pouhých pár týdnů a je jednoúčelová, není potřeba dlouhodobé údržby, kvůli které se automatické testy píšou. Jakmile je jednou aplikace zveřejněna, nepředpokládá se žádná její následná změna (rozšíření) funkcionality. Po splnění svého účelu během daného termínu je aplikace odstraněna ze serveru a nikdy více není používána.
- Kontrola dynamicky vytvářených SQL dotazů se dá pomocí automatických testů zrychlit, pokud jsou ovšem napsány správně.

Jejich složitost se totiž téměř rovná složitosti samotných testovaných databázových dotazů, protože i ony samy o sobě musí být dynamicky vytvářené. Možnost chyby v testu se tedy rovná možnosti chybě v testovaném kódu a psaní testů na samotné testy by postrádalo logiku.

Na druhou stranu se v aplikaci používají soubory, které jsou v několika jiných aplikacích sdílené a předpokládá se tedy jejich stejné použití a výsledky. Tyto soubory mohou být v několika aplikacích náhodou změněny a tím se pak změní jejich funkcionalita v ostatních projektech. Proto si myslím, že pro takové typy souborů se psaní automatických testů vyplatí. Jistou nevýhodou je, že soubory v sekci AppCode nejdou testovat přímo, ale musí se nejprve zkompilovat do knihovny tříd s příponou dll, čímž se opět doba vývoje pomocí TDD rapidně zvyšuje.

# 5. Závěr

Používání techniky automatického testování je dobrý způsob, jak se zabezpečit proti nedozírným následkům v dlouhodobě udržované aplikaci, kde se časem mění a vylepšuje zdrojový kód.

Pro konkrétní požadavky firmy Baader Computer jsou však časové nároky na psaní testů natolik velké, že z hlediska efektivnosti se tato činnost nevyplatí. Psaní testů pro kvantum souborů a tříd, které se používaly dříve, může být výhodné nanejvýš v případech, kdy jsou soubory sdílené, ale v případě rozsáhlé aplikace, jako je ta, kterou jsem vytvořil, je lepší raději spoléhat na to, že v těchto souborech chyba není a podstoupit toto riziko.

Rozhodně techniku automatických testů nezavrhuji a myslím si, že na příklad pro desktopové aplikace je tato metodika ideální. Komplexní užití všech technik extrémního programování by mohlo být hezké, ale podle mého názoru krajně neefektivní.

# 6. Seznam použité literatury


1. BECK, Kent. *Programování řízené testy*. 1. vyd. Praha : Grada, 2004. 203 s. ISBN 80-247-0901-5.
2. BECK, Kent. *Extrémní programování: knihovna programátora*. 1. vyd. Praha : Grada, 2002. 158 s. ISBN 80-247-0300-9.
3. LACKO, Luboslav. *SQL : Hotová řešení*. 1. vyd. Brno : Computer Press, 2003. 300 s. ISBN 8072269755.
4. PROVOST, Peter. Test-Driven Development in .NET. *The Code Project* [online]. 2003 [cit. 2007-05-03]. Dostupný z WWW: <http://www.codeproject.com/dotnet/tdd_in_dotnet.asp>.
5. *The Official Microsoft ASP.NET 2.0 Site* [online]. 2006 [cit. 2007-05-03]. Dostupný z WWW: <http://www.asp.net/>.
6. *NUnitAsp : ASP.NET Unit Testing* [online]. 2005 [cit. 2007-05-03]. Dostupný z WWW: <http://nunitasp.sourceforge.net/>.

# ÚDAJE PRO KNIHOVNICKOU DATABÁZI

| | |
|---|---|
| Název práce | Využití technik extrémního programování pro automatické testy a TDD |
| Autor práce | Radim Dubový |
| Obor | Informační technologie |
| Rok obhajoby | 2007 |
| Vedoucí práce | Ing. Lukáš Čegan |
| Anotace | Teoretická část obsahuje vysvětlení pojmů extrémní programování, automatické testy, Test-Driven Development a pojmů dalších souvisejících témat. Dále jsou popsány teorie psaní automatických testů a programové nástroje k tomuto testování vyvinuté a také stručný popis jejich funkcionality.<br><br>Praktická část obsahuje naprogramování webové aplikace pro firmu Baader Computer s implementací automatických testů pomocí nástroje NUnit nebo NUnitAsp (nebo podobného), to znamená vytvoření automatického testu na vytvořené třídy a ASP.NET stránky. |
| Klíčová slova | Test-Driven Development, TDD, automatické testy, extrémní programování |