MASARYKOVA UNIVERZITA
FAKULTA INFORMATIKY

Automatizované testování webových aplikací

DIPLOMOVÁ PRÁCE

Michal Pietrik

Brno, 2012

Prohlášení

Prohlašuji, že tato práce je mým původním autorským dílem, které jsem vypracoval samostatně. Veškeré zdroje, prameny a literaturu, které jsem pro vypracování použil nebo z nich čerpal, v práci řádně cituji s uvedením úplného odkazu na příslušný zdroj.

Vedoucí práce: Mgr. Šimon Suchomel

Poděkování

Rád bych na tomto místě poděkoval vedoucímu práce Mgr. Šimonu Suchomelovi za vedení a inspiraci při tvorbě diplomové práce. Dále bych chtěl poděkovat vývojářům ze společnosti Kentico Software za inspiraci, rady a možnost podílet se na vývoji úspěšného produktu. Největší poděkování ale patří mé rodině a všem, kdo mě podporovali v průběhu celého studia.

Shrnutí

Tato diplomová práce se zabývá testováním webových aplikací na platformě ASP .NET. Práce obsahuje popis jednotlivých druhů testování, rozdělených podle různých kritérií, a dále rozbor testovacích nástrojů, především těch, které jsou určeny pro automatizaci testování.

Praktická část práce se zabývá procesem částečné automatizace testování konkrétní zadané webové aplikace – Kentico CMS.

Klíčová slova

Testování softwaru, Testovací nástroje, WebAii framework, NUnit, Kentico CMS

1 Úvod

Testování je rozhodujícím aspektem, který určuje kvalitu vyvíjeného softwaru. Je to důležitá součást vývoje, která určuje, jestli daná aplikace splňuje veškeré uživatelské a technické požadavky. V případě testování webových aplikací význam samotného testování ještě roste. Internet je mocné médium, které zpřístupňuje výsledný softwarový produkt miliónům uživatelů. Je proto důležité neopomenout žádný z možných pohledů na danou aplikaci a při tom pečlivě zvážit místa potenciálních výskytů chyb.

Jsou dva hlavní důvody, proč ve výsledku software nemusí být otestován pořádně. Prvním jsou nedostatečné znalosti o testovacím procesu či konkrétních typech testování, případně opomenutí některých typů, anebo pokládání určitých druhů testů za nedůležité. Následující kapitola proto popisuje samotnou úlohu testování a její obtížnost a také jednotlivé typy testů, rozdělené podle různých pohledů na vyvíjený produkt nebo podle časového horizontu od napsání kódu.

Druhým potenciálním důvodem pro nedostatečné otestování je malé množství času. To je ovšem obecně velmi těžko řešitelný problém. Potenciálními řešeními jsou prodloužení doby vývoje tak, aby na testování bylo dostatek času, nábor nových zaměstnanců na pozice testerů, pokud nebude vadit jejich počáteční neznalost produktu, anebo zefektivněním testovacího procesu. Jednou z cest k naplnění posledně zmíněného je používání vhodných nástrojů, které testování ulehčí nebo jej částečně automatizují, a právě takovým nástrojům (ovšem jen pro webové aplikace na platformě ASP .NET) se v této práci věnuje kapitola 3.

Praktickou částí práce je částečné automatizování testování zadané webové aplikace, kterou je Kentico CMS. Dvě další kapitoly se proto věnují popisu testované aplikace (i z pohledu jejího životního cyklu), resp. samotnému procesu od analýzy částí, jejichž testování by bylo vhodné automatizovat, přes výběr konkrétních nástrojů až k samotné implementaci automatického testování vybraných částí zadané aplikace.

2 Testování softwaru

Testování je nezbytnou součástí životního cyklu softwaru. Jeho účelem je vyhledávání chyb, a to pokud možno co nejdříve, a zajištění jejich nápravy. Zajištěním nápravy se zde myslí dostatečně podrobné popsání chyby a další komunikace, na jejímž základě může být chyba opravena. Důvod, proč je potřeba chybu v softwaru odhalit co nejdříve, je zřejmý – pozdější nalezení chyby znamená vyšší náklady na její opravení. Nabízí se ovšem otázka, co přesně je softwarová chyba. Ron Patton ji definoval následovně [1]:

„O softwarovou chybu se jedná, je-li splněna jedna nebo více z následujících podmínek:

1) Software nedělá něco, co by podle specifikace dělat měl.

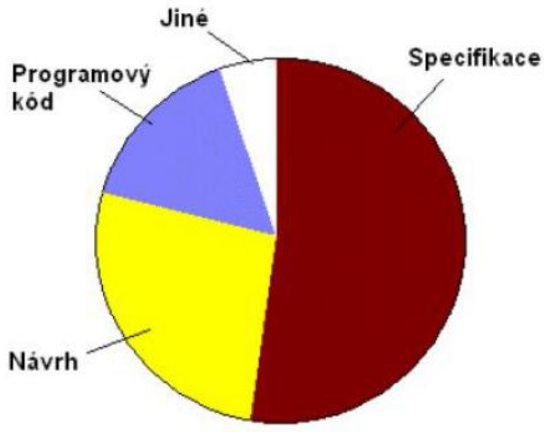
2) Software dělá něco, co by podle specifikace produktu dělat neměl.

3) Software dělá něco, o čem se specifikace nezmiňuje.

4) Software nedělá něco, o čem se produktová specifikace nezmiňuje, ale měla by se zmiňovat.

5) Software je obtížně srozumitelný, těžko se s ním pracuje, je pomalý nebo (podle názoru testera softwaru) jej koncový uživatel nebude považovat za správný.“

Z této definice plyne, že software je třeba testovat vůči jeho specifikaci – zda veškerá v ní popsaná funkcionalita odpovídá skutečnému chování programu, ale také je třeba brát v úvahu, jaká očekávání může mít koncový uživatel. Proto je před samotným testováním důležité si uvědomit, kdo a jak bude daný vyvíjený produkt nakonec používat. Důvodem, proč netestovat software jen porovnáním vůči specifikaci, je fakt, že samotná specifikace může obsahovat chyby. Na obrázku 2.1 je vidět, že více jak polovinu všech chyb má na svědomí právě specifikace.



Obrázek 1: Graf znázorňující příčiny chyb podle Pattona [1]

2.1 Obtížnost testování

Je potřeba brát v úvahu, že testování je náročný proces, zabírající nemalou část celého vývoje, a přesto si na jeho konci nemůžeme být úplně jisti, že byl daný produkt otestován dostatečně. Možných vstupů, výstupů a různých cest programem bývá většinou tak velké množství, že není možné otestovat všechny jejich kombinace v rozumném čase.

Dále je potřeba si uvědomovat problém, který popsal Dijkstra [2]:

„Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence.“

V překladu:

„Testování programu může být velmi efektivním způsobem, jak prokázat přítomnost chyb, ale je naprosto nevhodné k prokázání jejich nepřítomnosti.“

Mnoho rozhodnutí ohledně testování závisí na tom, jací lidé jsou ve vývojovém týmu a jaké mají zkušenosti. Proto se různý tým vypořádá různě s dalšími problémy, se kterými se v průběhu testování může setkat [1]:

- Specifikace nemusí být jednoznačná a není snadné odhalit, na co se v ní zapomnělo.
- Opakováním stejných testů se tyto testy mohou po čase stát neefektivními, proto je potřeba je aktualizovat a přizpůsobovat měnícím se okolnostem, tak aby pokrývaly co nejvíce nově vznikajících chyb.
- Výsledky testování mohou být ovlivněny špatnou komunikací v týmu nebo se zákazníkem.
- Není možné otestovat stejným způsobem různý software.
- Různé chyby se navenek mohou projevovat stejně, nebo naopak jedna chyba se navenek může projevovat různě.

2.2 Způsoby testování

Aby mohl být vyvíjený produkt otestován co nejlépe, je potřeba mít dopředu určeno, jaké druhy testů a kdy a do jaké míry budou aplikovány a jaké nástroje budou použity. Možných testů je velké množství, a proto se třídí do kategorií podle různých hledisek. Jak již bylo řečeno, pro každý konkrétní vyvíjený produkt se mohou hodit jiné způsoby testování. Při špatném zvolení použitých druhů testů může nastat, že se nepodaří odhalit velké množství závažných chyb nebo může dojít ke zneefektivnění celého testovacího procesu (chyby se podaří najít, ale mnohem později, než by bylo potřeba).

2.2.1 Černá a bílá skříňka

Asi nejznámější je rozdělení testů podle toho, na základě jakých znalostí o produktu k němu testeři přistupují.

Při testování typu černá skříňka (z anglického „black box testing“, známé též jako funkční testování) nemá tester k dispozici zdrojové kódy ani žádnou dokumentaci, která by je popisovala [1]. Produkt v tomto případě bývá testován podle testovacích scénářů (test suitů), které definují, jak přesně má tester postupovat při testování dané funkcionality. Typicky bývá jeden scénář věnován jednomu konkrétnímu logickému celku (např. zvláštní testovací scénář pro každý modul testované aplikace). V každém scénáři jsou jasně definované vstupy, uživatelem prováděné akce a očekávané výstupy. Pokud by měl testovací scénář být příliš dlouhý (např. pro rozsáhlejší modul, který je ovšem potřeba otestovat jako celek v rámci jednoho scénáře), bývá pak většinou rozdělen do menších logických celků (test casů), aby bylo samotné testování přehlednější. Testovací scénáře dostává tester již napsané anebo se sám zabývá jejich sestavením, například podle specifikace v době, kdy daná funkcionalita ještě není doprogramována. Výhodami testování typu černá skříňka jsou rychlost a snadnost, protože test může být prováděn bez znalosti daného programovacího jazyka. Nevýhodou pak je ale nižší kvalita kódu a také riziko, že testovací scénář nepokryje všechny potenciální chyby.

Zcela opačný přístup je u testování typu bílá skříňka (z anglického „white box testing“, známé též jako strukturální testování). Zde se totiž vyžaduje znalost vnitřních datových a programových struktur a také toho, jak je systém naimplementován. Tester musí zdrojovému kódu porozumět a analyzovat ho, proto je tento typ testování mnohem náročnější a také nákladnější. Většinou se používají různé debuggery pro analyzování kódu programu za jeho běhu. Výhodou testování typu bílá skříňka je možnost odhalení nežádoucího kódu, a tedy vyšší kvalita kódu výsledného produktu.

Jakýmsi kompromisem je testování typu šedá skříňka, u něhož tester má určité znalosti o implementaci a vnitřních pochodech systému, ale nejsou na tak vysoké úrovni, aby to mohlo být považováno za testování typu bílá skříňka. Dobrým příkladem je testování webové aplikace, kde tester nemá k dispozici celý zdrojový kód, ale jen HTML kód výsledné stránky.

2.2.2 Statické a dynamické testování

Toto rozdělení vychází z toho, zda je k provedení testu potřeba software spustit. Dynamické testování vyžaduje existenci spustitelné verze softwaru. Jeho principem bývá především posuzování výstupů na základě zadaných vstupů. Dynamické testování může být buď typu černá, nebo bílá skříňka.

Oproti tomu statické testování nevyžaduje běh softwaru, proto je možné s ním začít v době, kdy ještě není naimplementována první spustitelná verze daného programu. Statické testování může být buď typu bílá skříňka, kdy se jedná o revizi programového kódu, nebo typu černá skříňka, což je například revize specifikace [1].

2.2.3 Automatické a manuální testování

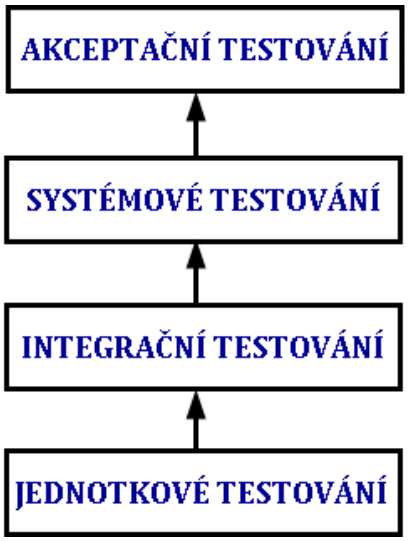
Testy lze dále rozdělit podle toho, zda jsou prováděny člověkem nebo softwarem. Pokud test vyžaduje lidské ohodnocení a úsudek, pak je vhodnější manuální testování. Pro opakované spouštění velkého množství testů nebo testu s velkým množstvím generovaných dat je vhodnější použití automatického testu. Pro automatické testování je potřeba zvolit vhodný software, který bude nejlépe splňovat naše požadavky, a rozhodnout, které často opakované testy by měly být automatizované. Je důležité se o automatizaci pokoušet tam, kde je snadno proveditelná. Příliš komplikované automatické testy mohou být velmi neefektivní, protože jejich implementací a následným udržováním můžeme nakonec strávit mnohem více času, než by bylo potřeba pro manuální otestování dané funkcionality.

2.3 Fáze testování

Testování lze rozdělit do následujících fází podle toho, v jakém časovém horizontu od napsání kódu se provádí [3]:

I. Jednotkové testování (Unit testing)
II. Integrační testování (Integration testing)
III. Systémové testování (System testing)
IV. Akceptační testování (Acceptance testing)

Jednotlivé stupně často provádí různí lidé a na každém stupni probíhá testování z trochu jiného úhlu pohledu.



Obrázek 2: Fáze testování

Za vůbec první revizi kódu lze jistě považovat kontrolu programátorem v průběhu jeho samotného psaní. Málokterý programátor by totiž odevzdal kód, který si alespoň jednou po sobě nezkontroloval nebo neověřil, jestli opravdu dělá to, co bylo zamýšleno.

2.3.1 Jednotkové testování

Jednotkové testování (z anglického „unit testing“) je proces podrobného testování co možná nejmenších částí kódu – tzv. jednotek. V ideálním případě by měl být každý testovaný případ nezávislý na ostatních. V procedurálním programování může být jednotkou funkce nebo procedura. Z pohledu objektově orientovaného programování je jednotkou většinou třída, v rámci níž se individuálně testují její metody. Při testování se snažíme testovanou část izolovat od ostatních částí programu. Za tím účelem se někdy vytvářejí pomocné objekty, které simulují předpokládaný kontext, ve kterém testovaná část probíhá. Jednotkové testování nejčastěji provádí sám programátor, protože si takto nejlépe může okamžitě ověřit kvalitu vlastní práce.

Jednotkové testy nemusí být za každou cenu napsané s použitím některého z frameworků[1] na tvorbu jednotkových testů, postačit může například jednoduchý formulář s vypisovaným výstupem a více tlačítky, které každé bude volat jinou metodu (resp. metodu s jinými parametry). Mnozí programátoři tak jistě napsali a použili své vlastní jednotkové testy, aniž by věděli, že je za jednotkové mohou považovat.

2.3.1.1. Programování řízené testy

Programování řízené testy (z anglického „test driven development“ – zkratka TDD[2]) je přístup k vývoji softwaru, jehož základem jsou jednotkové testy napsané dříve než testovaný kód [4].

Testy se poprvé spustí hned po jejich vytvoření, aby se ověřilo, že žádný z nich neprojde, protože ještě neexistuje kód, který mají testy pokrýt. Pokud by totiž některý z nich skončil úspěšně, znamenalo by to, že je špatně napsaný anebo končí úspěšně vždy, a tudíž postrádá smysl.

Programový kód je poté napsán tak, aby všechny testy prošly. Efektivita napsaného kódu se řeší až později, kdy už všechny jednotkové testy končí úspěšně. Tato poslední fáze, nazývaná refaktorování, řeší zlepšování čitelnosti a udržovatelnosti kódu, přičemž změny jsou nadále ověřovány spouštěním testů. Typickými činnostmi při refaktorování jsou rozdělení obsáhlých metod na menší nebo odstraňování duplicit, za žádnou cenu však nesmí dojít k přidávání nebo změně existující funkcionality.

---

[1] Pojem „framework“ se do češtiny často překládá jako „rámec.“ V této práci je ale kvůli přehlednosti ponecháno originální, nepočeštěné označení, protože termín „rámec“ je dále používán pro „frame“ z jazyka HTML (element sloužící k rozdělení dokumentu na samostatné části).

[2] Opakem pro test driven development je tzv. test after development (TDA), při kterém se jednotkové testy vytváří až v momentě, kdy je napsaný kód, který je předmětem testování.

Hlavními výhodou testy řízeného programování bezesporu je možnost testovat kód po jeho dopsání (resp. po dopsání funkčního celku). Programátor tak nedokončí kompletní funkcionalitu, aniž by nebyla ještě ani jednou otestovaná, přestože některé její části mohly být otestovány samostatně. To nám připomíná, že pozdější odhalení chyby zvyšuje cenu na její opravení.

Nezanedbatelnou výhodou je také fakt, že je daleko jednodušší psát samotný kód programu tak, aby prošly všechny již hotové testy, než vytvářet jednotkové testy pro (z hlediska testovatelnosti) špatně napsaný kód. Druhá varianta může být mnohdy téměř nemožná. Uvažme například metodu s mnoha parametry, jež se navzájem ovlivňují tak, že na základě různého datového typu jednoho parametru je různě pracováno i s dalšími parametry – může tak být nesmírně těžké pokrýt testem veškeré možné kombinace vstupů a výstupů.

Nevýhodami tohoto přístupu k vývoji softwaru jsou počáteční snížení produktivity, které může nastat, protože je kvůli tesům zvýšen objem kódu, který je potřeba napsat, a také skutečnost, že i testy mohou obsahovat chyby, a vést tak ke generování dalších chyb v samotném programu.

2.3.2 **Integrační testování**

Integrační testování ověřuje, že nově přidané funkcionality spolu nekolidují a všechny jednotlivé podsystémy pracují správně, i když jsou zapojeny spolu s ostatními částmi systému. Většinou se používá označení „testování vnitřní integrace“. Dále totiž existuje „testování vnější integrace“, kdy se ověřuje správné fungování testované komponenty s operačním systémem, hardwarem nebo rozhraním různých systémů [5]. Pro tuto fázi bývá typické dynamické testování černé skříňky.

2.3.3 **Systémové testování**

Systémové testy přicházejí na řadu až v pozdějších fázích vývoje. Jejich úkolem je ověření správné funkcionality daného vyvíjeného softwaru jako celku. Systém bývá většinou procházen podle testovacích scénářů, které simulují běžnou práci uživatele. Do té doby se jednotlivé týmy mohly zabývat pouze částmi systému, které spadaly pod jejich kompetenci. Obvykle procházení systému probíhá ve více kolech, přičemž nalezené chyby jsou, poté co jsou opraveny, znovu otestovány v dalším kole. V této fázi přichází na řadu také celá řada specifických druhů testů.

2.3.3.1. **Testování výkonnosti**

Jedním ze specifických druhů testů, které se provádějí ke konci vývoje, je testování výkonnosti, které je velice důležité především u webových aplikací, protože běží na vzdáleném serveru a často bývají určeny pro velké množství uživatelů. Typickým příkladem z hlediska výkonu špatně otestované aplikace může být internetový obchod, který nedokáže zpracovat všechny objednávky, trpí pomalými reakčními časy nebo zobrazuje chybové zprávy webového serveru.

Výkonnostní testování slouží k ověření, zda vyvíjená aplikace naplní nebo překročí aktuální či budoucí předpokládané požadavky uživatelů. Tyto požadavky lze rozdělit do tří základních kategorií [6]:

- Přijatelnost reakční doby
- Cíle propustnosti a počet souběžně zpracovaných uživatelů
- Požadavky na budoucí růst výkonnosti

V případě výkonnostního testování rozlišujeme jeho následující druhy:

- Zátěžový test (Load test) – zda systém funguje i pod určitou zátěží (dat nebo počtu uživatelů)
- Výkonnostní test (Performance test) – stabilita a reakční doba
- Test hraniční zátěže (Stress test) – spolehlivost při vzácně vysoké zátěži
- Test odolnosti (Soak test) – různě dlouhá doba zatížení systému
- Test selhání (Failover test) – vzpamatování se systému po výpadku, selhání hardwaru apod.
- Test objemu dat (Volume test) – zda systém funguje s různě obsáhlou databází či různě velkými datovými soubory

Metrikami typickými pro výkonnostní testování jsou počet zpracovaných požadavků za sekundu (RPS – z anglického requests per second), průměrná doba odpovědi (average response time) a počet souběžně aktivních uživatelů.

2.3.3.2. Testování použitelnosti

Testování použitelnosti (usability testing) se zaměřuje na to, jak s daným systémem nebo aplikaci pracují jeho uživatelé [4]. Z toho důvodu jej nelze automatizovat, protože lidské rozhodování a reagování na různé uživatelské prostředí nelze plnohodnotně programově podchytit.

Mnohdy se toto testování zadává skupině potenciálních uživatelů (tzv. user testing), která se s testovanou aplikací setkává vůbec poprvé (nebo skupina zkušenějších uživatelů zvyklá pracovat s dřívější verzí programu). Tým testerů plní roli uživatelů téměř od počátku vývoje, proto mu spousta skutečností může přijít samozřejmá a intuitivní, což nemusí platit u nových uživatelů.

2.3.3.3. Testování přístupnosti

Testování přístupnosti (accessibility testing) je typické převážně pro webové aplikace. Jeho cílem je ověření, jestli s danou aplikací mohou v pořádku pracovat i hendikepovaní uživatelé a uživatelé různých zobrazovacích zařízení. Je tedy potřeba dávat pozor například na:

- možnost měnit velikost zobrazovaného textu
- alternativní texty pro obsah netextového charakteru
  (kvůli uživatelům s poruchou zraku)
- validní značkování dokumentů
  (kvůli vyhledávacím robotům a uživatelům hlasových čteček)
- vhodnou volbu použitých barev
  (kvůli uživatelům s některým druhem barvosleposti)
- zobrazování na různých zařízeních a v různých prohlížečích

2.3.3.4. Testování bezpečnosti

Testování bezpečnosti je proces, který ověří, zda systém chrání svá data a funguje, jak bylo zamýšleno. Ze všech druhů testování je pravděpodobně nejnáročnější, protože u každého softwaru mohou být různé postupy k jeho zneužití. V případě webových aplikací bývá testování bezpečnosti zaměřeno na následující typy útoků [4]:

- *XSS (cross-site scripting)* – útočník dokáže do stránek, především díky neošetřeným vstupům, podstrčit svůj vlastní javascriptový kód
- *SQL injection* – napadení databázové vrstvy vlastním SQL skriptem přes neošetřený vstup na stránce
- Nahrání škodlivého souboru na server a jeho následné spuštění
- Získání informací o konkrétních objektech webové aplikace bez příslušné autentifikace
- *CSRF (cross site request forgery)* – podvržení formuláře na stránce
- Nedostatečně ošetřené chybové stavy aplikace - mohou útočníkovi sdělit informace použitelné pro další útok
- Útoky na přihlašovací část aplikace
- Nezabezpečené ukládání šifrovaných dat
- Nezabezpečená komunikace – útočník může odchytávat komunikaci, která mu není určena
- Zobrazování citlivých informací na stránkách, které nevyžadují autentifikaci

2.3.3.5. Testování dokumentace a lokalizace

Nedílnou součástí softwarového produktu je i jeho dokumentace. Ať už se jedná o podrobnou dokumentaci, čítající několik dokumentů, nebo pouze o kontextovou nápovědu, je také potřeba je náležitě otestovat – nejen z hlediska gramatických chyb a typografie, ale především z hlediska chyb faktických (zda všechny použité termíny a popsané postupy odpovídají skutečně aplikaci, anebo jestli zmíněné informace nejsou zastaralé).

Souvisejícím typem testování je obecná kontrola použitých termínů v samotném uživatelském rozhraní, resp. kontrola správné lokalizace, pokud je vyvíjený software vydáván ve více jazykových verzích. Lokalizace se totiž většinou provádí pouhým překladem konkrétních textových řetězců, vytržených z kontextu rozhraní, ve kterém jsou použity. Bez řádného otestování tak například v české lokalizaci kalendáře může být pro sedmý den v týdnu použito označení „slunce“ (po doslovném překladu zkráceného označení „Sun“ z angličtiny).

Zapomenout se nesmí ani na chybové hlášky programu. U nich je třeba kontrolovat, jestli jsou správně napsané a pochopitelné v momentě, kdy dojde k dané chybě.

2.3.4 **Akceptační testování**

Akceptační testování bývá prováděno v rámci převzetí produktu zákazníkem. Slouží k ověření, zda je daný produkt připraven k nasazení do ostrého provozu. Testováním se v tomto případě zabývá zákazník nebo jím pověřený testovací tým [4].

U mnoha softwarových produktů se můžeme setkat s akceptačním testováním prostřednictvím vydání tzv. Beta verze, kdy se sice nejedná o finální verzi, přesto je v ní ale většina plánované funkcionality (a teoreticky zbývá doladit jen drobné chyby). Beta verze bývá často k dispozici pouze vybrané skupině uživatelů (např. skupina VIP zákazníků nebo partnerů), kteří za možnost seznámit se s ještě nevydanou novou verzí softwaru poskytnou otestování reálnými scénáři v prostředí s vlastními nastaveními.

3 Nástroje pro testování

Jestliže je při testování určitého softwarového projektu potřeba vykonat obrovské množství testových případů, je velice pravděpodobné, že nezbude dostatek času pro jejich opakování. Přitom opakované provádění testů je nesmírně důležité – především v případech, kdy testy odhalily větší množství chyb v konkrétní funkcionalitě. Opravením chyb se totiž s velkou pravděpodobností mohly zanést chyby jiné. Řešením jsou vhodné nástroje pro testování softwaru, které jej v konkrétních případech automatizují nebo nám alespoň ulehčí jeho manuální provádění.

Nejdůležitějšími vlastnostmi testovacích nástrojů a automatizace testů jsou [1]:

- Rychlost – automatické testovací nástroje mohou testy provádět mnohonásobně rychleji, než by je dělal člověk.
- Efektivita – během automaticky vykonávaného testu se můžeme věnovat plánování dalších testových případů a manuálnímu testování případů, které automatizovat nejdou.
- Přesnost – na rozdíl od člověka provádí testovací nástroj práci vždy se stejnou přesností.
- Neúnavnost – automatické testy mohou běžet neustále.

V této kapitole jsou popsány nástroje, které jsou určeny k testování webových aplikací, a to především aplikací založených na platformě ASP .NET.

3.1 Frameworky pro jednotkové testování

Jednotkové testování se zaměřuje, jak již bylo zmíněno, na ověření správné funkcionality malých částí kódu – v případě objektově orientovaného programování tříd a jejich metod. Pro psaní jednotkových testů existuje napříč spektrem všech možných programovacích jazyků celá řada frameworků, jež jsou obecně označovány jako XUnit. Počáteční písmeno „X“ bývalo zpočátku většinou v názvu konkrétních frameworků zaměňováno za jiné, podle něhož bylo jasné, pro jaký programovací jazyk nebo platformu je daný framework určen[3]. S jejich přibývajícím množstvím pro stále stejné jazyky a platformy přestalo toto nepsané pravidlo platit.

Přínos těchto frameworků je pro autory testů podobný jako přínos různých vývojových prostředí (IDE[4]) pro programátory. Stejně jako například Visual Studio umožňuje uživatelům provádět s kódem všechny činnosti (psaní, kompilace, atd.) pohodlně v rámci jednoho rozhraní, tak i frameworky jednotkových testů umožňují

[3] Např. JUnit (Java), CppUnit (C++), CUnit (C), NUnit (.NET) nebo HUnit (Haskell)
[4] Zkratka pro integrated development environment

vývojářům snazší psaní samotných testů (za předpokladu alespoň základní znalosti konkrétního API[5]), spouštění hotových testů (pokud možno automatické) a prohlížení výsledků proběhnuvších testů [7].

Autoři testů mohou z knihovny daného frameworku dědit základní třídy a rozhraní a využívat třídy asercí s metodami pro ověřování správnosti testovaných dat. Framework dále většinou umožňuje buď konzolové spouštění testů (jednotlivých nebo vybrané skupiny), anebo jejich spouštění z vlastního uživatelského rozhraní, ve kterém bývají testy přehledně zobrazeny.

Následuje popis čtyř v současné době asi nejpoužívanějších frameworků na jednotkové testování na platformě ASP .NET. V případě frameworků, které tu nejsou pro tuto platformu zmíněny, se jedná často o projekty bez dalšího vývoje, anebo frameworky pro spíš okrajové případy použití, pro menší komunitu uživatelů.

3.1.1 NUnit

NUnit je určen k tvorbě jednotkových testů pro všechny jazyky na platformě Microsoft .NET, přičemž se jedná o první vzniklý framework na jednotkové testy pro tuto platformu. Napsán je pomocí programovacího jazyka C#. Na jeho vývoji se podíleli také Kent Beck[6] a Erich Gamma, autoři frameworku JUnit, který je určen pro programovací jazyk Java a v dnešní době jej lze považovat v podstatě za standard v oblasti jednotkového testování [7]. Původně se jednalo pouze o portaci již zmíněného JUnit frameworku, v dalších verzích byl však NUnit přepracován a je dál vyvíjen tak, aby využíval výhod platformy .NET, jako jsou například vlastní atributy pro třídy a metody [8], které jsou obdobou anotací z jazyka Java[7].

Základními atributy tohoto frameworku jsou:

- *TestFixture* – označuje třídu, která obsahuje testy a případně další metody spojené s testováním
- *Test* – označuje metodu, která pokrývá určitý testovací případ, tedy konkrétní jednotkový test
- *SetUp* – označuje metodu sloužící k přípravě běhového prostředí, která je automaticky spouštěna před každým jednotlivým testem
- *TearDown* – pro metody, které jsou volány po skončení jednotlivých testů
- *ExpectedException* – předepisuje, že se u daného testu očekává vyhození výjimky, jejíž typ je zadán jako parametr atributu (pokud je vyhozena výjimka jiného typu, test selže)

---

[5] Zkratka pro application programming interface, což představuje soubor tříd nějaké knihovny nebo programu, které může programátor využívat.

[6] Kent Beck je mj. tvůrce metodik Extrémní programování a Programování řízené testy

[7] Anotace v jazyce Java přidávají nějaké třídě, metodě nebo proměnné dodatečnou informaci (metadata) mimo běžný kód.

- *Ignore* – umožňuje dočasné označení třídy (resp. metody), která je označena atributem *TestFixture* (resp. *Test*) tak, aby testy nebyly prováděny
- *Values* – umožňuje pro jeden test definovat více různých hodnot pro jeho proměnné, test je pak spuštěn tolikrát, kolik je kombinací těchto hodnot

Dalším důležitým prvkem je statická třída *Assert*, která obsahuje metody pro ověření pravdivosti tvrzení, které jsou metodám předkládány jako parametry. Nejčastějším typem asercí jsou ty, které porovnávají testovanou a vzorovou proměnnou (resp. objekt). Porovnání může být pouze na základě hodnoty anebo úplné (tj. zda se jedná o dva objekty odkazující na tentýž objekt). Dále lze ověřovat například splnění zadané logické podmínky nebo datový typ testovaného objektu.

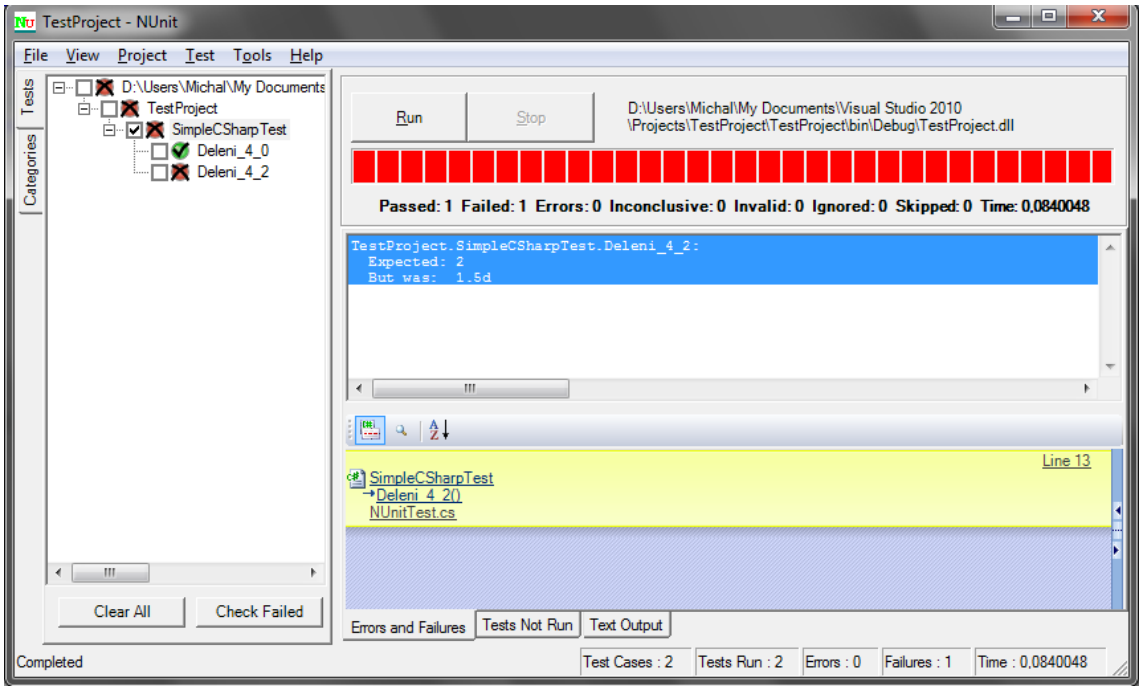
```
using System;
using NUnit.Framework;

namespace TestProject
{
    [TestFixture]
    public class SimpleCSharpTest
    {
        [Test]
        public void Deleni_4_2()
        {
            double vysledek = Pocitadlo.Vydel(4, 2);
            Assert.AreEqual(2, vysledek);
        }

        [Test, ExpectedException(typeof(DivideByZeroException))]
        public void Deleni_4_0()
        {
            double vysledek = Pocitadlo.Vydel(4, 0);
        }
    }
}
```

Příklad 1: Ukázka kódu NUnit testu

Výsledné testy je možné spouštět v aplikaci s grafickým rozhraním (nunit.exe) nebo konzolově (přes `nunit-console.exe testovaciProjekt.dll`). Výsledky testů jsou vždy uloženy do XML souboru (TestResult.xml v pracovním adresáři).



Obrázek 3: Grafické rozhraní pro spouštění NUnit testů

Přestože samotná dokumentace pro NUnit je na jeho webových stránkách poměrně nepřehledná a zčásti i nedokončená, lze tento framework rozhodně doporučit vývojářům bez větších zkušeností s jednotkovým testováním. Disponuje totiž širokou komunitou uživatelů, kteří jej používají, a tak se na internetu dá najít celá řada příkladů a postupů. Stejně tak většina odborné literatury, zaměřené na jednotkové testování na platformě ASP.NET, používá NUnit pro své ukázkové příklady.

3.1.2 MSUnit

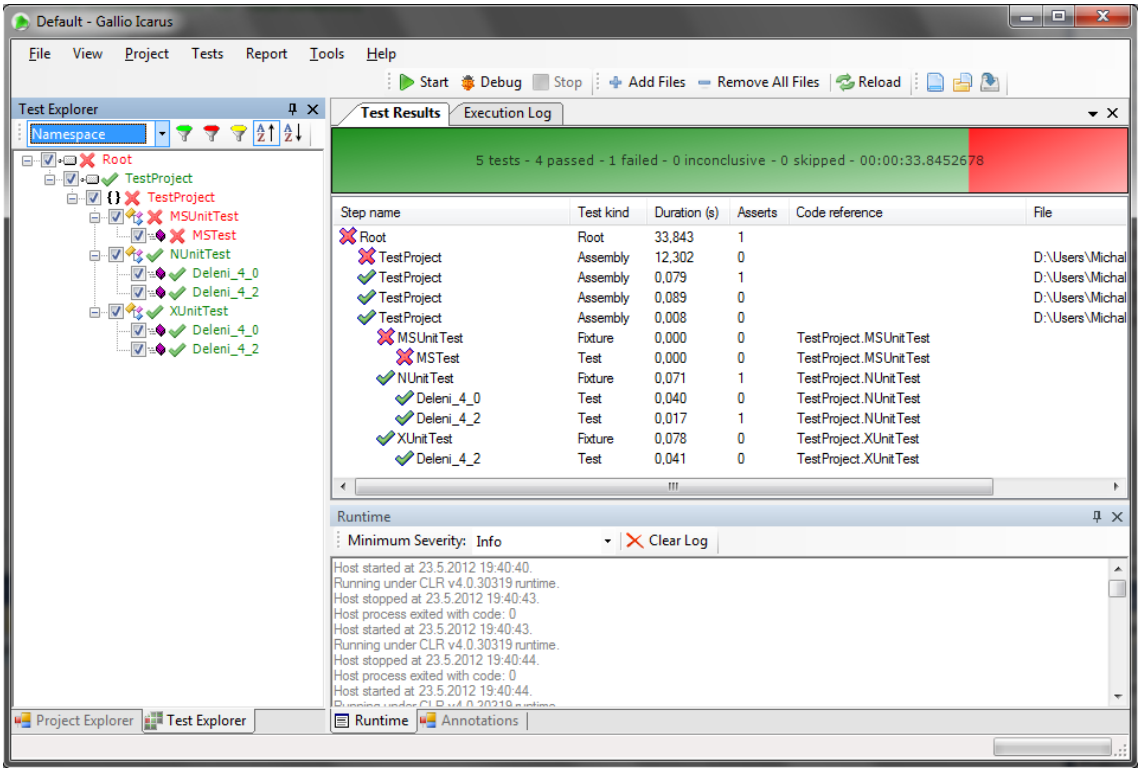
Dalším frameworkem pro tvorbu jednotkových testů je Visual Studio Unit Testing Framework od Microsoftu, označovaný také jako MSUnit nebo MSTest. Integrován je do vyšších verzí Visual Studia, ve kterém lze v něm napsané testy i spouštět. Konzolové spouštění testů je možné použitím pomocného programu *MSTest.exe*. Obsažen je ve Visual Studiu 2008 ve verzi Professional a vyšší a Visual Studiu 2010 ve verzi Premium a vyšší [7]. Pojmenování základních atributů a metod je podobné jako u NUnitu, například místo *TestFixture* se používá *TestClass*.

Nespornou výhodou tohoto frameworku je možnost využívat i k ladění testů debuggeru Visual Studia.

3.1.3 MbUnit

MbUnit, původně další samostatný framework pro jednotkové testování, je nyní součástí platformy pro automatizované testování – Gallio. Tato platforma umožňuje spouštět v rámci jednoho rozhraní testy napsané pomocí různých frameworků, kromě

uvedeného MbUnit například MSTest, NUnit, xUnit.net nebo csUnit[8]. Všechny testy jde pouštět buď konzolově pomocí nástroje *Echo*, anebo nástrojem s grafickým uživatelským rozhraním *Icarus GUI Test Runner* [9]. Rozhraní nástroje Icarus se mi oproti nástrojům s grafickým rozhraním ostatních frameworků (pomineme-li Visual Studio, které je placený produkt) jevilo přehlednější, intuitivnější a obsáhlejší z hlediska funkcionality (více možností filtrování v zobrazených testech apod.).



Obrázek 4: Icarus GUI Test Runner

Samotný MbUnit je framework, který se snaží o co největší množství funkcionality. V porovnání s ostatními obsahuje jeho třída pro aserce větší množství ověřovacích metod, například i metody pro porovnávání XML. Disponuje také více možnostmi v oblasti kategorizace testů – zatímco ostatní frameworky mají jen atribut Category, v MbUnitu existují navíc *FixtureCategory*, *Author*, *Importance* a *TestOn*. Rozšířeny jsou i možnosti, jak může proběhnuvší test skončit – kromě klasického skončení „zeleným“ úspěchem (passed) a „červeným“ neúspěchem (failed) – také „žlutým“ varováním (warning) v případě, pokud je během testu zavolána metoda *Warning* třídy

[8] csUnit je dalším z řady frameworků pro jednotkové testování na platformě .NET, protože ale jeho vývoj během posledních let nijak nepokračuje, není mu v této práci věnována větší pozornost.

Assert. Ukončení varováním je vhodné v případech, kdy test neselže úplně, ale narazí na drobnou chybu.

Užitečným atributem, který MbUnit obsahuje, je *RowTest* a k němu náležící atributy *Row*. *Rowtest* definuje test, který pokrývá různá testovaná data v rámci testovaného případu, přičemž jednotlivá data jsou definována v atributech *Row*. Takový scénář by byl řešitelný i při použití NUnit frameworku, ovšem pouze za předpokladu, že pro všechna data má test skončit stejně. V případě, kdy pro některá data je očekávána vyhozená výjimka, stačí ji v MbUnit frameworku definovat v patřičném *Row* atributu (viz následující příklad).

```
using System;
using MbUnit.Framework;
using MbUnit.Core.Framework;

namespace TestProject
{
    [TestFixture]
    public class MbUnitTest
    {
        [RowTest]
        [Row(4, 2, 2)]
        [Row(10, 2, 5)]
        [Row(4, 0, 0, ExpectedException = typeof(DivideByZeroException))]
        public void Deleni(double a, double b, double spravnyVysledek)
        {
            double vysledek = Pocitadlo.Vydel(a, b);
            Assert.AreEqual(vysledek, spravnyVysledek);
        }
    }
}
```

Příklad 2: Ukázka kódu MbUnit testu, který využívá atributu RowTest

Dalšími užitečnými atributy jsou *CsvData*, *XmlData*, *TextData* a *BinaryData*, které slouží k definici externího zdroje testovacích dat, což vývojáři ulehčí práci tím, že se nemusí zabývat implementací vlastních metod pro načítání takových dat.

3.1.4 XUnit

Zatímco u všech výše zmíněných frameworků je psaní jednotkových testů pomocí jejich API velmi podobné (většinou jde jen o menší syntaktické rozdíly), framework XUnit přichází s poměrně odlišným přístupem. Snaží se o minimalistický a elegantní přístup, obsahovat méně funkcionality (resp. ne více než konkurenční frameworky) a odlišně pojmenovávat atributy a metody [10]. Zásadní změnou je chybějící atribut pro označení třídy, která obsahuje testy. Při spuštění jsou procházeny všechny veřejné třídy a v nich jsou hledány metody s atributem *Test*, což umožňuje mít testovaný i testovací kód umístěný v rámci stejné třídy. Dle mého názoru může ale takovéto seskupování kódu vést k jeho nepřehlednosti.

Dále chybí atributy pro metody, které se mají vykonat před každým testem (resp. po něm). Podle tvůrců XUnitu je totiž jejich používání i v jiných frameworcích obecně špatné, přesto ale nabízejí alternativní cestu k dosažení stejné funkcionality v podobě neparametrizovaného konstruktoru, resp. použití rozhraní *IDisposable*. Označení samotné testovací metody se provádí pomocí atributu *Fact*.

Očekávaná výjimka se nedefinuje ve vlastním atributu, ale pomocí metody *Throws*[9] třídy *Assert*, což přináší výhodu přesného určení části kódu, která způsobí vyhození výjimky. Většina metod pro aserce je dále jinak pojmenovaná a několik očekávaných metod, známých z předešlých frameworků, chybí úplně, protože byly dle tvůrců téměř duplicitní k metodám jiným.

Díky své odlišnosti si XUnit získal mnoho příznivců, kteří jej proto upřednostňují, ale zároveň také mnoho vývojářů, zvyklých na práci například s NUnit, mívá problémy si na něj zvyknout.

```
using System;
using Xunit;

namespace TestProject
{
    public class SimpleCSharpTest
    {
        [Fact]
        public void Deleni_4_2()
        {
            double vysledek = Pocitadlo.Vydel(4, 2);
            Assert.Equal(2, vysledek);
        }

        [Fact]
        public void Deleni_4_0()
        {
            Assert.Throws<System.DivideByZeroException>(
                delegate
                {
                    Pocitadlo.Vydel(4, 0);
                });
        }
    }
}
```

Příklad 3: Ukázka kódu XUnit testu

Často bývá tento framework označován jako XUnit.net, protože obecné označení XUnit se již používá pro rodinu všech frameworků pro jednotkové testování. Jedním z jeho autorů je jeden ze spoluautorů NUnit frameworku, Jim Newkirk [7].

---

[9] Obdobná metoda pro aserci vyhozené výjimky je už například i v novějších verzích NUnit frameworku, atribut *ExpectedException* však zůstal zachován z důvodu zpětné kompatibility.

Vytvořené testy lze pouštět přes konzoli, vlastní nástroj *xunit.gui.exe* s grafickým rozhraním nebo za pomoci některého placeného doplňku (TestDriven.net, CodeRush Test Runner či Resharper) přímo ve Visual Studiu.

3.2 Nástroje pro automatické testování UI

Při používání nástrojů pro automatizaci testování uživatelského rozhraní je důležité mít na paměti, že jde o funkční testování, které simuluje aktivitu uživatele. Vždy je potřeba se zamyslet, jestli danou funkcionalitu není výhodnější přenechat na otestování skutečnému uživateli – testerovi. Mnoho lidí se domnívá, když poprvé zaslechne o možnosti automatizace testování uživatelského rozhraní, že od toho okamžiku může být úplně všechno testováno pomocí nějakého nástroje, což je velmi vzdáleno skutečnosti [4].

Nástroje pro automatizaci testování uživatelského rozhraní lze rozdělit do dvou kategorií, podle toho o jaký typ testované aplikace se jedná:

1. Webové aplikace (běží většinou na webovém serveru a k jejich rozhraní se přistupuje pomocí webového prohlížeče)
2. Desktopové aplikace (běží samostatně na stolním počítači nebo laptopu)

V této podkapitole budou dále popsány pouze ty nástroje, které jsou určené pro automatizované testování webových stránek, resp. uživatelského rozhraní webových aplikací.

3.2.1 Selenium

Selenium je open-source sada nástrojů určených pro automatické testování webových aplikací. Umožňuje psaní testů v mnoha rozšířených programovacích jazycích – konkrétně v C#, Java, Groovy, Perl, PHP, Python a Ruby. Spouštění hotových testů je možné ve webových prohlížečích na platformách Windows, Linux a Macintosh.

První z nástrojů začal být vyvíjen v roce 2004 Jasonem Hugginsem. Od té doby se zvýšil jak počet rozdílných nástrojů, obsažených v sadě Selenium, tak i oblíbenost samotného produktu – v dnešní době je, díky širokému pokrytí různých platforem, asi nejrozšířenějším řešením pro testování webových rozhraní [11].

3.2.1.1. Selenium IDE

Selenium IDE je vývojové prostředí pro tvorbu Selenium testů, které je implementováno jako doplněk pro webový prohlížeč Mozilla Firefox. Již tento fakt s sebou přináší první výhody, kterými jsou snadná a rychlá instalace a možnost vytvářet a spouštět testovací scénáře přímo ve webovém prohlížeči, tedy bez potřeby mít spuštěný navíc další nástroj. Vázanost na jeden konkrétní typ prohlížeče je ovšem také

nevýhodou, protože některé požadované testované scénáře mohou vyžadovat ověření funkcionality ve více podporovaných prohlížečích.

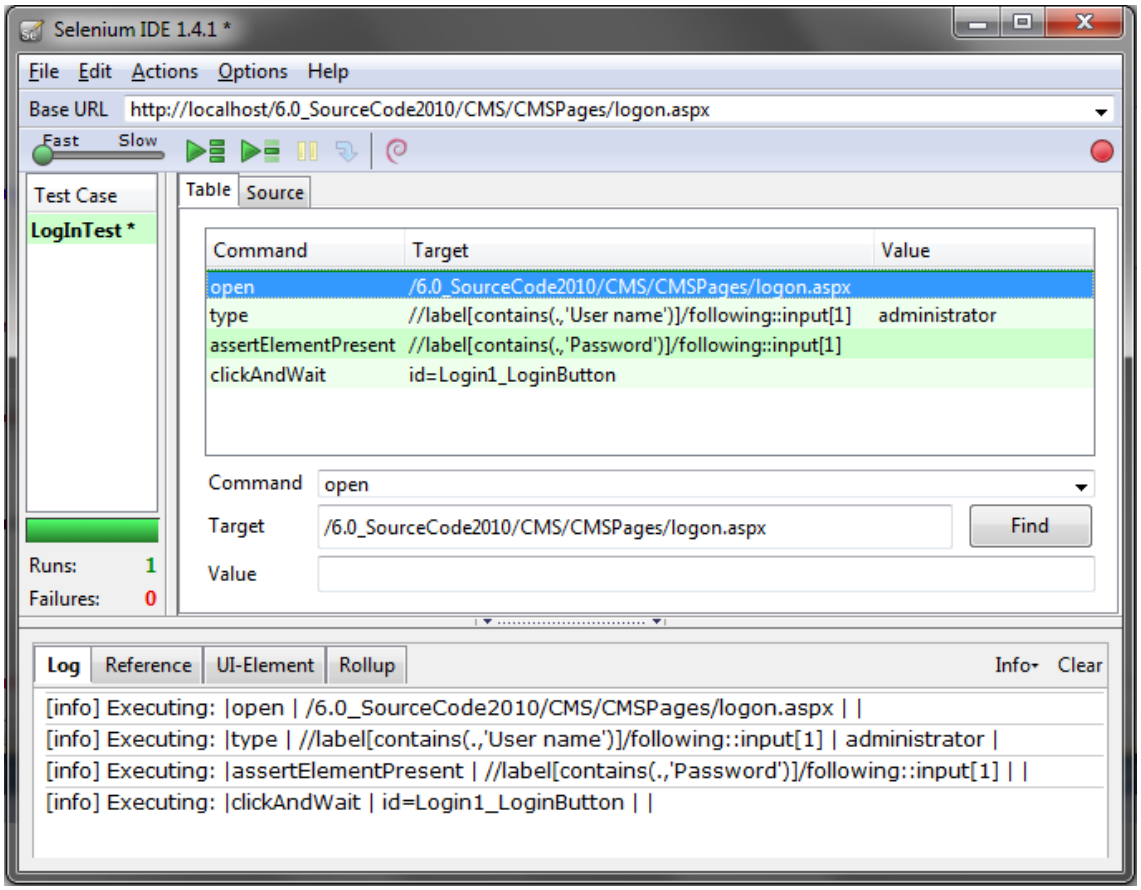
Selenium IDE má podobu jednoduchého okna, ve kterém uživatel může provádět všechny činnosti spojené s testy: jejich nahrávání, editaci, spouštění a vyhodnocování. Nahrávání testu vyžaduje vykonání testovaného scénáře uživatelem, tedy proklikání testovaného webového rozhraní v prohlížeči a případné zadávání vhodných testovacích dat. Všechny akce provedené uživatelem jsou zaznamenávány jako kroky sestávající z trojice hodnot:

- příkaz (reprezentující provedenou akci)
- cíl (identifikace elementu, nad kterým se má příkaz vykonat)
- hodnota (případně zadaná nebo zvolená příkazem)

Výsledná posloupnost kroků (trojic) je uložena do klasické HTML tabulky, což představuje specifický jazyk domény (Domain-Specific Language[10]) Selenese, který je typický pro všechny nástroje z rodiny Selenium. Konkrétní příklad tohoto tabulkového zápisu je vidět i na následujícím obrázku rozhraní Selenium IDE.

Nově nahraný test je ještě potřeba doplnit o aserce (tzn. další příkazy pro ověření správných hodnot). Zároveň je vhodné při první editaci upravit identifikaci jednotlivých elementů stránky, protože při nahrávání se pro jejich další rozpoznání zaznamenává hodnota jejich „id“ atributu, která se v ASP.NET může snadno změnit změnou zanoření daného elementu (například když vývojář přidá na stránku nějaký kontejner). Pro identifikaci lze použít hodnotu jiného atributu nebo XPath výraz.

[10] Jazyk určený k plnění specifického úkolu.



Obrázek 5: Selenium IDE

Zaznamenané testy lze dále uložit do některého z následujících jazyků: JUnit, Ruby, Python a C#. Pro žádný z těchto jazyků ovšem není možné testy v Selenium IDE spouštět. Účelem této funkce je příprava testů, které dále budou spravovány v jiném nástroji z řady Selenium.

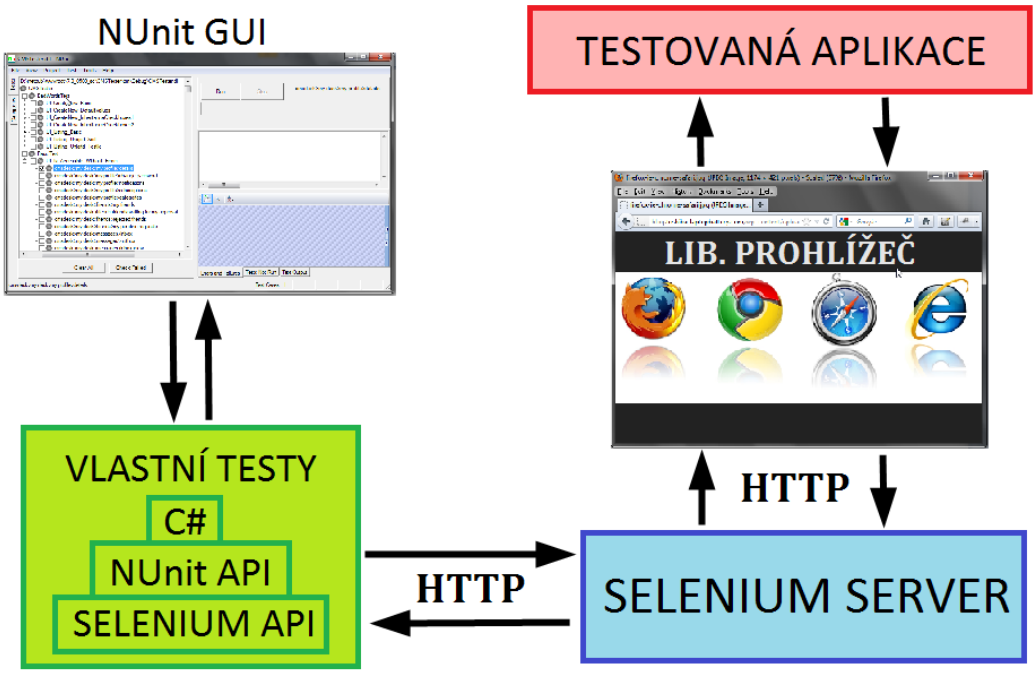
3.2.1.2. **Selenium RC**

Selenium Remote Control, někdy také označované jako Selenium 1, sestává ze dvou důležitých komponent: Selenium Server a klientské knihovny.

Selenium Server slouží ke spouštění, kontrole a zavření prohlížeče, ve kterém se mají testy vykonat. Aby bylo možné samotné testy psát v různých programovacích jazycích, probíhá ovládání serveru a veškerá komunikace prostřednictvím protokolu HTTP. Selenium Server slouží jako proxy server, tedy jako prostředník mezi prohlížečem a testovanou webovou aplikací [4]. Výhodou serveru je, že se nemusí instalovat, protože se jedná o archiv jazyka Java (soubor s příponou jar), a tudíž je přímo spustitelný v běhovém prostředí Java Runtime Environment. Pro vykonávání

konkrétní akce v prohlížeči Selenium RC předá tuto akci prohlížeči, během jeho načítání, jako javascriptovou funkci.

Pro psaní samotných testů poskytuje Selenium klientské knihovny pro všechny podporované programovací jazyky. V libovolném vývojovém prostředí (např. Visual Studiu) stačí na knihovnu zvoleného jazyka přidat referenci do projektu, který má obsahovat testy. Pomocí Selenium API by se pak dal jistě vytvořit vlastní testovací nástroj, většinou je ale výhodnější psát testy pomocí tohoto API v kombinaci s API některého frameworku pro jednotkové testy – lze tak využít výhod vybraného frameworku, jako jsou všechny již definované metody pro aserce, jednotlivé atributy a samotné rozhraní pro spouštění testů.



Obrázek 6: Diagram znázorňující práci se Selenium RC

Základním objektem při psaní testů je samotný prohlížeč, který je reprezentovaný instancí třídy dědící z rozhraní *ISelenium* (příkladem takové třídy je *DefaultSelenium*). Přes něj pak lze volat metody reprezentující jednotlivé akce v prohlížeči (např. *click* nebo *open*) nebo metody pro získávání zobrazených informací (např. *getText* nebo *isVisible*). Většině metod je předáván jako atribut textový řetězec (tzv. „locator“), podle něhož se má určit, se kterým elementem na zobrazené webové stránce má metoda pracovat. Najít konkrétní element jde pomocí jeho identifikátoru, jména, procházením DOM[11] modelu daného HTML dokumentu nebo pomocí jazyka XPath[12].

[11] DOM je zkratka pro Document Object Model – objektově orientovaný model XML nebo HTML dokumentu

```
// Otevreni testovane stranky v Internet Exploreru
DefaultSelenium browser = new DefaultSelenium("localhost", 4444,
                                    "*iexplore", "http://localhost/");
browser.Start();
browser.Open("/Test.aspx");

// Klikni na tlacitko 'myBtn'
browser.Click("name=myBtn");
```

Příklad 4: Ukázka kódu použitelného uvnitř testovací metody pro Selenium RC

3.2.1.3. Selenium Webdriver

Selenium WebDriver je součástí Selenium 2 – lze jej považovat za nástupce Selenium RC. Mezi jeho přednosti patří, že má přímo ve svém API integrovanou podporu pro otevírání webového prohlížeče, a tak v případě, že jsou testy pouštěny na stejném počítači jako prohlížeč, není potřeba, aby vůbec běžel Selenium Server. Další výraznou změnou je, že WebDriver nepředsunuje prohlížeči každou akci v podobě javascriptové funkce, ale využívá přímé podpory pro automatizaci každého jednotlivého typu prohlížeče [12].

Samotné API WebDriveru, které je oproti Selenium RC obsáhlejší, mi přišlo mnohem intuitivnější, a proto i výsledný kód čitelnější. Pro instanci prohlížeče tentokrát slouží rozhraní *IWebDriver*, samotná inicializace je jednodušší, protože stačí použít bezparametrický konstruktor. Vyhledávání jednotlivých elementů je pochopitelnější díky jednotlivým metodám pro každý typ lokalizace. Velmi užitečné je také nové rozhraní *IWebElement,* díky němuž je možné si vytvářet objekty reprezentující jednotlivé nalezené HTML elementy a s nimi dál pracovat, tedy využívat přímo metody a atributy tohoto rozhraní.

```
// Najdi dany element a klikni na nej
IWebElement myBtn = driver.FindElement(By.Name("myBtn"));
myBtn.Click();
```

Příklad 5: Testovací kód pomocí API WebDriveru (vytváření objektu pro HTML element)

3.2.1.4. Selenium Grid

Posledním nástrojem z rodiny Selenium je Selenium Grid, který slouží k paralelnímu pouštění testů na více webových prohlížečích. V prvé řadě je zapotřebí mít na více počítačích běžící Selenium RC. Spouštění testů na nich pak zajišťuje centrální server, tzv. Hub, který je zodpovědný za udržování testovacích relací a směrování požadavků mezi testem a odpovídající instancí Selenium RC. [13]. Tento přístup je velice užitečný, protože testování ve webovém prohlížeči je kvůli častému

---

[12] XPath je jazyk pro selekci XML elementů a jejich atributů

čekání na úplné načtení testované stránky mnohem pomalejší než běžné jednotkové testy. Distribucí testů mezi více testovacích instancí se celý proces výrazně urychlí.

3.2.2 Tellurium IDE

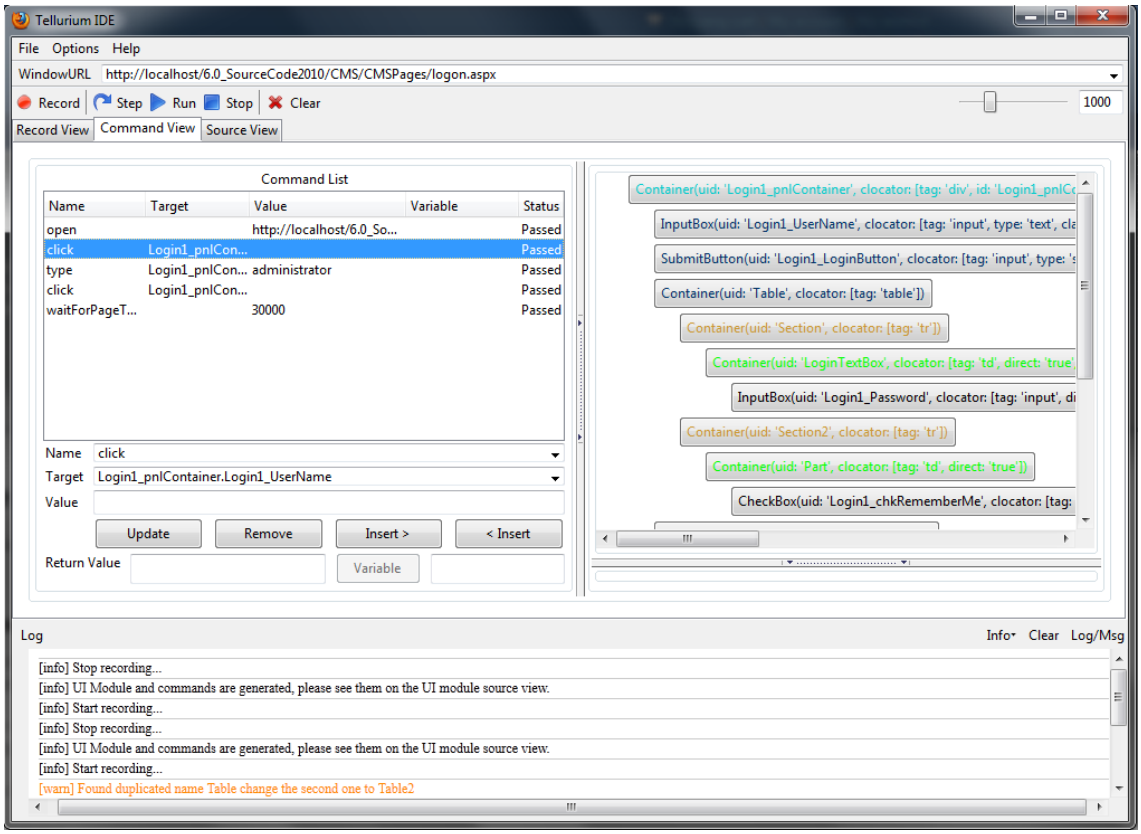
Tellurium IDE je vývojové prostředí pro testy, na první pohled velmi podobné Selenium IDE – také se jedná o doplněk webového prohlížeče Mozilla Firefox, který umožňuje nahrávání uživatelem „naklikaných" testovacích scénářů a jejich editaci a spouštění s následným vyhodnocením. Tellurium využívá jádra Selenium frameworku, ovšem přístup k určování elementů testovaného uživatelského rozhraní je odlišný. Většina frameworků pro testování webových rozhraní, jako například právě Selenium, se zaměřuje na identifikaci jednotlivých elementů (jako jsou tlačítka nebo hypertextové odkazy). Tellurium oproti tomu seskupuje UI elementy jako UI objekty do kolekce UI modulu ve skriptovacím jazyce Groovy [14]. V použitých metodách je pak odkazováno na jednotlivé elementy pomocí jim definovaného identifikátoru (`uid`). Díky separaci definice prvků uživatelského rozhraní od testovacího kódu je spuštěný test schopný najít patřičné elementy i v případě, kdy testované rozhraní je lehce upravené (např. je přidán nový panel jako nadřazený element).

```
ui.Container(uid: "Login1_pnlContainer", clocator: [tag: "div", id:
"Login1_pnlContainer"], respond: ["keyPress"])
       {
       InputBox(uid: "Login1_UserName", clocator: [tag: "input", type: "text", class:
       "LogonTextBox", id: "Login1_UserName", name: "Login1\$UserName"])

       SubmitButton(uid: "Login1_LoginButton", clocator: [tag: "input", type:
       "submit", class: "LogonButton", id: "Login1_LoginButton", value: "Log on",
       name: "Login1\$LoginButton"])
       }

open "http://localhost/6.0_SourceCode2010/CMS/CMSPages/logon.aspx"
type "Login1_pnlContainer.Login1_UserName", "administrator"
assertNotNull "Login1_pnlContainer.Login1_LoginButton"
```

Příklad 6: Ukázka testovacího skriptu pro Tellurium



Obrázek 7: Tellurium IDE

3.2.3 WebAii Framework

WebAii je framework pro automatizaci testování webových aplikací, původně vyvíjený firmou ArtOfTest, Inc., která byla později spojena se společností Telerik[13], která tento framework nechala dostupný k použití zdarma [15]. Na rozdíl například od nástrojů ze sady Selenium se ale nejedná o „open source“ řešení, takže v případě potřeby nelze upravovat kód samotného frameworku. Testy lze psát v programovacích jazycích C# nebo Visual Basic .NET, ideálně (podobně jako v případě Selenia) v kombinaci s některým frameworkem na tvorbu jednotkových testů (oficiálně jsou podporovány NUnit, MSUnit, XUnit a MbUnit). Pro tvorbu testů tedy stačí mít ve zvoleném vývojovém prostředí přidány k testovacímu projektu reference na knihovny použitých frameworků.

Základními stavebními kameny WebAii API jsou třídy *Browser*, který reprezentuje instanci webového prohlížeče, v němž je spuštěné testování, a *Manager,* který spravuje životní cyklus jednotlivých prohlížečů (instancí třídy *Browser*) a řídí komunikaci mezi

[13] Telerik je jedním z předních výrobců komponent a nástrojů pro vývoj a testování na platformě Microsoft .NET.

frameworkem a všemi otevřenými prohlížeči. Testování může probíhat na prohlížečích Internet Explorer, Firefox, Google Chrome a Safari, přičemž u každého je potřeba lehce upravit jeho základní nastavení, což je dobře popsáno v dokumentaci samotného frameworku.

Obecný přístup k tvorbě i vykonávání testů je podobný jako v případě Selenium WebDriver. V čem se tedy tyto dvě řešení liší? Dokonce i možnosti identifikace jednotlivých elementů na testované stránce jsou velmi podobné – pomocí třídy *Find*, resp. jejích konkrétních metod lze elementy vyhledávat například podle jejich jména, identifikátoru, obsahu (i pomocí regulárního výrazu) nebo pomocí XPath výrazu. Výhoda WebAii frameowrku spočívá v množství již definovaných tříd pro reprezentaci jednotlivých typů elementů značkovacího jazyka HTML. Zatímco u WebDriveru je každý nalezený element založen na *IWebElement* rozhraní, v případě WebAii můžeme použít třídu konkrétního typu elementu, pakliže se nespokojíme s instancí obecné třídy *Element*. Díky tomu lze při hledání přidat omezující podmínku v podobě třídy, jejíž nalezení je danou metodou očekáváno.

```
// Spusti prohlizec a jdi na testovanou stranku
Manager.LaunchNewBrowser(BrowserType.InternetExplorer);
ActiveBrowser.NavigateTo("http://localhost/test.aspx");

// Najdi libovolny html element se zadanym id
Element elem = ActiveBrowser.Find.ById("MyBtn");

// Najdi element <a> obsahujici zadany text
HtmlAnchor odkaz = ActiveBrowser.Find.ByContent<HtmlAnchor>("Test");
```

Příklad 7: WebAii kód

Přínosná je také třída *Settings*, která definuje, s jakými nastaveními mají být testy spouštěny. Nastavení probíhá v inicializační metodě testu, kde je možné hodnoty definovat buď přímo v kódu, nebo je načíst z konfiguračního souboru[14] testovacího projektu – tedy XML souboru, v němž jsou všechna nastavení definována. V případě, že některému nastavení není přiřazena žádná hodnota, použije se hodnota defaultní. Díky tomu je možné mít testy napsány obecně tak, že budou vždy pracovat s aktuálně požadovanou konfigurací, aniž by bylo třeba znovu kompilovat testovací projekt – prostě se jen změní hodnoty v konfiguračním souboru. Mezi nejdůležitější nastavení patří:

- *DefaultBrowser* – definuje prohlížeč, ve kterém se má test spustit (pouze pokud metoda LaunchNewBrowser nemá specifikovaný konkrétní typ prohlížeče jako parametr)

[14] Soubor s příponou *config* v kořenovém adresáři zkompilovaného projektu, který bývá většinou pojmenován stejně jako projekt, k němuž patří, nebo jen jednoduše *app.config*.

- *BaseUrl* – základní URL adresa webové aplikace (při navigaci pak jde používat relativní cestu)
- *AnnotateExecution* – určuje, zda mají být zvýrazněny elementy, na něž je v průběhu testu kliknuto
- *ExecuteCommandTimeout* – doba (v milisekundách), po kterou se má po odeslání příkazu prohlížeči čekat na jeho odpověď
- *ExecutionDelay* – určuje, jak dlouho se má čekat mezi vykonáním jednotlivých příkazů
- *LogLocation* – cesta k adresáři, do něhož se má ukládat záznam o proběhnutých testech

Mezi další zajímavé možnosti WebAii frameworku patří vyvolávání Javascriptových funkcí, odchytávání Javascriptových událostí, spravování cookies prohlížeče nebo vlastní zpracování nestandartních dialogových oken. Celkově mi API WebAii frameworku přijde velmi obsáhlé a přesto přehledné.

3.2.3.1. **Telerik Test Studio**

Společnost Telerik využívá WebAii framework i pro svůj komerční testovací nástroj, Telerik Test Studio, z čehož plyne podpora stejných webových prohlížečů a frameworků na jednotkové testy. Nástroj disponuje přehledným grafickým rozhraním pro vytváření testů, jejich následné upravování a spouštění a prohlížení výsledků. Kromě testů na webové aplikace podporuje Test Studio také automatické testy WPF[15] aplikací a zátěžové testy, u nichž simuluje souběžné přístupy definovaného počtu různých uživatelů na testovanou webovou aplikaci. Dále může tento nástroj sloužit pro správu a ukládání manuálních testovacích scénářů.

3.2.4 **WatiN**

WatiN je open-source framework inspirovaný dalším nástrojem pro testování webových aplikací, kterým je Watir.[16] Na rozdíl od něj však není určen pro psaní testovacích skriptů v programovacím jazyku Ruby, ale v jazyku C#, pomocí kterého je tento framework také napsán [7]. Podobně jako v případě WebAii frameforku i WatiN potřebuje pracovat v kombinaci s některým frameworkem na jednotkové testy, od něhož se odvíjí i nástroj, ve kterém se finální testy mohou pouštět.

Podporované webové prohlížeče, ve kterých se mohou testy přehrávat, jsou Internet Explorer a Firefox, ovšem v případě druhého jmenovaného se mi test v jeho poslední verzi (Firefox 12) nepodařilo pustit.

---

[15] zkratka pro Windows Presentation Foundation, což jsou desktopové aplikace s tzv. „bohatým uživatelským rozhraním“ vytvořeným v jazyce XAML na platformě .NET

[16] zkratka anglického „Web application testing in Ruby“

Elementy na stránce umí WatiN vyhledávat podle jejich textového obsahu nebo libovolného atributu (předdefinovanou metodu má snad pro všechny běžné atributy jazyka HTML), nikoliv však pomocí XPath výrazu, což je asi největší nedostatek tohoto nástroje. V některých případech mi nepřišlo pojmenování tříd, zastupujících jednotlivé HTML elementy, dost intuitivní. V následujících případech dokonce jedna třída odpovídá dvěma různým elementům:

- *TextField* může být `<textarea />` nebo `<input type="text" />`
- *Image* může být `<img />` nebo `<input type="image" />`
- *Button* může být `<input type="button" />` nebo `<input type="submit" />`

Další nevýhodou jsou poměrně minimální možnosti konfigurace testů – není například možné definovat, v jakém prohlížeči se test provede, protože v kódu testu se používá konstruktoru konkrétního prohlížeče.

Mnoho ze zmíněných problémů by sice šlo vyřešit vlastními úpravami kódu samotného frameworku, případně vytvořením pomocných testovacích metod, ale je to zbytečná práce navíc, zvláště pokud existují řešení, u nichž se s těmito problémy není třeba potýkat.

3.2.5 **TestComplete**

Test Complete je komerční produkt, vyvíjený společností SmartBear Software, který nabízí širokou škálu využití při testování softwaru. Umožňuje testovat uživatelské rozhraní webových i desktopových (Windows i WPF) aplikací, provádět zátěžové testy, spouštět testy jednotek nebo testy pokrytí kódu. Základní editaci nahraných testů lze provádět v jednoduchém tabulkovém zobrazení, kde jednotlivé řádky odpovídají akcím nad elementy. V případě složitějších změn je potřeba přepnout se do zobrazení zvoleného skriptu. Podporované skriptovací jazyky pro psaní testů jsou VBScript, JScript, DelphiScript, C++Script a C#Script. [16].

4 Kentico CMS

Kentico CMS je systém pro správu obsahu[17], který slouží k vytváření a správě webových stránek, komunitních stránek, e-shopů a intranetů. V současné době je Kentico využíváno více než 7 000 webovými portály v 84 zemích světa. Kentico CMS je postaveno na platformách ASP. NET a Microsoft SQL server. Vyvíjeno je společností Kentico Software, s.r.o., jež byla založena Petrem Palasem[18] v roce 2004, s centrálou v Brně a dalšími pobočkami v USA, Velké Británii a Austrálii [17]. Tato společnost, která je také průmyslovým partnerem Fakulty informatiky Masarykovy univerzity, patří k nejrychleji rostoucím technologickým firmám, působícím ve střední Evropě (v roce 2010 obsadila čtvrté místo v žebříčku sestavovaném společností Deloitte [18]).

4.1 Popis aplikace

Protože je Kentico CMS webová aplikace, přistupuje se k jejímu uživatelskému rozhraní prostřednictvím webového prohlížeče, přičemž momentálně jsou oficiálně podporovány Internet Explorer, Firefox, Chrome a Safari.

Funkcionalita systému Kentico CMS pokrývá následujících pět základních oblastí:

- Správa obsahu (Content management)
- Elektronické obchodování (E-commerce)
- Sociální software (Social networking)
- Intranet
- Online marketing

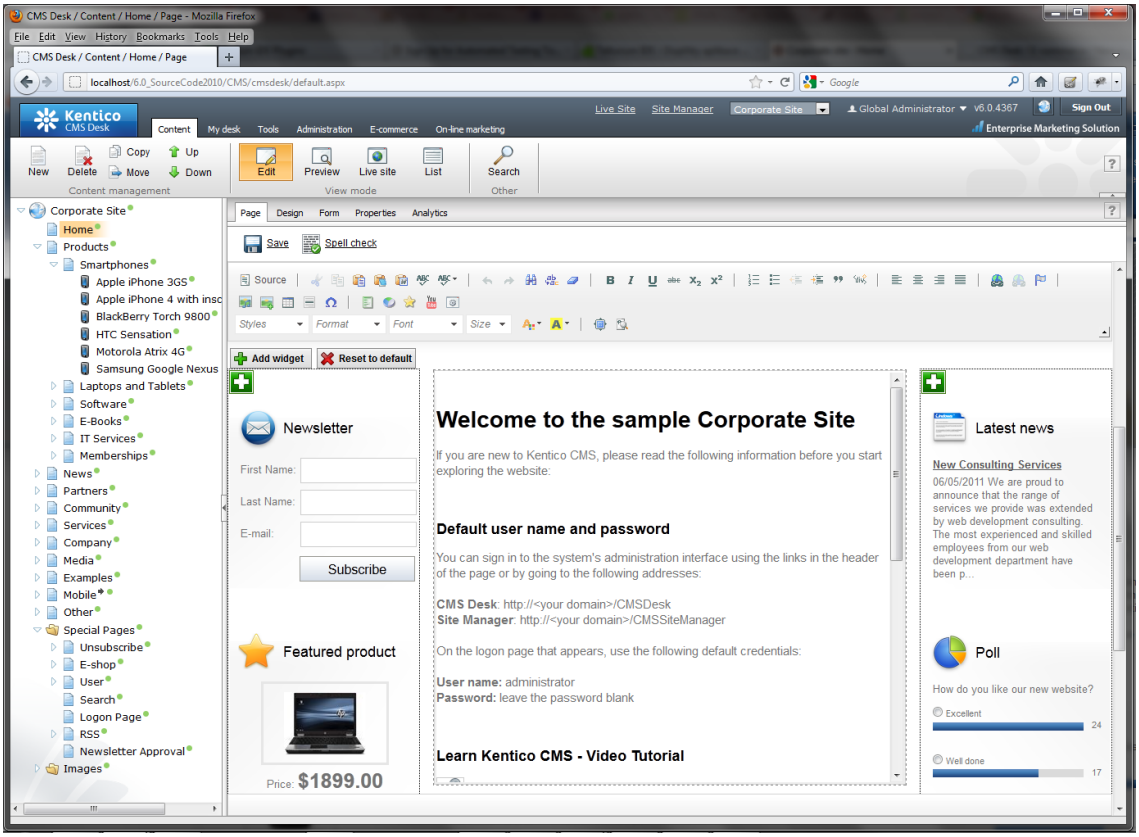
Pod každou z těchto oblastí spadá několik modulů, přičemž aktuální verze „Kentico CMS 6.0“ jich celkem obsahuje téměř 50.

Jelikož se jedná převážně o komerční produkt, tak některé moduly jsou pouze ve vyšších edicích anebo jsou v nižších edicích obsaženy s omezeními. Většinou se takové omezení týká maximálního počtu existujících objektů pod daným modulem (např. shora omezený počet uživatelů v roli editora webu). Ověřování probíhá na základě do systému zadaného licenčního klíče, který určuje, pro které domény bude přístupná daná funkcionalita. V rámci jedné instance aplikace je tedy možné spravovat více webů, z nichž každý využívá funkcí jiné edice Kentica. Nejvyšší edice, obsahující veškerou

[17] CMS je anglická zkratka pro Content Management System – do češtiny často překládáno jako „redakční systém“, přesnějším termínem je ale „systém pro správu obsahu“

[18] Petr Palas, absolvent Fakulty Informatiky Masarykovy Univerzity, vytvořil v rámci své závěrečné práce aplikaci, která vedla k první verzi systému Kentico.

funkcionalitu bez omezení, nese označení Kentico EMS[19]. Existuje i neplacená varianta (Free edition), která ale je určena pouze pro nekomerční použití [19].



Obrázek 8: Administrační rozhraní Kentico CMS

4.2 Vývoj a testování

Protože se v současnosti firma Kentico Software zabývá neustálým vývojem jednoho produktu, je maximum programátorské a testerské kapacity společnosti využito v prvé řadě na vývoj příští verze, ve druhé řadě pak na opravu chyb odhalených v poslední vydané verzi. V současné době pracuje na vývoji okolo 40 lidí (z toho 12 testerů), kteří jsou rozdělení do týmů po 4 až 7 lidech. Každý tým je zodpovědný za určité moduly.

Nová verze produktu bývá vydávána přibližně jednou až dvakrát za rok. Z toho vyplývá, že během řádově několika měsíců se musí stihnout celý proces vývoje nové verze, který se dá rozdělit do dvou fází:

I. fáze vývoje nové funkcionality
II. fáze testování systému

[19] zkratka pro Enterprise Marketing Solution

Během první fáze pracuje většina týmů podle agilní metodiky SCRUM, takže je vývoj rozčleněn do definovaných period, v nichž mají jednotliví členové týmu naplánované úkoly, které se mají v dané periodě stihnout dokončit. Úkolem v tomto případě rozumíme přidání nové funkcionality do systému. Provedení úkolu představuje naprogramování dané funkcionality a její následné otestování, přičemž úkol lze prohlásit za dokončený až v případě, kdy projde testováním bez chyb. Z pohledu testování tato fáze odpovídá zčásti integračnímu testování a zčásti testování jednotek.

Fáze testování celého systému může začít až v momentě, kdy je do systému přidána veškerá nová funkcionalita, naplánovaná pro vyvíjenou verzi. Předmětem této fáze je testování aplikace podle testovacích scénářů. Jednotlivé testovací scénáře většinou odpovídají konkrétnímu modulu, anebo nějaké logické části většího modulu. Obsahem testovacího scénáře je popis veškeré funkcionality testovaného celku a konkrétní postupy popisující, jak otestovat stanovené funkce a vlastnosti. V rámci finálního testování se správnost chování aplikace ověřuje podle všech testovacích scénářů, tedy i těch, které odpovídají modulům, v nichž se zdánlivě nic neměnilo, protože i do nich se může promítnout chyba, zanesená na první pohled úplně na jiném místě. Zvýšená pozornost je ovšem samozřejmě věnována úplně novým modulům, pro něž byl testovací scénář vytvořen v průběhu fáze vývoje nové funkcionality.

Před začátkem testování podle scénářů je jednotlivým testerům přidělena konfigurace, kterou mají při testování používat, díky čemuž se odhalí chyby existující pouze při konkrétních nastaveních, resp. kombinacích nastavení. Konfigurace sestává z kombinace následujících nastavení:

- verze operačního systému Windows, na kterém Kentico CMS běží
- verze použitého databázového serveru
- typ a verze webového prohlížeče, v němž je aplikace otevřena
- verze .NET frameworku, pro který je aplikace nainstalována
- zda je použit projekt typu Web site, Web application nebo Windows Azure
- nastavení aplikace, aby ukládala soubory na disk nebo do databáze
- případná další nastavení

Všechny testovací scénáře se procházejí ve dvou kolech, aby se odhalily případné chyby, jež se do systému zanesly opravami jiných chyb, které byly nalezeny při prvním procházení scénářů. V případě velkého množství nalezených chyb během druhého kola se vybrané scénáře otestují ještě jednou.

Po skončení všech kol testovacích scénářů už není vývojářům dovoleno provádět úpravy v kódu (ojedinělé případy pouze po schválení CTO[20]). Celý systém se ještě jednou prochází, tentokrát však pouze základní funkcionalita, a ověří se znovu všechny konfigurace. Poté je verzi možné prohlásit za dokončenou a oficiálně ji vydat.

[20] Chief Technology Officer je vedoucí zaměstnanec celého vývojářského týmu

Pokud jde o specifické druhy testů, tak testování použitelnosti a přístupnosti je prováděno v rámci schvalovacího procesu nově dokončeného modulu. Testování bezpečnosti má pro celou aplikaci na starosti jeden tým.

Jediným používaným automatickým druhem testů bylo do dokončení této práce zátěžové testování, pro které se používají nástroje obsažené ve Visual Studiu 2010 ve verzi Ultimate.

5 Vlastní automatizace testování

Protože je celkový objem funkcionality celého produktu Kentico CMS pravidelně zvyšován, roste i množství všech testovacích scénářů. Momentálně jich je celkem 157, přičemž průměrné doby jejich procházení se pohybují od 1 do 8 hodin (v případě velkého množství chyb však trvají déle). Tím se tedy průběžně zvyšuje i doba na řádné otestování na konci vývoje.

Ze stejného důvodu se zvyšuje i celkový počet částí uživatelského rozhraní aplikace. Pokud je z důvodu úpravy nebo přidané funkcionality nějakému obecnému ovládacímu prvku potřeba otestovat jeho správné chování na všech místech v aplikaci, kde je použit, může testování i nepatrné změny trvat několik dní.

Zavedení automatizace testování u takto rostoucího projektu je potřeba, aby nedocházelo k prodražení testování nebo snížení ověřené kvality výsledného produktu.

V této kapitole je popsán proces částečné automatizace v případě webové aplikace Kentico CMS.

5.1 **Požadavky**

Při ověřování kvality aplikace Kentico CMS se v současné době nepoužívají ve větší míře jednotkové testy – momentálně se takto testují vybraná kritická místa, jako je například výpočet objednávky v modulu elektronického obchodování. Před potenciálním obecným zavedením jednotkového testování na nižších úrovních je potřeba provést ještě několik úprav v jádru celého systému, k čemuž by mělo dojít během vývoje příštích verzí. Obecně si vývojáři nepíší jednotkové testy (v pravém slova smyslu) pro své třídy a metody především proto, že na to nejsou zvyklí. S vývojáři s potenciálním zájmem o testy řízené programování byla proto později řešena volba vhodného frameworku na psaní jednotkových testů.

Po diskusi s vývojáři tedy je stanovena za cíl částečná automatizace testování uživatelského rozhraní, konkrétně ovládacích prvků použitých ve velkém množství stránek aplikace. Pokrýt se tak má testování částí uživatelského rozhraní, které je zdlouhavé a monotónní. Přesně pro takové procesy je automatizace vhodná, protože člověk při dlouhodobém manuálním testování stále stejné funkcionality nemusí vydržet ověřovat vše úplně přesně po celou dobu. Navíc bývá taková práce značně neoblíbená.

Mezi obecné požadavky patří snadná udržovatelnost existujících testů, tzn. úprava testů v budoucnu musí být mnohem méně náročná než úprava libovolné funkcionality.

Důležitým požadavkem je také volba nástroje, který mohou používat libovolní pracovníci vývoje.

5.2 Analýza uživatelského rozhraní aplikace

Uživatelské rozhraní aplikace Kentico CMS je rozděleno na dvě hlavní velké části. Tou první je „Site Manager“, který je dostupný pouze hlavním administrátorům a přes který se dá dostat k rozhraním konkrétních modulů s globálními objekty (tj. objekty, které jsou dostupné pro všechny weby provozované na dané instanci Kentica). Druhou částí je „CMSDesk“, který je určen editorům konkrétní spravované webové stránky a ve kterém jsou jednotlivá rozhraní pro editaci objektů a nastavení spojených s tímto konkrétním webem. Obecně je tedy přístup do administračního rozhraní možný až po přihlášení pod uživatelem s příslušnými právy.

Aplikace využívá v administraci rámce[21] pro oddělení všech uživatelských rozhraní jednotlivých modulů od navigačních prvků. Pro každou větší část uživatelského rozhraní tak existuje konkrétní ASPX stránka. Jednotlivé rámce v uživatelském rozhraní aplikace do sebe bývají velmi často zanořeny – takže rámce obsahuje i stránka, která už je zobrazena jako rámec uvnitř jiné stránky. Při volbě konkrétního řešení automatizace tak je třeba dát pozor, jestli zvolený nástroj zvládne se zanořenými rámci pracovat. Výsledné řešení také musí fungovat na stránkách, které používají Javascript.

5.2.1 Části rozhraní vhodné pro automatické testování

Uživatelské rozhraní pro správu konkrétních objektů vždy obsahuje stránky pro jejich vytváření, editaci a zobrazení jejich seznamu, proto je právě opakující se funkcionalita v rámci těchto tří typů rozhraní vybrána pro automatizaci.

5.2.1.1. UniGrid

V rámci rozhraní systému Kentico patří k nejčastěji používaným ovládacím prvkům UniGrid, který slouží k zobrazování seznamu objektů v přehledné tabulce. Každý řádek obsahuje jeden konkrétní objekt. Položky zobrazené v jednotlivých sloupcích odpovídají atributům daného typu objektu, přičemž seznam atributů, které se mají zobrazovat v daném rozhraní, je nastaven v kódu. Data lze zobrazovat seřazená abecedně podle zvoleného sloupce. V případě většího počtu objektů se aktivuje stránkování, kdy se dá určit, kolik objektů na stránku bude zobrazeno.

První sloupec UniGridu obsahuje ikony akcí, které lze s daným objektem provádět. Téměř vždy je mezi těmito akcemi možnost editace a smazání objektu. Konkrétně přes ikonku editace je možné se dostat do editačního rozhraní daného objektu, které může sestávat z jediné stránky s formulářem (v případě jednoduchého objektu) nebo z více stránek (v případě složitějšího objektu s vazbami), ke kterým je záložkové menu.

---

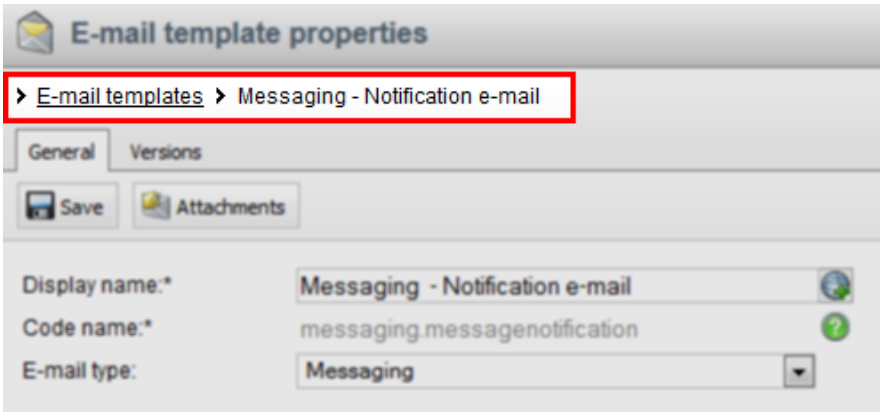
[21] Rámce jsou oblasti v HTML stránce (definované elementem `frame`), které rozdělují okno prohlížeče do několika oblastí. Každý rámec zobrazuje jiný dokument, který je ale většinou hypertextově nebo pomocí javascriptu provázán s ostatními rámci.

Obrázek 9: Popis ovládacího prvku UniGrid

V rámci aktuálně vyvíjené verze Kentica CMS je implementováno nové nastavení pro zapamatování posledního stavu UniGridu. Po opuštění a následném vrácení se na stránku s UniGridu má zůstat nastaveno stejné řazení, počet položek na stránce i aktuálně zobrazená stránka.

5.2.1.2. Breadcrumbs

Dalším rozšířeným ovládacím prvkem je tzv. Breadcrumbs, což je navigační prvek zobrazovaný většinou v rozhraní editace. Je v něm zobrazován aktuální název právě editovaného objektu a odkaz na seznam objektů v UniGridu. V minulosti se často v konkrétním editačním rozhraní neaktualizovalo již zobrazené pojmenování objektu (Display name) poté, co bylo ve formuláři změněno a uloženo. Právě tuto potenciální chybu bude vhodné pokrýt automatickým testem.



Obrázek 10: Zvýrazněný navigační prvek Breadcrumbs

5.2.1.3. Ukládací tlačítka

Další novou funkcionalitou v celém uživatelském rozhraní, která je implementována v rámci vyvíjené verze, je sjednocení potvrzovacích tlačítek „OK“ a „Save“. Ve starších verzích kvůli tomu uživatelské rozhraní vypadalo nejednotně. Protože sjednocení se neprovádí individuálně pro každou stránku, ale obecně, bude vhodný automatický test, který zkontroluje výskyt tlačítka „Save“ ve všech editačních rozhraních.

Obrázek 11: Ukázka zobrazené potvrzovací zprávy po uložení objektu

5.2.1.4. Ostatní

Kromě tří výše zmíněných ovládacích prvků by se jistě dala najít řada dalších aspektů společných pro jednotlivá uživatelská rozhraní, které by se daly pokrýt automatickým testem.

Jedním z nich je například kontrola, jestli na stránce není zobrazována nějaká obecná systémová chyba.

5.3 Výběr vhodných nástrojů

Protože hotové automatické testy mají mít možnost pouštět všichni pracovníci vývoje, nepřipadá pro ně téměř v úvahu volba některého z komerčních nástrojů, protože by nemusel být vždy dostatečný počet licencí pro všechny zaměstnance a zároveň by to celé řešení akorát prodražilo. Při výběru se tedy uvažovalo nad použitím některého z následujících: Selenium WebDriver, WebAii framweork a Watin.

Celkem jednoznačně pak bylo vybráno řešení založené na WebAii frameworku, jehož jedinou nevýhodou bylo, že se nejedná o open-source. Naopak za jeho výhody pro použití na automatizaci Kentica CMS lze považovat:

- velké množství vlastních tříd pro jednotlivé HTML elementy nalezené na stránce
- možnost použití jazyka XPath pro vyhledávání elementů
- schopnost pracovat s rámci a javascriptem
- konkrétní cílový programovací jazyk je C#
- možnost používat defaultní konfigurační soubor
- přehlednější dokumentace než dalších dvou produktů
- pravidelně vydávané nové verze
- dobrá zákaznická podpora díky tomu, že je na tomto frameworku postaven i komerční produkt

Poslední dvě zmíněné výhody jsem měl možnost si ověřit, když jsem v průběhu implementace narazil na chybu ve vyhledávání zanořených rámců, která se projevovala jen za určitých okolností. Komunikace s pracovníkem zákaznické podpory proběhla bez problému a chyba byla opravena v příští vydané verzi (asi 2 měsíce od jejího nahlášení).

Dále bylo potřeba zvolit některý framework pro jednotkové testování, který se spolu s WebAii frameworkem použije. Po konzultaci se zainteresovanými vývojáři byl vybrán framework NUnit.

Třetí, nezanedbatelnou složkou výsledného řešení je použití samotného Kentico API pro pomocné metody na urychlení specifických nastavení a práci s objekty.

```
// Initialize CMSContext
SettingsKeyProvider.WebApplicationPhysicalPath =
                    Manager.Settings.Web.WebAppPhysicalPath;
CMSContext.Init();
```

Příklad 8: Ukázka kódu, který zajistí, že testy poběží v kontextu aplikace Kentico CMS

5.4 Vytvoření automatických testů

Před samotnou implementací automatických testů uživatelského rozhraní je důležité se zamyslet nad jejich budoucí udržovatelností v případech, kdy v testovaném rozhraní dojde k různým změnám.

I v případě této práce nebylo zpočátku postupováno ideálně. Při implementaci prvních testů byla jako množina jejich společných rysů uvažována posloupnost kroků vedoucí k ověření správné funkčnosti. Každá taková posloupnost pak měla být zapouzdřena do globální pomocné metody, která by se volala při potřebě jejich vykonání.

Jako příklad uvažme metodu, která by UniGridu nastavila počet zobrazených položek na stránce a následně zkontrolovala počet skutečně zobrazených řádků tabulky. Tato metoda by se pak volala v každém jednotlivém testu. Takový test by pak měl

podobu navigace prohlížeče do testovaného rozhraní a následné zavolání pomocné metody.

Kamenem úrazu tohoto přístupu bylo množství testů (z pohledu testu jako existujícího kódu metody), které by bylo potřeba vytvořit. V případě nějaké menší aplikace by to tolik nevadilo, ale v případě Kentico CMS by se jednalo o obrovské množství testů, čímž by utrpěla zmiňovaná udržovatelnost.

Ve finálním řešení je posloupnost kroků k ověření konkrétní funkcionality umístěna do jediného testu, kterému je prostřednictvím atributu *TestCaseSource* předán seznam všech rozhraní, v nichž je tento test proveditelný. Při pohledu zvenčí je pak pro tento test zobrazena množina podpoložek odpovídající seznamu v *TestCaseSource* atributu. Díky tomu je k dispozici taky velké množství testů ovšem s minimálním množstvím skutečného testovacího kódu.

5.4.1 Automatizace navigace v uživatelském rozhraní

Kvůli zmíněnému předávání seznamu rozhraní jako parametru bylo zapotřebí vymyslet, jak takové rozhraní definovat a především jak jednoduše zajistit, aby se do něj samotný test dokázal „proklikat“. Byla proto vytvořena třída *TestableUI* určená pro definici jednoho rozhraní. Každé rozhraní je tak definováno pomocí trojice:

- cesta – řetězec reprezentující posloupnost kroků (kliknutí v prohlížeči) oddělených speciálním znakem, které vedou k požadovanému uživatelskému rozhraní
- typ uživatelského rozhraní – určuje, zda se jedná o rozhraní s výpisem objektů v UniGridu nebo rozhraní pro editaci nebo vytváření objektu
- typ objektu – řetězec převzatý z Kentico API, který definuje, s jakým druhem objektu se v daném rozhraní pracuje

Protože jsou jednotlivá rozhraní v aplikaci zobrazována uvnitř rámců, nabízelo se místo cesty použít konkrétní URL dané stránky. Tímto by ale zaprvé došlo k vytržení z kontextu celé aplikace a zadruhé by URL mohla postrádat parametry, které do ní mohou být při průchodu aplikací přidány.

Pro navigaci do konkrétního rozhraní se použije metoda *NavigateTo,* která postupuje po jednotlivých krocích definované cesty. Přesný postup je následovný:

- řetězec cesta vždy začíná buď klíčovým slovem „CMSDesk“, nebo „SiteManager“, které určuje, jaká URL se má v prohlížeči otevřít jako první
- pokud *krok* odpovídá klíčovému slovu „New“, tak se v prohlížeči klikne na tlačítko pro vytvoření nového objektu
- pokud *krok* odpovídá klíčovému slovu „Edit“, tak se v prohlížeči klikne na editaci testovaného objektu v UniGridu

- v dalších případech odpovídá *krok* položce v navigaci, na kterou se má v prohlížeči kliknout

Protože je celá aplikace tvořena rámci, pracuje test vždy pouze uvnitř dvou nejvíce zanořených rámců, protože ty odpovídají rámci s uživatelským rozhraním, které se má testovat, a rámci s odpovídajícím menu. Při navigaci do konkrétního rozhraní se proto seznam nejvíce zanořených rámců pravidelně obnovuje po každém provedeném kroku.

Seznam všech rozhraní je definován v pomocné třídě *CMSTestHelper*, z něhož jsou pro jednotlivé testy rozhraní vždy vyfiltrovány podle typu objektu a typu rozhraní.

5.4.2 Další pomocné třídy

Již zmíněná statická třída *CMSTestHelper* obsahuje kromě seznamu všech rozhraní také pomocné metody pro vyhledání nejběžnějších ovládacích prvků použitých v rámci aplikace (např. UniGrid, tlačítko pro uložení, zaškrtávací políčko atd.). Pro vyhledání konkrétního elementu je většinou použit jazyk XPath. Důležité je, že vyhledávací řetězce jsou vždy uloženy v rámci konkrétních metod nebo proměnných, aby byly testy v budoucnu snadno udržovatelné v případě nějakých závažných změn v uživatelském rozhraní systému Kentico. Další metody slouží například pro přihlášení do systému nebo pro úvodní inicializaci.

Dalšími pomocnými třídami jsou *CMSUniGrid* a *CMSUniGridRow*, které slouží k vyhledávání a ovládání jednotlivých částí UniGridu, resp. jednoho jeho definovaného řádku.

5.4.3 Konfigurační soubor

Pro změny nastavení, s jakými mají být testy spouštěny, je použit konfigurační soubor, který kromě standartních nastavení WebAii frameworku obsahuje i nastavení vlastní:

- *generatedObjectNamePrefix* – určuje, jaký prefix je použit pro předem vytvořené testovací objekty
- *windowStyle* – určuje, jak se má při testování otevírat webový prohlížeč (minimalizovaný, maximalizovaný nebo schovaný)

V konfiguračním souboru jsou dále uloženy přihlašovací údaje k databázi, kterou testovaná instance Kentico CMS používá pro ukládání dat. Pomocí Kentico API lze v rámci testů přistupovat do této databáze a konfigurovat nastavení samotné webové aplikace.

6 Závěr

Tato diplomová práce měla za úkol popsat nástroje určené k testování webových aplikací a dále navrhnout a implementovat automatické testy pro zadanou webovou aplikaci.

Byly popsány druhy testování rozdělené podle toho, jak k testované aplikaci přistupují, na jaké aspekty se zaměřují a v jaké fázi vývoje se většinou provádějí.

Představeny byly frameworky pro jednotkové testování na platformě ASP .NET a nástroje pro automatizované testování uživatelských rozhraní webových aplikací, přičemž byly popsány jejich výhody a nevýhody.

Ze zmíněných nástrojů byl vybrán WebAii framework, pomocí něhož v kombinaci s NUnit frameworkem na jednotkové testování byly vytvořeny automatické testy zaměřené na často se opakující ovládací prvky v zadané aplikaci, Kentico CMS. Byl popsán proces od analýzy testovaného uživatelského rozhraní, přes volbu vhodných nástrojů až po samotnou implementaci testů. Po celou dobu byl kladen velký důraz na udržovatelnost výsledného řešení.

Je důležité si však uvědomit, že zvolené nástroje a postupy byly vybrány pro konkrétní webovou aplikaci. V případě testování jiné aplikace může být vhodnější použití jiných nástrojů a zvolení odlišného přístupu.

Vypracované automatické testy byly použity při vývoji nové verze Kentico CMS a pomohly k nalezení několika chyb. Počítá se s jejich dalším používáním a rozšiřováním.

Teoretická část práce může sloužit jako zdroj informací vývojářům a testerům webových aplikací.

Literatura

1. **Patton, Ron.** *Testování softwaru.* Praha : Computer Press, 2002. ISBN 80-7226-636-5.

2. **Dijkstra, Edsger W.** The humble programmer. [Online] 1972 Turing Award lecture; published as EWD:EWD340pub. http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD340.PDF

3. **Wikipedia contributors.** Software testing. *Wikipedia, The Free Encyclopedia.* [Online] 10. 11. 2011 [Citace: 12. 11. 2011] http://en.wikipedia.org/w/index.php?title=Software_testing&oldid=493585825

4. **McWherter, Jeff a Hall, Ben.** *Testing ASP.NET Web Applications.* Indianapolis : Wiley Publishing, Inc., 2010. ISBN 978-0-470-49664-0.

5. **Hlava, Tomáš.** Úrovně provádění testů. *Testování softwaru.* [Online] 21. 8. 2011 [Citace: 25. 5. 2012] http://testovanisoftwaru.cz/druhy-typy-a-kategorie-testu/urovne-provadeni-testu/

6. **Microsoft ACE Team.** *Výkonnostní testování webových aplikací .Net.* Praha : Grada Publishing, 2004. ISBN 80-247-0822-1.

7. **Osherove, Roy.** *The Art of Unit Testing*. Greenwich : Manning Publications Co., 2009. ISBN 978-1-933988-27-6.

8. **NUnit.org.** *NUnit.* [Online] [Citace: 16. 5. 2012] http://nunit.com

9. **Gallio.org.** Gallio. *Gallio Wiki.* [Online] 16. 2. 2011 [Citace: 20. 5. 2012] http://www.gallio.org/wiki/doku.php?id=gallio

10. **Wilson, Brad.** Comparisons. *xUnit.net.* [Online] 21. 1. 2012 [Citace: 22. 5. 2012] http://xunit.codeplex.com/wikipage?title=Comparisons

11. **Selenium Project.** Introduction. *SeleniumHQ.* [Online] 20. 5. 2012 [Citace: 24. 5. 2012] http://seleniumhq.org/docs/01_introducing_selenium.html#test-automation-for-web-applications

12. —. Selenium WebDriver. *SeleniumHQ.* [Online] 23. 5. 2012 [Citace: 24. 5. 2012] http://seleniumhq.org/docs/03_webdriver.html

13. **OpenQA.** How it works. *Selenium Grid.* [Online] [Citace: 24. 5. 2012] http://selenium-grid.seleniumhq.org/how_it_works.html

14. **Fang, John Jian.** Tellurium User Guide: Tellurium UI Objects. [Online] 10. 11. 2010 [Citace: 19. 12. 2011] http://code.google.com/p/aost/wiki/UserGuide070UIObjects

15. **ArtOfTest, Inc.** ArtOfTest forges a strategic alliance with Telerik, Inc. *ArtOfTest, Inc. Blog*. [Online] 16. 2. 2009 [Citace: 18. 5. 2012] http://artoftestinc.blogspot.com/2009/02/artoftest-forges-strategic-alliance.html

16. **SmartBear Software.** TestComplete Automated Testing. *SmartBear.* [Online] [Citace: 25. 5. 2012] http://smartbear.com/products/qa-tools/automated-testing-tools

17. **Kentico software.** History. *Kentico.com*. [Online] [Citace: 5. 12. 2012] http://www.kentico.com/Company/History

18. **Deloitte Česká republika.** Kentico software nejrychleji rostoucí technologickou firmou v ČR. *Deloitte*. [Online] 19. 10. 2010 [Citace: 6. 2. 2011] http://www.deloitte.com/view/cs_cz/cz/tiskove-centrum/147f381a9e4cb210VgnVCM3000001c56f00aRCRD.htm

19. **Kentico Software.** Licensing Overview. *Kentico.com*. [Online] [Citace: 25. 5. 2012] http://www.kentico.com/Purchase/Licensing/Overview