

MASARYKOVA UNIVERZITA
FAKULTA INFORMATIKY



Príprava výukových materiálov o automatickom testovaní software

BAKALÁRSKA PRÁCA

Peter Dúcky

Brno, jeseň 2016

Prehlásenie

Prehlasujem, že táto bakalárska práca je mojím pôvodným autorským dielom, ktoré som vypracoval samostatne. Všetky zdroje, pramene a literatúru, ktoré som pri vypracovaní používal alebo z nich čerpal, v práci riadne citujem s uvedením úplného odkazu na príslušný zdroj.

Peter Dúcky

Vedúci práce: Mgr. David Gešvindr

Zhrnutie

Cieľom práce je vytvoriť výukový materiál v oblasti testovania. Hlavný dôraz je kladený na manuálne a automatické testovanie desktopových a webových aplikácií v prostredí .NET.

Kľúčové slová

Test, tvorba testov, automatické testovanie, testovateľný kód, manuálne testovanie, white framework, selenium, canopy.

Pod'akovanie

Chcel by som pod'akovať vedúcemu práce Mgr. Davidovi Gešvindrovi za odborné rady a spoluprácu.

Obsah

1	Úvod	1
2	Metódy vývoja softvéru	2
2.1	Vodopádový model	2
2.2	Prototypový model	3
2.3	Evolučný model	4
2.4	Inkrementálny model	5
2.5	Agilný model	6
3	Testovanie softvéru	8
3.1	Tester	8
3.2	Typy testovania	9
3.2.1	Funkčné a nefunkčné testovanie	9
3.2.2	Statické a dynamické testovanie	9
3.2.3	Testovanie čiernej, bielej a šedej skrinky	9
3.2.4	Testovanie splnením a zlyhaním	10
3.3	Testovanie v rámci vývoja	10
3.3.1	Testovanie požiadaviek	11
3.3.2	Testovanie programátorom	11
3.3.3	Jednotkové testy	11
3.3.4	Assembly testy	11
3.3.5	Smoke testy	12
3.3.6	Integračné testy	12
3.3.7	Systémové testy	12
3.3.8	Akceptačné testovanie	13
3.3.9	Testovanie po akceptovaní	13
3.3.10	Regresné testovanie	13
3.4	Hlásenie a sledovanie chýb	14
4	Manuálne testovanie	17
4.1	Testovanie so scenárom	17
4.2	Testovanie bez scenára	17
4.3	Tvorba scenára	18
4.4	Údržba scenára	19
5	Automatické testovanie	20
5.1	Testovateľný kód	21
5.1.1	Vloženie závislostí (Dependency injection)	22
5.1.2	Mock objekt	22
5.2	Štruktúra testu	23
5.3	Jednotkové testy	24

5.4	Testy užívateľského rozhrania	25
5.5	Testovanie windowsovej aplikácie	25
5.5.1	White framework	26
5.5.2	UIVerify	27
5.5.2.1	Ako získať UIVerify	29
6	Tvorba testu pre windowsovú aplikáciu	30
6.1	Príprava prostredia na tvorbu testu	30
6.2	Obsah Testu	30
6.2.1	Usingy	30
6.2.2	Globálne premenné	31
6.2.3	Inicializácia	32
6.2.4	Telo testu pre windowsovú aplikáciu	33
6.2.5	Upratanie	34
7	Tvorba testu pre webovú aplikáciu	35
7.1	Hľadanie elementov vo webovej aplikácii	35
7.1.1	Obsah testu Selenium	36
7.1.1.1	Založenie testu s použitím Selenia	36
7.1.1.2	Usingy a inicializácia	36
7.1.1.3	Telo testu	37
7.1.2	Obsah testu Canopy	38
7.1.2.1	Založenie testu s použitím Canopy	38
7.1.2.2	Inicializácia testu	38
7.1.2.3	Telo testu	39
8	Záver	41
9	Prílohy	44

1 Úvod

Žijeme vo svete plnom moderných technológií, kde jeden softvér strieda druhý. Kde jednoduché programy riadia zariadenia od hodínok cez semaforey na cestách až po medicínske prístroje. V takom svete si treba byť istý kvalitou týchto programov. Práve preto je testovanie dôležitou súčasťou vývoja softvéru. Či už ide o splnenie požiadaviek zákazníka, alebo bezpečnosť aplikácie, nie je možné testovanie vynechať.

V tejto práci som sa sústredil na oboznámenie čitateľa so spôsobmi testovania, s rôznymi prístupmi k testovaniu, ako aj s metódami na testovanie. Práca je rozdelená na niekoľko častí, kde sa venujem základnému rozdeleniu vývoja softvéru, typov testov pri jednotlivých fázach vývoja, ako aj tvorbe samotných testov. Poukazujem na značnú výhodu automatických testov, ktoré napriek vysokým počiatočným nákladom značne zjednodušujú prácu testerom. K práci sú tiež priložené videá znázorňujúce tvorbu automatických testov pre windows a web. Takisto vo videách vysvetlím, ako vytvoriť testovací scenár spolu s užitočnými radami, na čo si pri testovaní dať pozor. K príloham patria aj demoštračné programy s ukážkami jednotlivých typov automatických testov.

Cieľ práce je vytvoriť výukový materiál v oblasti testovania, ktorý bude slúžiť ako základný kameň pre ľudí, ktorí sa budú venovať testovaniu.

2 Metódy vývoja softvéru

Spôsobov, ako testovať aplikáciu alebo program, je v súčasnosti celá rada. Dá sa prejsť od jednoduchého preklikávania až po komplikované automatické testy. Aby sa tento výber čo najviac zúžil a bolo možné vybrať najefektívnejší spôsob, treba sa pozrieť na metódy vývoja softvéru, pretože každá metóda pristupuje k vývoju inak a teda sa bude líšiť, či zvolíme automatické alebo manuálne testovanie.

2.1 Vodopádový model

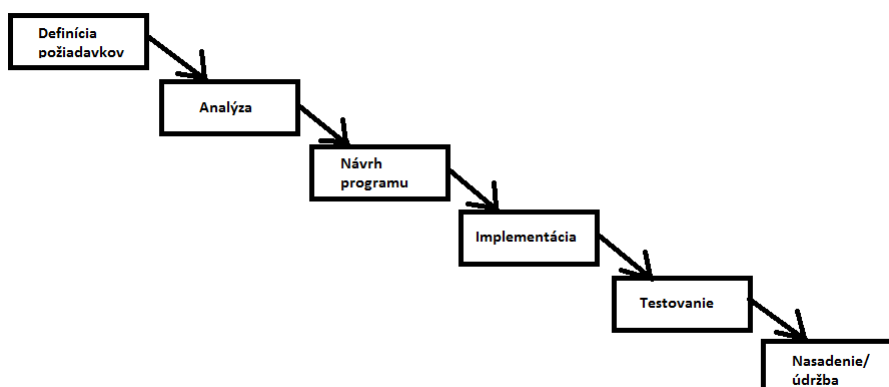
Je to prvý a najjednoduchší model vývoja softvéru. Pozostáva z vývojových fáz, kde každá fáza musí byť ukončená predtým, než sa začne ďalšia fáza. Jeho najväčšie využitie je v malých projektoch, pri ktorých sú všetky požiadavky jasne zadane a možnosť ich zmien je minimálna. Najdôležitejším faktom pri tomto modeli je to, že sa jednotlivé fázy vývoja neprekrývajú. To znamená, že pokiaľ chcem testovať, je potrebné počkať kým sa vývoj úplne ukončí[1].

Výhody

- jednoduchý a ľahký na pochopenie
- jednoznačne definované fázy vývoja
- dobrý model pre menšie projekty, pri ktorých sa požiadavky nemenia
- prehľad vo vývoji (ktoré fázy sú vykonané a čo treba spraviť)

Nevýhody

- ťažký návrat z novej fázy do staršej
- reálny fungujúci softvér je produkován až ku koncu vývoja
- nie moc dobrý model pre komplikovanejšie a zdĺhavejšie projekty
- nevhodný model pre projekty kde sa požiadavky často menia



Obr. 2.1: Vodopádový model

2.2 Prototypový model

Je to vývoj pomocou tvorby prototypov. Prototyp je neúplný funkčný model aplikácie, ktorý má obmedzenú funkčnosť. Väčšina prototypov je vyvíjaná len na ukážku, a nemá sa z neho stať finálny produkt. V niektorých prípadoch môže dôjsť k výnimkám a takýto prototyp sa stane funkčnou aplikáciou. Jeho hlavnou výhodou je možnosť demonštrácie určitej funkcionality zákazníkovi, ktorý na základe tohto prototypu môže poskytnúť odozvu, ktorá pomôže vývojárom objasniť, poprípade upraviť funkcionality výsledného produktu. Výsledkom je to, že zákazník je viac zapojený do samotného vývoja a postupne vidí ako práca napreduje[2].

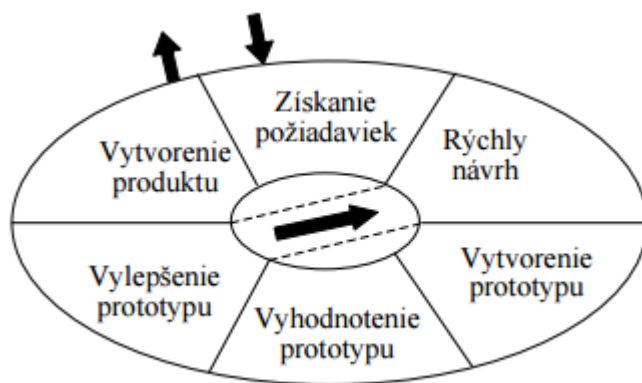
Výhody:

- zvýšené zapojenie zákazníka do vývoja
- vďaka ukážke prototypu si zákazník vie lepšie predstaviť výsledný produkt a teda dokáže aj presnejšie zadať požiadavky na to, čo požaduje
- chyby môžu byť odhalené omnoho skôr, a teda aj ich odstránenie bude lacnejšie

Nevýhody:

- tvorba prototypu sa môže stať extrémne nákladnou

- recyklácia prototypu na výsledný produkt
- zákazník sa môže stratiť v tom, čo očakáva a čo zatiaľ vidí



Obr. 2.2: Prototypový model

2.3 Evolučný model

Na začiatku je vytvorená základná funkčnosť pozostávajúca z istej časti požiadavkov. Táto základná funkčnosť je následne prezentovaná zákazníkovi, ktorý ďalej zadáva potrebné požiadavky na smer a vývoj aplikácie. Každá iterácia pridáva k hotovému produktu ďalšiu funkcionality. Testovanie sa uskutočňuje po vyvinutí funkcionality definovanej v danej iterácii, a následne aj po implementácii iterácie do celého projektu [3].

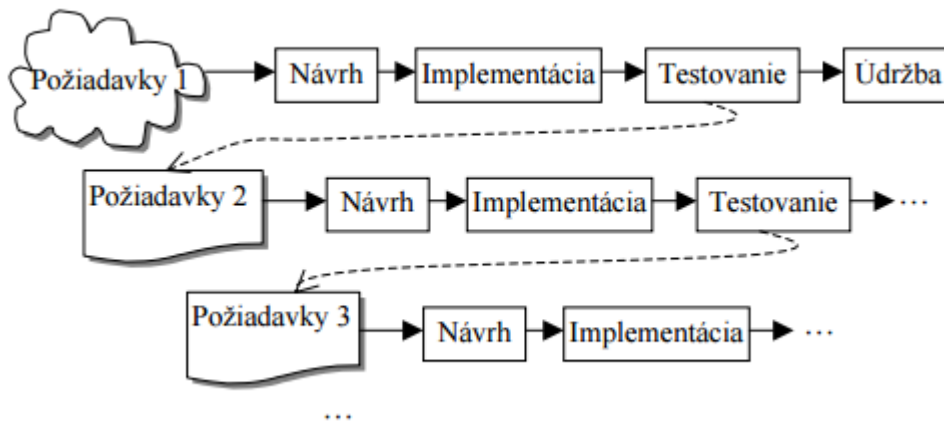
Výhody:

- kvalitnejšie a častejšie testovanie
- základná funkčnosť je vyvinutá relatívne skoro na začiatku projektu
- je vhodný pre projekt s kratšou dobou životnosti

Nevýhody:

- proces nie je viditeľný
- komplikovanejšia správa projektu

- množstvo zdrojov vynaložených na vývoj môže byť väčšie ako u iných modelov



Obr. 2.3: Evolučný model[3]

2.4 Inkrementálny model

Na začiatku vývoja sú jednoznačne definované požiadavky. Vývoj produktu prebieha po jednotlivých inkrementoch. Inkrement je jedna časť z celkového produktu. Takže pokiaľ by šlo o systém na predaj kníh, jeden inkrement môže byť registrácia používateľov, druhý inkrement objednávanie kníh a tak ďalej. Takto po častiach je systém odovzdávaný používateľovi. Samotné testovanie nastáva po vyvinutí jednotlivých inkrementoch, ako aj po zapojení ikrementu do celkového systému[3].

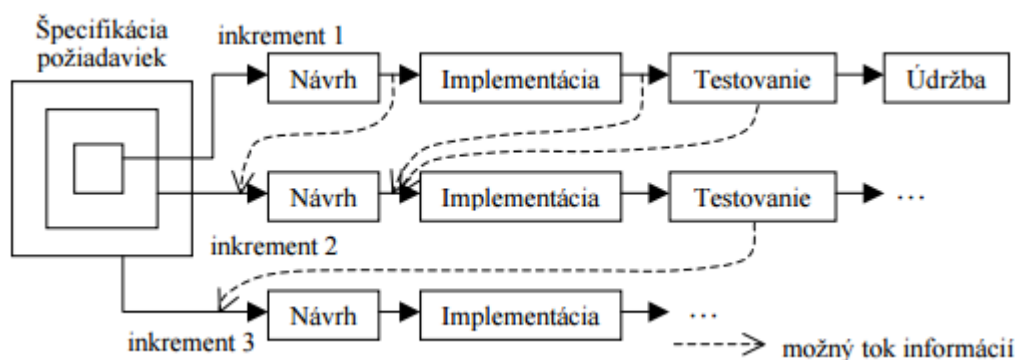
Výhody:

- všetky požiadavky sú známe od začiatku vývoja
- proces je dobre viditeľný
- systém je čiastočne funkčný pred koncom vývoja

Nevýhody:

- spravovanie tohto modelu môže byť náročnejšie ako iné modely

- v prípade neúplnej špecifikácie môže nastať vážny problém v návrhu



Obr. 2.4: Inkrementálny model[3]

2.5 Agilný model

Hlavným znakom agilného vývoja je jeho možnosť prispôbiť sa daným podmienkam. Sú tu využité rôzne modely vývoja, a každá je prispôbená potrebám pre daný projekt. Projekt je rozdelený na menšie časti, ktoré sú vyvíjané v tzv. šprintoch. Šprint je krátke obdobie 1-4 týždňov, počas ktorého prebieha definícia požiadavkov, vývoj, testovanie a implementácia do výsledného projektu. Často sa stáva, že sa jednotlivé šprinty prelínajú. To je spôsobené snahou o dosiahnutie čo najlepšej efektivity počas vývoja. Môžeme mať prebiehajúci šprint, ktorý sa blíži ku koncu tzn. už sa len doladujú detaily, končí sa testovanie, akceptujú sa požiadavky a programátori nemajú v tomto čase čo robiť. Práve vtedy nastáva príprava na ďalší šprint, rozoberajú sa jednotlivé požiadavky od zákazníka, hodnotí sa ich význam, obtiažnosť a podobne. Po začatí ďalšieho šprintu sa predchádzajúci dostáva do fázy nasadzovania, teda začlenenia do aktuálneho projektu, ktorý už zákazník môže využívať. Podľa potreby sa v tomto čase spúšťajú tzv. regresné testy, ktoré majú za cieľ odhaliť chyby po implementácii najnovšieho šprintu do celého projektu[4].

Výhody:

- jednoduchá správa
- dobrý model pre väčšie projekty

- funkčné časti sú rýchlo za sebou vyvíjané a ukázané zákazníkovi
- zákazník je zapojený do procesu vývoja
- väčšia flexibilita pri vývoji

Nevýhody:

- kvôli pravidelnosti tvorby nových verzií je nutné striktne dodržiavať termíny
- výrazne väčšia závislosť na klientovi
- noví členovia tímu sa môžu ťažšie začleniť

3 Testovanie softvéru

3.1 Tester

V dnešnej dobe má správny tester okrem klasického prístupu k testovaniu aj mnoho iných pomôcok a zodpovedností. Dôležité je si uvedomiť, že testerom nie je len človek, čo je na to špecificky vyhradený, ale testerom môže byť aj vývojár.

V menších tímoch, v ktorých na samostatného testera nezostávajú prostriedky sa stáva, že takýmto testerom sa stávajú sami programátori. V takých prípadoch je najlepšie, pokiaľ svoju vyvinutú časť netestuje vývojár, ktorý ju naprogramoval, ale niekto iný. Dôvod je ten, že pokiaľ niečo vývojár naprogramuje, je možné, že sa na to nedokáže pozrieť s dosť veľkým odstupom a testuje len to, čo mu príde dôležité. Zatiaľ čo iný vývojár sa na to pozrie a prístupuje k tomu inak, a teda je väčšia šanca, že nájde chyby aj tam, kde by ich človek, ktorý danú funkcionality vytvoril neuvidel.

Z vyššie uvedeného je zrejmé, že najlepšie je, pokiaľ má tím k dispozícii človeka, čo je zameraný výhradne na testovanie. Tester nesie pri vývoji softvéru podobnú a mnoho krát aj väčšiu zodpovednosť než vývojári, pretože ak tester prehliadne zásadnú chybu a dôjde k predaniu tohto chybného softvéru zákazníkovi, nastane mnoho problémov. V krajnom prípade môže celý projekt neúspešne skončiť [6].

Primárnou vlastnosťou každého testera je schopnosť myslieť trochu inak. Na rozdiel od programátora, ktorý sa snaží urobiť funkčnú aplikáciu, cieľom testera je naštudovať si aplikáciu a následne odhaliť jej chyby aj tam, kde by ich nikto iný nečakal. Zaujíma sa o veci typu: čo ak toto urobím inak, prečo to funguje týmto spôsobom a podobne. Nejde mu len o to, aby všetko fungovalo tak, ako je to poznamenané, tester musí vyskúšať aj scenáre, ktoré nie sú očakávané. Práca testera nekončí len pri samotnom testovaní, do jeho kompetencie spadajú aj iné činnosti ako:

- tvorba testovacích scenárov
- údržba týchto scenárov
- písanie automatických testov
- akceptovanie vytvorenej funkcionality so zákazníkom
- zadávanie chýb
- tvorba špecifikácie

Samozrejme, vždy záleží na firme, skúsenostiach a znalostiach testera, čo všetko bude v tíme vykonávať. Často sa stáva, že práve tester príde s rôznymi návrhmi na zmenu niektorých častí projektu, ktoré vedú k zvýšeniu jeho kvality a spokojnosti zákazníka.

3.2 Typy testovania

3.2.1 Funkčné a nefunkčné testovanie

- Funkčné testovanie je spôsob testovania, kde sa na aplikáciu pozeráme ako celok, alebo časť celku. Na začiatku aplikácii podám vstup a očakávam od nej výstup. Na jeho základe dokážem určiť, či daná činnosť bola vykonaná správne alebo nie. Záleží na tom, aby výsledok odpovedal požiadavkám zákazníka.
- Nefunkčné testovanie je typ testovania, ktoré overuje splnenie nefunkčných požiadaviek aplikácie. Tieto požiadavky nesúvisia s funkcionalitou aplikácie, ale ovplyvňujú jej kvalitu. Ide o oblasti ako: bezpečnosť, spoľahlivosť, rýchlosť, dokumentácia a podobne.

3.2.2 Statické a dynamické testovanie

- Statické testovanie je metóda testovania, ktorá k svojmu vykonávaniu nevyžaduje, aby bol program spustený. Ide hlavne o testovanie zadaných špecifikácií, požiadaviek a podobne. Statické testovanie tiež zahŕňa kontrolu kódu v porovnaní s dokumentáciou. Cieľom je čo najskôr odhaliť chyby[6].
- Dynamické testovanie k svojmu vykonávaniu už vyžaduje, aby aplikácia bola spustená. Overuje sa interakcia so systémom, teda vkladanie vstupných hodnôt a následná kontrola výstupu[6].

3.2.3 Testovanie čiernej, bielej a šedej skrinky

- Testovanie čiernej skrinky: Spočíva v tom, že k aplikácii pristupujeme pri testovaní ako k celku bez znalosti vnútorného fungovania. Záleží nám na tom, aby po vložení vstupných dát výsledok odpovedal očakávaniam. Výhoda tohto prístupu je v tom, že k aplikácii pristupujeme ako akýkoľvek iný používateľ a môžeme vyskúšať, ako sa aplikácia správa, keď s ňou pracuje zákazník. Nevýhodou je,

že pokrytie testovaného kódu aplikácie môže byť značne nerovnomerné a niektoré časti aplikačnej logiky nemusia byť vôbec otestované[6].

- Testovanie bielej skrinky: Spočíva v tom, že tester už nevidí len vonkajšiu časť systému, ale aj jeho zdrojový kód. Vidí ako jednotlivé funkcie na seba naväzujú a spolupracujú. Na základe toho je schopný lepšie odhadnúť, kde je systém zraniteľnejší a lepšie naplánovať testy. Cenou za to je fakt, že pri tomto spôsobe testovania strácame pohľad bežného používateľa[6].
- Testovanie šedej skrinky: Nachádza sa medzi bielou a čiernou skrinkou. Tester síce vidí na interné fungovanie aplikácie, ale nemá už prehľad o kóde. Vie o vnútornej štruktúre programu a vďaka tomu dokáže na rozdiel od čiernej skrinky písať presnejšie a vhodnejšie testy.

3.2.4 Testovanie splnením a zlyhaním

- Testovanie splnením: Ide o postup pri testovaní, kde do aplikácie vkladáme správne dáta a kontrolujeme, či sa výstup zhoduje s tým, čo sme očakávali. Cieľom nie je testovať odolnosť aplikácie na nesprávne dáta, ale overujeme, že pri správnych vstupoch sa aplikácia chová podľa očakávaní[6].
- Testovanie zlyhaním je presným opakom testovania splnením. Naším cieľom je otestovať aplikáciu práve tým, že do nej vkladáme nesprávne dáta. Pri tomto spôsobe testovania sa práve najviac prejaví schopnosť testera uvažovať inak a hľadať krajné scenáre použitia, ktoré vývojár mohol zabudnúť ošetriť. Príkladom tohto postupu môže byť napríklad: mám pole pre vstup s názvom rodné číslo. No namiesto rodného čísla zadám reťazec „Ahoj“ poprípade, ak je toto pole povinné, tak ho nevyplním vôbec a všimam si ako aplikácia reaguje. Snahou je odhaliť slabiny v kóde a slabo ošetrené miesta[6].

3.3 Testovanie v rámci vývoja

Počas vývoja aplikácie sa stretneme s mnohými skupinami testov. Každá z nich pristupuje ku kontrole softvéru inak a tiež kontroluje iné časti. Niektoré z nich sa dajú automatizovať a niektoré je lepšie nechať na ručnej práci testera.

3.3.1 Testovanie požiadaviek

Než sa začne so samostatným vývojom, je dobré najskôr venovať istý čas kontrole toho, čo zákazník požaduje od aplikácie. Už v tejto skorej fáze vývoja je možné odhaliť mnohé slabé miesta a nedostatky, ktoré by neskôr mohli spôsobiť problémy, alebo by si vyžadovali zmenu samotného kódu. Príkladom môže byť chýbajúca alebo neúplná funkcionálna aplikácia. Táto oprava býva spravidla jednoduchá a lacná pre zákazníka. Pri tomto type testovania sa využijú ako programátori, tak aj samotní tester. Obe skupiny sa na požiadavky dokážu pozrieť z iného uhla pohľadu, a teda objaviť najkritickejšie miesta[6].

3.3.2 Testovanie programátorom

Ide o základne testovanie, ktoré po sebe uskutočňuje programátor. Zaujíma sa hlavne o funkčnosť jednotlivých metód, tak isto ale aj o to, či je kód dobre a čitateľne napísaný. Malo by byť základnou povinnosťou každého vývojára vždy po sebe otestovať svoj kód, než je tento kód začlenený do projektu, či už manuálnym spustením danej časti aplikácie alebo vytvorením automatického testu.

3.3.3 Jednotkové testy

Sú to testy, ktoré skúmajú správnu funkčnosť menších celkov aplikácie, ktoré väčšinou zahŕňajú testy izolovaných tried alebo ich vzájomnej interakcie. Vo väčšine prípadov tieto testy tvorí práve vývojár, ktorý naprogramoval danú funkcionálnu. Najlepšie keď každá vytvorená metóda má svoj vlastný jednotkový test. Tento test metóde posúva vstupné dáta a kontroluje, či sa výsledok zhoduje s očakávaniami. Veľkou výhodou týchto testov je to, že keď dôjde k úprave metód či už z dôvodu refaktoringu, alebo úprave spôsobu akým metóda funguje, tak na konci stačí spustiť už predtým napísaný jednotkový test, ktorý ukáže či funkcia metódy stále odpovedá tomu, čo sa od nej očakáva. Pri jednotkových testoch sú dosť striktné požiadavky na jasne definované rozhrania, ako spolu triedy komunikujú, aké sú medzi nimi závislosti, inak je testovanie izolovanej časti aplikácie veľmi komplikované[11].

3.3.4 Assembly testy

Ide o skupinu testov, ktoré slúžia na skontrolovanie, či je možné jednotlivé časti kódu spojiť do jedného celku. Túto skupinu testov môže mať na

starosti vývojár, prípadne je možné mať na assembly testy priamo vyhradeného človeka. To sa stáva hlavne pri väčších projektoch.

3.3.5 Smoke testy

Tieto testy už nemá na starosti vývojár, ale tester. Ide o testy základnej funkcionality, ktoré sa vykonávajú pred tým, než sa tester pusti do hlbšieho skúmania aplikácie. Smoke testy majú za úlohu objaviť základne chyby vo funkcionalite. Pri väčších aplikáciách sa tieto testy môžu pravidelne opakovať, a vtedy je vhodné ich automatizovať. Výhoda automatických smoke testov je v tom, že pokiaľ vývojár upraví istú časť kódu, stačí spustiť túto sadu testov a tým sa overí, či základná funkcionality a komunikácia medzi modulmi alebo aplikáciami zostala zachovaná. Po úspešnom ukončení smoke testov už nič nebráni podrobnému testovaniu.

3.3.6 Integrované testy

Jedny z najdôležitejších testov sú práve integračné testy. Ich úlohou je kontrola komunikácie medzi jednotlivými modulmi aplikácie, poprípade komunikácia medzi aplikáciami. Pri testovaní tester postupuje od kontroly jednotlivých modulov, až ku kontrole aplikácie ako celku. Pokiaľ máme väčšiu aplikáciu, na ktorej spolupracuje viac tímov, je táto časť kritická na komunikáciu medzi jednotlivými tímami, keďže sa nepracuje len s modulmi, ktoré boli vyvinuté v rámci jedného tímu. Pokiaľ ide o automatizáciu tak pri integračných testoch sa už nedá jednoznačne povedať či automatizovať alebo nie. Vždy záleží od testu, či má zmysel ho automatizovať alebo nie, toto rozhodnutie nakoniec spadá na testera[5].

3.3.7 Systémové testy

Dajú sa považovať za časť integračných testov. Stáva sa, že tieto dva druhy testov sa spájajú do jedného tzv. SIT - Systémové integračné testy. Pri systémových testoch sa tester pozerá na aplikáciu ako na jednu entitu z pohľadu zákazníka. Kontroluje, či aplikácia funguje správne ako celok, či vracia požadované výstupy. Kontrolujú sa hraničné situácie ako aj požiadavky zákazníka a teda, či ich aplikácia spĺňa presne tak ako boli definované. Pri odhalení chyby je testovaná časť vrátená vývojárovi, ktorý na nej pracoval a po jej opravení dochádza ku kontrole, či sa daná chyba už nevyskytuje. Pod systémové testy patria aj iné testy ako napríklad:

- Bezpečnostný test sa zaoberá bezpečnosťou aplikácie, ako napríklad pracuje s osobnými údajmi, ako je chránená proti možným útokom. Na tieto typy testov sú väčšinou vyčlenení špeciálni testeri, ktorí sa venujú práve bezpečnosti.
- Zátťažový/Výkonnostný test skúma, ako sa aplikácia bude chovať pod veľkým náporom používateľov. Či nespadne, či sa nezhorší odozva jednotlivých akcií a podobne.
- Údržbový test kontroluje, ako sa zachová aplikácia pri zmene HW, SW, upgrade atď.

3.3.8 Akceptačné testovanie

Z pohľadu zákazníka sú toto najdôležitejšie testy. Aplikácia je vo svojej finálnej podobe a je prezentovaná zákazníkovi, ktorý kontroluje, či boli všetky jeho požiadavky splnené. Stáva sa, že v tejto fáze si zákazník testuje aplikáciu sám, aby si ju vyskúšal podľa seba. Pokiaľ aplikácia, alebo jej časť neprejde aplikačným testovaním, je vrátená do vývoja na opravu chyby, po ktorej nasleduje znova rovnaký proces, ako po prvotnom vývoji. Stať sa však môže aj to, že si zákazník uvedomí, že určité časti sa mu nepáčia a sú potrebné navrhnuť odznova, vtedy sa pri nich začína od návrhu. Tento prípad je najdrahší, keďže sa celý proces opakuje. Po úspešnej akceptácii, dostáva zákazník aplikáciu pre svoje použitie.

3.3.9 Testovanie po akceptovaní

To, že aplikácia bola akceptovaná ešte vždy nemusí znamenať koniec testov alebo vývoja. Aj keď boli vykonané všetky fázy testovania, stále je dosť vysoká šanca, že niektoré chyby zostali neodhalené. Tieto chyby sa objavujú už počas reálneho fungovania aplikácie. Po opravení nových chýb je potrebné znovu otestovať aplikáciu, či boli dané chyby naozaj odstránené. Okrem toho nemusí ísť len o nové chyby, ale aj o pridanie novej funkcionality. V tomto prípade sa testuje už pomocou vyššie spomenutých postupov.

3.3.10 Regresné testovanie

Úlohou regresných testov je overiť, že oprava chýb alebo pridávanie novej funkcionality nepridalo nové chyby v častiach, ktoré predtým fungovali. Regresné testy pozostávajú zo scenárov, ktorých počet závisí od veľkosti aplikácie. Tieto scenáre zostávajú väčšinou nemenné, zvyknú sa však rozširovať

po pridaní novej funkcionality, aby aj ona bola pokrytá. Regresné testy je vďaka tomu veľmi výhodné testovať automaticky.

3.4 Hlásenie a sledovanie chýb

Po úspešnom odhalení chyby je potrebné s ňou ďalej pracovať. V tomto prípade ide o jej zavedenie do systému na hlásenie chýb. Tento krok je dôležitý, keďže s chybou nepracuje len tester, ale aj vývojár. Preto sa očakáva od testera, že je schopný chybu dostatočne kvalitne opísať. Ide hlavne o kroky ktoré ho k nej doviedli. Môže sa stať, že chybu objavil náhodou alebo sa vyskytuje len v určitých špecifických prípadoch. Vtedy je najlepšie oblasť chyby čo najviac spresniť, nech je jasné kde sa treba zamerať a hľadať. Pokiaľ si tester nie je istý správnou funkčnosťou je potrebné všetko konzultovať. Oznamovanie chýb prebieha rôznymi spôsobmi. Dá sa to riešiť ústne, teda tester vývojárovi oznámi, kde chyba je a ako k nej prišiel a vývojár sa následne pustí do opravy a po jej dokončení znova informuje testera, že ju môže skúsiť otestovať. Samozrejme tento spôsob nie je efektívny, dá sa využiť pri malých tímoch, inak je najlepšie použiť systém na sledovanie chýb[6]. Medzi takéto systémy môžeme počítať aj technológie také jednoduché, ako zdieľaný excel, ale aj:

- OneNote - Ide o program MICROSOFT-u na tvorenie poznámok ktorý je sofistikovanejší v porovnaní s notepadom[7].
- Asana - Komplexnejší systém pre web a mobilné telefóny. Slúži na kontrolovanie vývoja častí aplikácie ako aj na sledovanie chýb[8].
- JIRA - Jeden z uznávaných softvérov slúžiaci na kontrolovanie chýb ako aj správu projektu[9].
- Bugzilla - Open source softvér na sledovanie chýb[10].

Zadávanie a správa chýb má istú štruktúru, aby sa v zadanej chybe dalo čo najlepšie orientovať a aby dôležité informácie boli čo najľahšie dostupné.

- Identifikátor - Jednoznačné označenie chyby, aby pri vyhľadávaní nedošlo k zámene za inú chybu.
- Názov - Mal by to byť stručný a jasný popis, ktorý identifikuje chybu a je podľa neho možné porovnať iné názvy, a tak nájsť podobné chyby.
- Popis - Obsahuje jednotlivé kroky, ktoré viedli k vzniku chyby.

- Stav - Popisuje v akom stave sa chyba nachádza. Či je len zadaná, alebo na nej už niekto pracuje, poprípade či je nasadená a pripravená na opätovné otestovanie.
- Priorita - Označuje závažnosť chyby a jej dopad na danú funkcionálnosť. Podľa priority sa vývojár rozhoduje, kedy začne s opravou a aké dôležité je v danom momente odstránenie tejto chyby.
- Priradenie - Označuje, kto na chybe práve pracuje.
- Prostredie - Pokiaľ sa testuje na viacerých verziách jednej aplikácie, alebo aj na rôznych databázach je dobré uviesť kde a v ktorej verzii sa daná chyba vyskytla.

Každá chyba si prejde potrebnými stavmi až je opravená a pripravená na nasadenie, po ktorom nasleduje otestovanie chyby. Pokiaľ je chyba opravená, je uzavretá a pokračuje sa ďalej v testovaní, pokiaľ sa objavili ešte problémy, je chyba vrátená naspäť na opravu. Nie je neobvyklé, ak opravenie jednej chyby, vyvolá chybu úplne inú, v takom prípade sa stará chyba ukončí a zadáva sa nová chyba, ktorá prechádza rovnakým procesom.

[Show Advanced Fields](#) (* = Required Field)

* **Product:** Spider
Sécretions

Reporter: ducky.peter@gmail.com

* **Component:** Digestive Goo
Venom
Web

Component Description
Select a component to read its description.

* **Version:** unspecified

Severity: normal

Hardware: PC

OS: Windows NT

We've made a guess at your operating system and platform. Please check them and make any corrections if necessary.

* **Summary:**

Description:

Comment	Preview
Steps : 1. Spustil som aplikaciu 2. Klikol som na policko Expected result : Uloženie malo prebehnúť v poriadku Result : Objavila sa chyba pri ukladaní	

Attachment:

Obr. 3.1: Systém na zadanie chyby

4 Manuálne testovanie

Je to prvý a najjednoduchší spôsob testovania. Nebolo to tak dávno, keď ručné testovanie bolo hlavným spôsobom testovania. Postupom času sa tento stav mení, no dôležité však je to, že ručného testovania sa nikdy nezabúvame a vždy bude potrebný aj tento typ testovania. Môže prebiehať dvoma spôsobmi.

4.1 Testovanie so scenárom

Tester má k dispozícii scenár, kde sú zadané jednotlivé kroky čo má otestovať. Tester si naštuduje scenár, prečíta si jednotlivé kroky a následne sa pustí do ich vyhodnocovania. Väčšinou scenár pozostáva z dvoch častí. Jedna je, aký krok sa má vykonať a druhá je, čo má byť výsledkom daného kroku. Príkladom tohto môže byť: používateľ vyplní na obrazovke správa osoby všetky povinné polia a následne klikne na tlačidlo uložiť. Výsledok: údaje sú vyplnené a uloženie prebehlo úspešne. V tomto bode je potrebné skontrolovať, či sa tak stalo. Ak áno, tester pokračuje ďalším krokom, pokiaľ nie mohli nastať dve situácie. Prvá, došlo k zmene funkcionality a teda po stlačení tlačidla ulož sa môže objaviť napríklad kontrolné okno, či si používateľ naozaj praje uložiť údaje. V takomto prípade treba túto zmenu konzultovať s analytikom, poprípade vývojárom, ktorý to má na starosti a ak je to naozaj tak, potom je potrebné danú zmenu zaznamenať do scenára a tester môže v testovaní ďalej pokračovať. Druhá možnosť je, že sa objaví chybová hláška, alebo dané okno sa nemalo objaviť a tiež je to chyba. V tomto momente zadáva tester chybu do systému na sledovanie chýb tzv. bug tracking. Následne sa testovací scenár označí, že sa v ňom našla chyba a tester prejde na ďalší. Po opravení chyby sa tester vracia k tomuto scenáru a testuje ho znova. Po úspešnom prejdení scenáru je scenár označený ako ukončený[6].

4.2 Testovanie bez scenára

Keď tester nemá k dispozícii scenár, musí k testovaniu pristupovať pomocou svojho najlepšieho úsudku a skúšať testovať, čo ho napadne. V takomto prípade je veľmi dôležité, aby si udržal prehľad a postupoval metódicky. Pokiaľ začne testovať chaoticky, tak s týmto prístupom rýchlo stratí prehľad čo otestoval a čo nie. Najlepšie pri takomto testovaní je po otestovaní istej časti vytvoriť testovací scenár, kde zaznamená jednotlivé kroky a

očakávané výsledky, aby v budúcnosti keď sa vráti nemusel testovať "na slepo". Kontrola chýb je tiež o niečo komplikovanejšia, keďže nemá scenár, ktorý by ho viedol v tom, ako ma výsledok po kliknutí na tlačidlo vyzeráť, musí sa spoľahnúť znovu na svoj úsudok a posúdiť, či je to rozumné alebo nie. V momente keď sa mu niečo nezdá, alebo si nie je istý je najlepšie ísť sa spýtať vývojára, poprípade analytika, ktorý má daný projekt na starosti. Následne už pokračuje tak, ako keby mal scenár. Ak to nie je chyba pokračuje ďalej, ak to chyba je zadáva ju do systému na sledovanie chýb.

4.3 Tvorba scenára

Než sa vôbec pustíme do písania testovacieho scenára, je veľmi dôležité oboznámiť sa s funkcionalitou, ktorú budeme v tomto scenári opisovať. Najlepšie je zobrať si špecifikáciu, ktorá sa venuje danej časti a prečítať si ju. Odtiaľto získame potrebné znalosti, ako sa jednotlivé časti majú chovať a podľa toho vieme, čo treba do scenára zahrnúť. Pokiaľ však špecifikácia napísania nie je, musíme sa spýtať ľudí, ktorí na danej funkčnosti pracovali a zistiť toho čo najviac. Následne si získane vedomosti vyskúšame v aplikácii a premyslíme si, čo sa nám v scenári hodí, čo necháme na ďalší scenár, poprípade, čo nie je potrebné. Na samotnú tvorbu môžeme použiť čokoľvek, čo nám príde vhodné. Môže to byť notepad, excel, poprípade novší program ako test manager a podobne.

Názov scenára má byť jednoduchý a presný. Keď si prečítame názov, malo by nám byť jasné, kde v danej aplikácii sa približne budeme pohybovať, poprípade čo budeme testovať. Ak ideme testovať ukladanie zmien na osobe, môžeme nazvať scenár: zmena detailu osoby. Dbáme na to, aby bol názov presný a nebol príliš dlhý, preto v názve scenára môžeme používať zápis bez medzier teda: ZmenaDetailuOsoby poprípade použiť _: Zmena_detailu_osoby. Pokiaľ máme viac scenárov z jednej sekcie je vhodné ich aj označovať číslom, zlepší to prehľad o počte scenárov na daný modul. Teda ZmenaDetailuOsoby_01.

Obsah scenára musí obsahovať postup, ako bude tester prechádzať daným scenárom a tým aj testovať funkčnosť. Takže sa v scenári majú nachádzať jednotlivé kroky, ktoré je treba vykonať, ako napr. kliknem na tlačidlo ulož, zadám rodné číslo do text boxu s názvom rodné číslo a podobne. Následne za každým jednotlivým krokom, nasleduje výsledok daného kroku napríklad: po kliknutí na tlačidlo uložiť došlo k uloženiu formulára. Rovnako môžeme zadať kroky, ktoré budú obsahovať kontrolu, či niečo funguje správne pri zadaní nesprávnych dát, napríklad do poľa rodné číslo za-

daj emailovú adresu, pričom výsledok bude: objavila sa hláška o nesprávnom vstupe. Záleží od toho ako sa aplikácia v danej situácii má zachovať. Samozrejme, najlepšie je pokiaľ v scenári nie je vždy všetko presne napísané, napr. aké hodnoty ma tester zadať. Stačí ukázať príklad a zvyšok nechať na testerovi. Dôležité sú kroky a výsledky, aké majú nastať a ostatné je na testerovi, čo zadá do daných text boxov a podobne.

Dodatočný obsah scenára, môže zahŕňať napríklad popis toho, čo je povolené do jednotlivých polí nastaviť a v akom formáte, ak však je viac scenárov pracujúcich s jedným oknom, je lepšie vytvoriť samostatný súbor, kde tieto údaje budú popísané a v scenári naň nechať len odkaz. Ak sú potrebné SQL dotazy, tak môžu byť pripravené v scenári, aby si ich tester nemusel konštruovať sám, poprípade tu môžu byť obrázky, ako má vyzeráť dané okno po otvorení.

4.4 Údržba scenára

Tvorbou scenára naša práca pri scenároch ešte nekončí. Ako sa aplikácia vyvíja, tak sa nepridáva len nová funkcionálna, ale upravuje sa aj stávajúca. A preto pri testovaní scenárov treba dávať pozor práve na takéto zmeny, poznačiť si ich na neskoršiu úpravu, alebo ak je čas, upraviť dané scenáre hneď. Dôležité je to preto, aby sme netestovali so starými verziami scenárov a tiež ak dané scenáre bude testovať po nás niekto iný, aby mal presný popis toho čo má robiť. Takáto údržba sa hlavne využíva pri väčších projektoch, kde sa pravidelne testuje nová aj stávajúca funkcionálna. Pri údržbe scenárov je vhodné si zachovávať aj staršie verzie, napríklad pokiaľ niekto urobí zmeny v scenári no nakoniec sa zistí, že dané zmeny nemajú zmysel, alebo je lepšie už vytvoriť rovno nový scenár. Vďaka zálohám je ľahšie sa vrátiť k starším verziám. Aby sme čo najlepšie odlíšili rôzne verzie rovnakého scenáru, je možné k názvu pripísať napríklad v0.5 a keď upravujeme scenár tak tento starý zálohujeme a upravenému zmeníme názov z DetailOsoby_01_v0.5 na DetailOsoby_01_v0.6. Samozrejme, ručné ukladanie a prepisovanie scenárov nie je jediná možnosť. Využiť sa dajú aj mnohé verzovacie systémy, napr. SVN, Git a podobne.

5 Automatické testovanie

Bez manuálneho testovania sa pri tvorbe softvéru nezaobídeme. Je to kľúčová činnosť pre overenie kvality produktu. Pre manuálne vykonanie testu je zakaždým potrebné vynaložiť rovnaké úsilie a venovať mu čas. Preto s počtom takto vykonávaných scenárov, rastie aj čas a zdroje zamerané na testovanie. Naopak, automatické testy v momente keď už sú napísané, nevyžadujú takmer žiadnu pozornosť. Treba ich spustiť a po dobehnutí len overiť výsledok, ako testy skončili. Automatické testy vyžadujú najviac zdrojov na začiatku, teda počas ich tvorby, no následne sa dajú opakovať niekoľko-krát denne s minimálnymi požiadavkami od testerov, ktorí sa môžu sústrediť na iné povinnosti.

Pokiaľ ide o testovanie, dochádza k rozdeleniu zamestnancov na testerov a programátorov automatických testov. Rozdiel je v náplni práce. Tester, je hlavne zameraný na manuálne testovanie, tvorbu scenárov, akceptáciu atď. Zatiaľ, čo vývojár automatického testu má hlavne na starosti premenu scenára na automatický test. Sú ňmu vyžadované hlbšie znalosti programovania a je viac zameraný na tvorbu automatických testov. Tvorba automatického testu pozostáva z niekoľkých krokov[6].

- Určenie, ako sa bude tvoriť automatický test. Teda či pôjde o UI test, integračný, alebo jednotkový test.
- Kontrola scenára. Než je vôbec možné začať písať automatické testy je dobré si overiť, či scenár podľa ktorého bude automatický test vznikať je upravený aby obsahoval najnovšie zmeny, alebo či má vôbec zmysel ho automatizovať. Popríklad sa automatický tester poradí s tvorcom scenára, pokiaľ mu niektoré kroky nie sú jasné.
- Tvorba automatického testu. V tomto momente dochádza k prepísaniu manuálneho testu do automatickej podoby.
- Testovanie testu na lokálnom zariadení. Keď je automatický test napísaný, je potrebné ho skontrolovať, či vykonáva kroky aké sú popísané v scenári, popríklad či nedochádza k chybám v kóde počas spúšťania testu a tiež je potrebné, aby sa výsledky testu zhodovali s očakávaniami.
- Kontrola kódu. V tejto časti dochádza ku kontrolovaniu kódu automatického testu iným programátorom. Pozerá sa hlavne na spôsob, akým je test napísaný, či sú dodržané štandardy alebo test nie je príliš neefektívny.

- Kontrolné spúšťanie na cieľovom prostredí. Keďže automatické testy sú spúšťané na iných zariadeniach, ako na ktorých sú vytvorené, je potrebné otestovať, či zmena prostredia neovplyvní výsledok testu.
- Údržba testu. Aj keď najväčšia časť práce skončila a test je napísaný, treba brať do úvahy aj to, že pôvodný scenár sa môže v rámci vývoja aplikácie meniť. A preto je potrebné kontrolovať, či nie je nevyhnutné upraviť aj daný automatický test.

Každý scenár je možné automatizovať, ale ako je spomenuté vyššie, nie vždy je to rozumné. Treba mať na pamäti, že prvotný vklad do tvorby automatického testu je vysoký, a preto pred každým automatizovaním je potrebné rozhodnúť, či sa to oplatí. Najvýhodnejšie je automatizovať scenáre, ktoré sú stabilné. To znamená scenáre, ktoré sa vykonávajú pravidelne, no zásahy do funkcionality opísanej v týchto scenároch sú minimálne. Často sa stretneme so scenármi obsahujúce viac technológií, ktoré navzájom komunikujú a než ich prepájať a tvoriť spôsoby ako budú v teste spolu komunikovať, je mnoho krát ľahšie testovať takýto scenár ručne.

5.1 Testovateľný kód

Pod pojmom testovateľný kód rozumieme kód aplikácie, napísaný tak, že zjednodušuje tvorbu jednotkových, či integračných testov. Najlepšie ho vystihuje pravidlo KISS (keep it simple stupid). Keď je tvorená čisto nová aplikácia, môžeme rovno toto pravidlo implementovať a písať jednoduchý kód, kde každá metóda vykonáva presne to, na čo je vytvorená a neobsahuje zbytočne ďalšiu funkcionality. Často sa však programátor stretne s tzv. Legacy kódom. Ide o kód, ktorý bol vytvorený dávnejšie a už nemusí byť podporovaný alebo udržiavaný, no stále sa využíva. Pokiaľ chceme tvoriť jednotkové testy na takýto kód, ktorý na to nebol stavaný, vzniká problém. Je obvyklé, že pri takomto kóde sa v jednej metóde nachádza napríklad počítanie premennej, zápis tejto premennej do databáze, načítanie hodnoty od používateľa a na konci výpis na výstup. No a presne takto testovateľný kód nevyzerá. Pokiaľ sa programátor stretne s niečím podobným a chce vytvoriť jednotkový test, potrebuje začať úpravou starého kódu. Teraz prichádza na radu refactoring, kde postupne rozkladá celú metódu na menšie časti, kde jednotlivé časti vykonávajú len jednu činnosť. Napríklad vznikne metóda na vypočítanie hodnoty, metóda na zápis do databáze a tak ďalej[12].

Samozrejme, je možné napísať jednotkový test aj na metódu vyššie spomínanú. Na metódu ktorá v sebe vykonáva viac operácií, avšak keďže test

dostane len určitý vstup a očakáva daný výstup a dôjde k chybe ktorá zapríčiní, že test skončí neúspešne, nedokážeme povedať kde nastala chyba. Bolo to v počítaní? Bolo to v zápise do databáze alebo vo výpise na výstup? Preto dôležitou vlastnosťou testovateľného kódu je to, že keď test skončí chybou, programátor je jednoznačne schopný povedať, ktorá časť nefungovala správne.

Preto k tvorbe čo najlepšieho testovateľného kódu je možné použiť techniku test first (najsôr testy), čo znamená, že najprv sa navrhnu rozhrania tried. Proti takému rozhraniu sú napísané testy a následne sa implementuje logika daných tried, ktorá by mala svojim chovaním odpovedať testom.

Dobrý testovateľný kód má k sebe pripísanú aj dokumentáciu a nepotrebuje mnoho komentárov k samotnej funkcionalite. Pokiaľ ide o názov metódy, mal by korešpondovať s tým, čo metóda vykonáva a tým názov prispeje k prehľadneniu.

5.1.1 Vloženie závislostí (Dependency injection)

Vkladanie závislostí je softvérový návrhový vzor, kde si pod závislosťou môžeme predstaviť určitý objekt a pod vložení (injection) predanie referencie na tento objekt klientovi, alebo objektu, ktorý s ním bude pracovať. Ak by objekt chcel využívať služby iného objektu bez použitia vkladania závislostí, bol by zodpovedný za celý životný cyklus tohto objektu. Pri vkladaní závislostí potom klient nevie o kóde ktorý prijíma, alebo ako vytvoriť službu, ktorú chce použiť. Klient vie len o rozhraní služby, ktoré používa[13]. Vloženie závislostí sa dá vykonať tromi spôsobmi:

- vkladanie konštruktorom (constructor injection)
- vkladanie metódou (method injection)
- vkladanie vlastnosťou (property injection)

5.1.2 Mock objekt

Mock-ovacie objekty, sú využívané v objektovo orientovanom programovaní pri tvorbe jednotkových testov. Ide o postup pri ktorom sa vytvorí tzv. mock objekt, ktorý simuluje správanie reálneho objektu. Využíva sa to hlavne pri objektoch, ktoré je počas testovania ťažké vytvoriť, alebo je s reálnym objektom náročné pracovať. V tom momente programátor môže využiť jednu z mnohých knižníc, ktoré umožnia vytvoriť takýto objekt a použiť ho namiesto reálneho[14]. Kedy využiť mock-ovací objekt:

- ak je náročne vytvoriť reálny objekt
- reálny objekt je pomalý a vďaka nemu by vykonanie testu trvalo príliš dlho
- neexistuje alebo sa jeho štruktúra môže zmeniť

Pre platformu .NET je možné využiť knižnicu Moq, ktorá je priamo celá od základu napísaná aby mohla využívať výhody, ako sú lambda abstrakcie alebo Linq[15].

5.2 Štruktúra testu

Automatické testy bývajú triedy, ktoré vďaka špeciálnej anotácii nájde testovací framework. Tento framework pri spustení testu vytvorí ich inštanciu, zavolá metódu s inicializáciou testu, metódu so samotným testom, ktorá nahlási, či test uspel alebo nie a nakoniec metódu na upratanie. Podľa potreby sa v ňom nemusia nachádzať všetky časti. Výhodou štrukturalizácie testu je hlavne to, že celý test nie je spísaný len v jednej časti, ale je rozdelený do viacerých, a teda dochádza k prehľadneniu testu, a pokiaľ využívame rovnaké premenné vo viacerých testoch zabránime duplicitám.

- Inicializačná časť. V tejto časti dochádza k prvotnému nastaveniu potrebných premenných, ktoré sú v teste využité a budú vždy potrebné pre správny chod testu/testov. Jednotlivé premenné vytvoríme pred touto metódou, a tu ich nastavíme na hodnoty aké chceme. Pred každým spustením testu je vykonaná inicializácia, ak je definovaná. Pokiaľ trieda obsahuje viac testov, tak inicializácia je spustená pred začiatkom každého testu.
- Telo testu. Tu je popísaná samotná logika automatického testu. Jednotlivé kroky, ktoré popisuje scenár sa nachádzajú práve tu. Tiež je zvykom, že pokiaľ využívame kontrolné asserty, ktoré slúžia na vyhodnotenie testu, tak sú práve tu. Pokiaľ test neobsahuje ďalšie časti, tak po vykonaní tela dôjde k vyhodnoteniu testu. V jednej testovacej triede sa môže vyskytovať viac ako jeden test. Avšak, vždy by mal jeden test testovať len jednu konkrétnu funkcionálnu, aby pri negatívnom výsledku bolo hneď jasné, čo nefunguje.
- Upratovanie testu. Pokiaľ test pracuje s komplikovanejšími štruktúrami, oknami atď. Vtedy práve využijeme upratovanie. Je tu definované, čo sa má vykonať po ukončení testu, aby bolo možné pustiť

d'alšie testy. Ide napríklad o pozatváranie okien aplikácie, alebo ak sa namiesto assertov v teste využíva zberač chýb, je zvykom ho umiestniť do tejto časti. Pokiaľ trieda obsahuje viac testov, v tom prípade sa upratanie vykoná za každým testom. Táto časť je presne ako inicializácia dobrovoľná, a preto sa nemusí vyskytovať v každej testovacej triede.

```
[TestClass()]
public class CalculateTests
{
    Calculate calc;

    [TestInitialize()]
    public void Initialize()
    {
        calc = new Calculate();
    }

    [TestMethod()]
    public void plus5Test()
    {
        Assert.AreEqual(9, calc.plus5(4), "Vysledok nie je spravny");
    }

    [TestCleanup()]
    public void Cleanup()
    {
        Console.WriteLine("Test ukonceny");
    }
}
```

Obr. 5.1: Štruktúra testu

5.3 Jednotkové testy

Najzakladanejšia jednotka automatického testovania určená na kontrolu malých častí, ako sú triedy a ich metódy. Na začiatku je vhodné triedu inicializovať do potrebného stavu, tento stav je zhodný pre všetky testy v danej triede. Preto, sa na to využije metóda `TestInitialize`. Trieda s testom obsahuje viac testovacích metód, každá z nich má svoj účel a obsahuje len také asserty, ktoré presne overujú to, čo daná testovacia metóda má kontrolovať. Príkladom kontrolovania môže byť napríklad vkladanie správnych vstupov, nesprávnych vstupov, kontrola na vyhadzovanie výnimky a podobne[12].

- Assert: Konštrukcia využívaná pri testovaní ktorá vracia TRUE alebo FALSE. Vďaka týmto konštrukciám sa test vyhodnotí buď úspešne

alebo neúspešne. K dispozícii máme pri assertoch viac možností, ako určité údaje vyhodnotiť a koľko ich kontrolujeme.

- Porovnávanie dvoch hodnôt: `AreEqual`, `AreNotEqual`, `AreSame`, `AreNotSame`. Porovnáva dva zadané objekty / hodnoty a podľa toho vráti `TRUE` alebo `FALSE`.
- Kontrola jednej hodnoty: `IsTrue`, `IsFalse`, `IsNull`, `IsNotNull`. Hodnotu ktorú predáme assertu skontroluje podľa zadanej podmienky a vyhodnotí.
- Chybová hláška: Ku každému assertu je možné dopísať aj text, ktorý nám po chybnom vyhodnotení dodá ďalšie informácie.

5.4 Testy užívateľského rozhrania

Rozdiel medzi užívateľskými testami a jednotkovými je hlavne v tom, že keď sme písali jednotkové testy vedeli sme, aké metódy a triedy existujú a ktoré priamo testujeme, bol to prístup testovania bielej skrinky. Zatiaľ, čo pri testoch užívateľského rozhrania ide o prístup testovania čiernej skrinky a teda máme pred sebou aplikáciu a chceme simulovať bežnú prácu zákazníka s aplikáciou. Takže využívame technológie, ktoré nám pomáhajú nasimulovať pohyb myšky, klikanie, zadávanie textu a podobne. Avšak to, čo sa deje na pozadí aplikácie, napríklad aká metóda sa zavolała keď sme klikli na tlačidlo uložiť, keď sme sa snažili zadať email a došlo k overeniu validity emailu, to už nevieme a zaujíma nás len to, či sa výstup zhoduje s tým, čo očakávame. Aby sme si boli istí výsledkami nejakej akcie je potrebné vedieť, čo sa naozaj má stať, a k tomu slúži dokumentácia ktorou sa riadime.

Keď hovoríme o testovaní užívateľského rozhrania sú dva typy aplikácií, ktoré testujeme a na ktoré sa ďalej pozrieme. Ide o windowsové a webové aplikácie. Na ich úspešné a ľahšie testovanie máme mnoho technológií a prístupov.

5.5 Testovanie windowsovej aplikácie

Aby sme dokázali ľahko a rýchlo vytvárať testy pre WinFormy, budeme potrebovať určité pomôcky. Nimi sú `white framework` a `UIVerify`. `White framework` využijeme na samotnú tvorbu testov, obsahuje mnohé užitočné pomôcky, vďaka ktorým, budeme schopní pracovať s aplikáciou a pracovať s jednotlivými prvkami ako sú textboxy, comboboxy, buttony atď.

UIVerify je veľmi šikovný nástroj pokiaľ ide o orientáciu v aplikácii, použijeme ho na to, aby sme boli schopní jasne identifikovať jednotlivé elementy našej aplikácie.

5.5.1 White framework

White framework je open-source projekt¹ napísaný v jazyku C#, ktorý slúži na prácu s mnohými .net technológiami ako sú: Win32, WinForm, WPF a SWT,² ktorý je pre JAVU. White framework ponúka objektovo orientované API³. Zatiaľ čo výhodou je, že white framework nevyžaduje prácu so skriptovacími jazykmi, tak za nevýhodu sa dá považovať to, že pracuje s elementami aplikácie, a teda aby sme s ním mohli pracovať potrebujeme program, ktorý nám zobrazí informácie o týchto elementoch.[16] Ide hlavne o informácie typu AutomationID⁴, meno, súradnice elementu a podobne. Medzi takéto programy patria napríklad:

- UIVerify
- UISpy
- Inspect

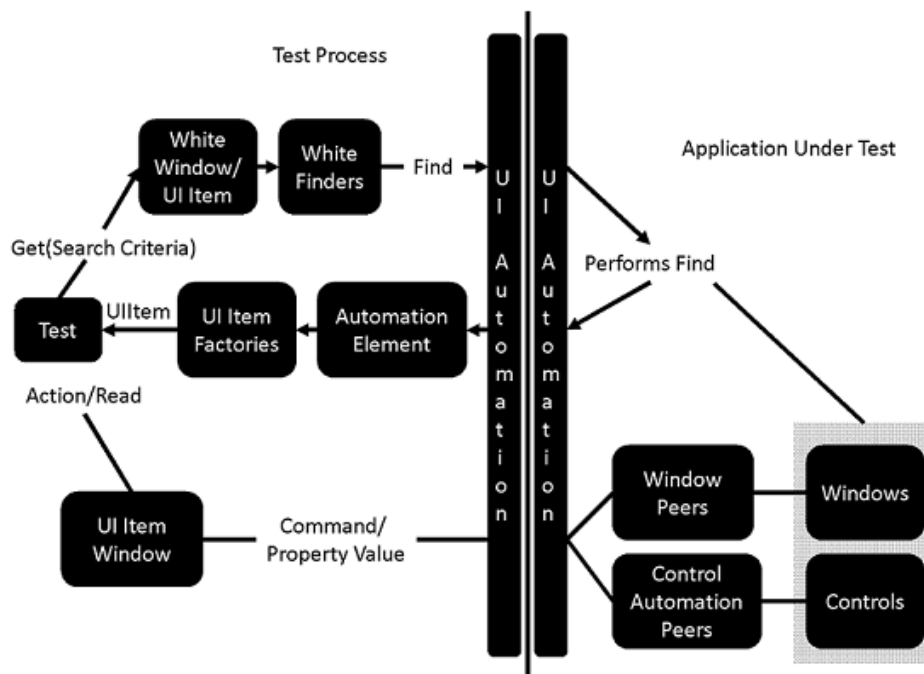
V momente, keď vďaka takémuto programu zistíme informácie o elemente, ktorý potrebujeme, dokážeme si ho pomocou white framework-u vyhľadať v aplikácii a pracovať s ním. Keď už máme vyhľadaný element a potrebujeme do neho niečo zapísať alebo prečítať, tak white framework má už priamo definované metódy na prácu s jednotlivými typmi, ako sú date-Formy, textboxy a podobne[16].

1. Open-source projekt alebo program je program ktorý ma verejný zdrojový kód a jeho šírenie je bez obmedzení.

2. Ide o technológie ktoré umožňujú pracovať s grafickým rozhraním aplikácie

3. API je zbierka metód tried alebo programov vďaka, ktorým sa vie akým spôsobom sa volajú funkcie knižníc zo zdrojového kódu.

4. AutomationID je jednoznačný identifikátor pre elementy aplikácie.



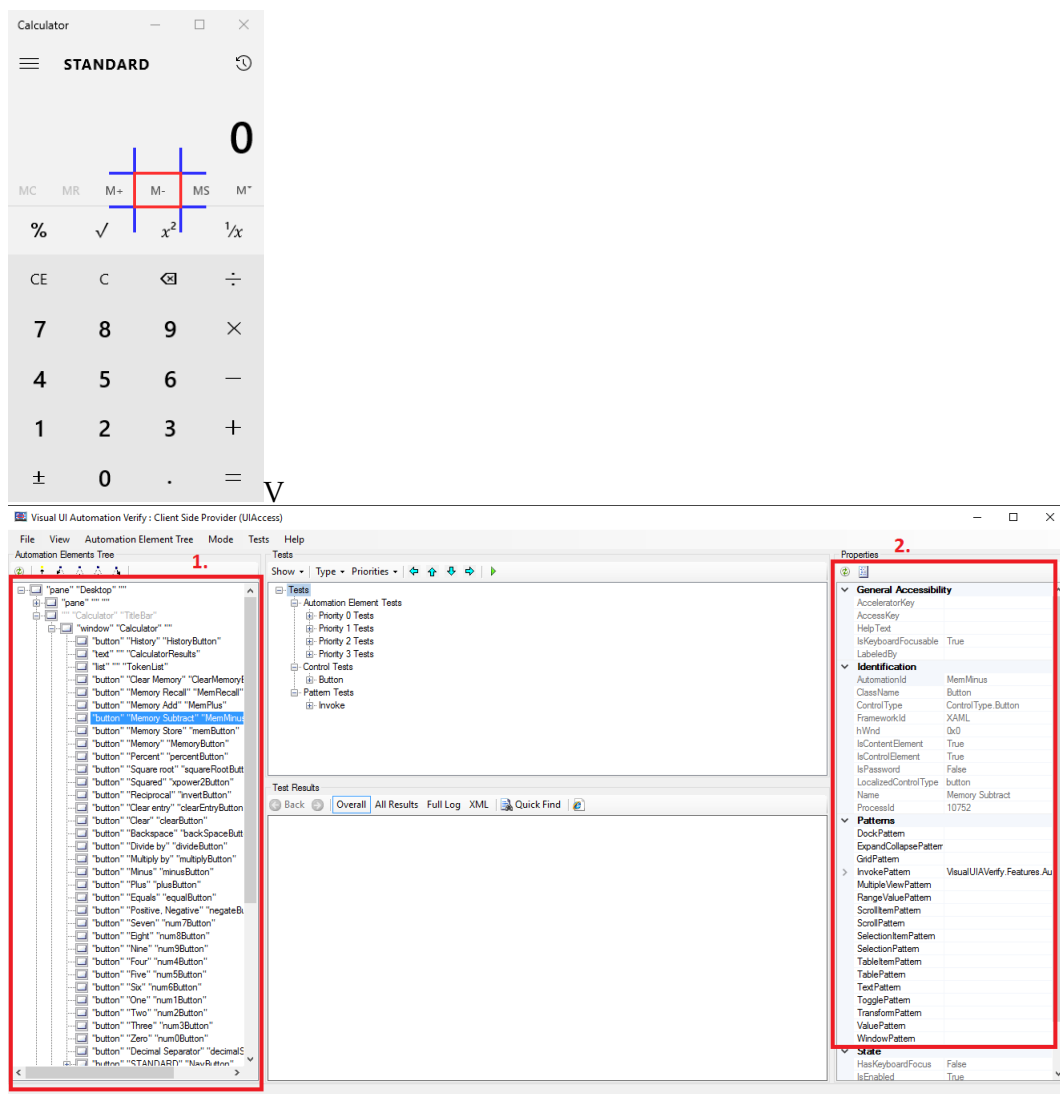
Obr. 5.2: Obrázok interakcie white framework-u s aplikáciou[17]

5.5.2 UIVerify

Ide o program, slúžiaci na prehľadávanie elementov v aplikácii a následné získavanie informácií o týchto elementoch pre potreby testovania, či už automatického alebo ručného. Po spustení si UIVerify načíta všetky otvorené aplikácie s ktorými dokáže pracovať. Tieto aplikácie sú zobrazené v stromovej štruktúre, pomocou ktorej dokážeme prechádzať medzi jednotlivými aplikáciami a jej elementami. Tak isto si po spustení UIVerify zaregistruje klávesové skratky pre ľahšie pracovanie. Pokiaľ sa stane, že niektorá skratka je už používaná iným programom, sme o tom informovaný hláškou o neúspešnej registrácii.

V okne označenom číslom 1, vidíme spomínanú stromovú štruktúru, kde v tomto momente je zobrazená kalkulačka a k nej prislúchajúce tlačidlá. Pokiaľ zvolíme element kalkulačky v tomto prípade ide o tlačidlo M- tak dôjde k tomu, že UIVerify označí toto tlačidlo priamo aj na aplikácii kalkulačky, aby tester/používateľ mohol vizuálne skontrolovať, k čomu sa

5. AUTOMATICKÉ TESTOVANIE



Obr. 5.3: UIverify s kalkulačkou

daný element priamo viaže. Pokiaľ má element nejakých potomkov, je možné ďalej tento element roztvoriť znamienkom + a tým sa nám zobrazia elementy prislúchajúce nadradenému elementu. V okne označenom číslom 2, vidíme informácie vzťahujúce sa priamo k zvolenému a označenému elementu, ako napríklad AutomationID, Name a podobne. UIVerify obsahuje aj ďalšie okná, no pre potreby tvorby automatických testov nám stačia práve tieto dve [18].

5.5.2.1 Ako získať UIVerify

Stiahnuť program UIVerify, alebo podobné ako UISpy, alebo Inspect nie je možné samostatne. UIVerify je súčasťou balíčku Windows Software Development Kit, ktorý obsahuje mnoho iných technológií, či už na vývoj alebo testovanie, no práve tento balík obsahuje UIVerify. Zistiť, či sa u vás v počítači nachádza UIVerify je ľahké, stačí vyhľadať súbor s týmto názvom, poprípade sa pozrieť do zložky program files, kde treba vyhľadať Windows Kits, v ktorých sa nachádza. Príklad úplnej cesty: C:\Program Files (x86)\Windows Kits\8.1\bin\x86\UIAVerify. Pokiaľ sa však takéto zložky u vás nenachádzajú, je potrebné dané súbory stiahnuť z Microsoft stránok. Odkaz na použité Windows kits v tejto bakalárskej práci je: <https://msdn.microsoft.com/en-us/windows/desktop/bg162891.aspx>. Potom treba len daný kit nainštalovať a je možné UIVerify používať.

6 Tvorba testu pre windowsovú aplikáciu

Samotnú tvorbu testu si rozdelíme na jednotlivé časti, ktoré budú zahŕňať prípravu prostredia na tvorbu testu. Teda založenie nového projektu a pripojenie white framework-u, takisto aj rozdelenie a zvolenie potrebných usingov¹ a v neposlednej rade aj spísanie prvého testu.

6.1 Príprava prostredia na tvorbu testu

Než môžeme písať samotný test, je potrebné založiť si nový projekt. Konkrétne ide o základný „Unit Test Project“ ktorý Visual Studio ponúka, ako štandardný projekt pre testovanie. Po vytvorení nového projektu je ešte potrebné pripojiť white framework, vďaka ktorému budeme pri teste môcť ovládať testovanú aplikáciu. Aby sme splnili túto časť, potrebujeme nájsť package manager console. Tá, sa nachádza v Tools -> NuGet Package Manager. Po úspešnom spustení package manager console, je potrebné do nej pridať a spustiť príkaz: Install-Package TestStack.White, ktorý pripojí white framework pre náš projekt. A v tomto momente máme projekt pripravený na tvorbu automatického testu[19].

6.2 Obsah Testu

6.2.1 Usingy

Každý test bude implementovať určité usingy. Niektoré základné sa budú vyskytovať v každom, no niektoré špecifické použijeme len v určitých prípadoch. Dosť zlým zvykom sa stáva, že pokiaľ automatický tester píše viac testov po sebe, tak pravé kopíruje usingy z jedného testovacieho projektu do druhého, prípadne pridá ďalšie, čo potrebuje a pri ďalšom projekte znova všetko skopíruje a pokračuje v teste. Takto dôjde k prehusteniu usingov a objaví sa tu mnoho nepoužívaných. Aj keď nepoužívaný using nič nepokazí, spôsobí neprehľadnosť kódu. Existuje mnoho pomôcok, ktoré pomáhajú vývojárovi a označujú ktoré premenné, v tomto prípade usingy sú využívané a ktoré nie. Vďaka tomu môže premazať nepotrebné usingy.

Na obrázku 6.1 vidíme základne usingy pre prácu s white frameworkom. Samozrejme tento framework ich obsahuje omnoho viac s dodatočnou funkcionalitou, ale pre vytvorenie základného testu ďalšie nie sú potrebné.

1. Using: Umožňuje programátorovi uviesť prefix už v čase kompilácie a nemusí teda vždy používať úplne názvy typov.

```
using TestStack.White.UIItems.Finders;  
using TestStack.White;  
using TestStack.White.UIItems;  
using TestStack.White.UIItems.WindowItems;
```

Obr. 6.1: Usingy v projekte

- Finders: obsahuje v sebe konštrukcie, ktoré nám pomôžu vyhľadať elementy pomocou ich mena alebo ID.
- White: vďaka tomuto usingu si môžeme vytvoriť typ aplikácie, ktorý nám bude na našu testovanú aplikáciu držať referenciu.
- UIItems: obsahuje konštrukcie pre základné typy ako sú tlačidlá, textBoxy a podobne.
- WindowItems: keďže aplikácie na windowsoch sa skladajú z okien, budeme potrebovať držať referenciu na okno s ktorým práve pracujeme. A tieto typy okien vytvoríme vďaka tomuto usingu.

6.2.2 Globálne premenné

Táto časť nepatrí k najdôležitejším v teste, vždy sa tieto premenné dajú vložiť aj do iných častí testu, avšak kvôli prehľadnosti je vhodné mať vytvorené aj globálne premenné. Väčšinou ide o základné informácie, ako cesta k testovanej aplikácii, premenná pre základne okno aplikácie a podobne. Globálne premenné sa nachádzajú ešte pred inicializačnou časťou testu, v ktorej sa niektoré z týchto premenných môžu naplniť.

```
[TestClass]  
public class UnitTest1  
{  
    private const string src = @"C:\Users\Petusko\Desktop\Calculator\Calculator\bin\Debug\Calculator.exe";  
  
    private static Application _application;  
  
    private static Window _mainWindow;
```

Obr. 6.2: Globálne premenné

Obrázok 6.2 ukazuje príklad jednoduchých globálnych premenných, s ktorými sa bude ďalej v teste pracovať a bude potrebné, aby nezanikli počas vykonávania testu a je možné ich použiť aj v ďalšom teste.

- `src`: skratka pre source (zdroj) v tomto prípade je do premennej vložená cesta na ktorej sa nachádza spustiteľný `.exe` súbor pre testovanú aplikáciu
- `_application`: premenná, ktorá po inicializácii bude držať referenciu na testovanú aplikáciu a z ktorej dostaneme okno s ktorým sa bude ďalej pracovať
- `_mainWindow`: ide o premennú, ktorú tiež naplníme v inicializácii a bude držať referenciu na aktuálne okno s ktorým pracujeme

6.2.3 Inicializácia

Časť testu, ktorá je vykonávaná pred započatím každého testu v danom projekte. Obsahuje inicializáciu súčastí, ktoré bude test potrebovať na svoje správne vykonanie. Táto časť môže byť zahrnutá v tele samotného testu, avšak pokiaľ by sme mali viac ako jeden test, ktorý by vykonával rovnaké úkony na začiatku, napríklad spúšťanie tej istej aplikácie. Tak v tom momente by sa nám kód opakoval a preto je lepšie použiť inicializačnú sekciu kde daný kód je napísaný len raz. Tiež pomáha k sprehľadneniu a pokiaľ sa rozhodneme, že je potrebné na začiatku testu vykonávať ešte nejakú činnosť stačí ju pripísať sem, namiesto do každého testu.

```
[TestInitialize]
public void Initializce()
{
    _application = Application.Launch(src);
    _mainWindow = _application.GetWindow("Calculator");
}
```

Obr. 6.3: Inicializácia

- Metóda `Launch` (spustiť) zapína testovanú aplikáciu a potrebuje ako premennú cestu k tej aplikácii, ktorú sme si nastavili ako globálnu premennú `src`.
- Metóda `GetWindow` nám z našej spustenej aplikácie vráti otvorené okno. Táto metóda potrebuje názov okna, aby ho bolo možné identifikovať.

6.2.4 Telo testu pre windowsovú aplikáciu

Do samotného testu už píšeme potrebné veci súvisiace priamo s vykonávaním a kontrolovaním výstupov. Obsah tela testu sa podľa komplexnosti testov môže líšiť. Pokiaľ má aplikácia viac okien s ktorými pracujeme, tak v tele testu bude volanie metód preddefinovaných v triedach, vytvorených pre tieto okná. Takéto metódy, budú mať logiku naprogramovanú priamo v sebe, no pokiaľ ide o aplikáciu s jedným oknom, tak túto logiku môžeme uviesť aj do testu. V opačnom prípade je vhodné pre každé okno aplikácie vytvoriť vlastnú triedu. V tele sa nachádzajú aj asserty, ktoré vyhodnotia, či je test úspešný alebo nie.

```
[TestMethod]
public void TestMethod1()
{
    //inicializacia
    Button bt8 = _mainWindow.Get<Button>(SearchCriteria.ByAutomationId("number8"));
    Button bt2 = _mainWindow.Get<Button>(SearchCriteria.ByAutomationId("number2"));
    Button btPlus = _mainWindow.Get<Button>(SearchCriteria.ByAutomationId("buttonPlus"));
    Button btEquals = _mainWindow.Get<Button>(SearchCriteria.ByAutomationId("buttonEquals"));
    TextBox tbResult = _mainWindow.Get<TextBox>(SearchCriteria.ByAutomationId("textBox1"));
    //nastavenie hodnot
    bt8.Click();
    btPlus.Click();
    bt2.Click();
    btEquals.Click();
    //vyhodnotenie
    Assert.AreEqual("9", tbResult.Text, "Hodnoty sa nerovnajú");
}
```

Obr. 6.4: Telo testu

- Názov testu: v tomto prípade ide o názov štandardne vygenerovaný Visual Studiom. Pokiaľ tvoríme vlastné testy, je vhodné napísať názov, ktorý odzrkadľuje to, čo test bude kontrolovať.
- Metóda Get, ktorú môžeme vidieť na obrázku 6.4 vyhladá v danom okne typ premennej, teda button, textbox a podobne, podľa vyhladávacieho kritéria. V tomto prípade je to AutomationID.
- Button: ide o premennú typu tlačidlo, ktoré nájdeme podľa ID elementu. Takáto premenná, keď je priradená už obsahuje preddefinované metódy, ktoré môžeme používať, pri tlačidle je to hlavne metóda Click, ktorá spôsobí, že myška klikne na dané tlačidlo.
- TextBox: ide o premennú podobnú tlačidlu, avšak do tejto sa zapi-

suje text poprípade sa text z nej číta.

- Assert vyhodnotí, či sa zadané hodnoty v tomto prípade rovnajú a teda či test uspel alebo nie.

Ak ide o button alebo textBox, toto nie sú jediné konštrukcie, ktoré sa v aplikácii môžu vyskytovať, ďalšie sú napríklad: Combobox, DateForm, Table a mnoho ďalších.

6.2.5 Upratanie

Nepovinná časť, ktorá sa vykoná po ukončení každého testu. Pokiaľ testujeme aplikáciu, ktorú si na začiatku inicializujeme, tak pred spustením ďalšieho testu je potrebné všetko vrátiť do pôvodného stavu. K tomu slúži upratanie alebo Cleanup. Aj túto časť kódu, je možné vložiť do tela testu, avšak kvôli sprehľadneniu a zjednodušeniu práce s viacerými testami je výhodné vytvoriť metódu Cleanup.

```
[TestCleanup]
public void Cleanup()
{
    _mainWindow.Close();
}
```

Obr. 6.5: Upratanie

Pre prípad vyššie popisovaného testu je tu obsiahnuté len uzavretie a ukončenie aplikácie.

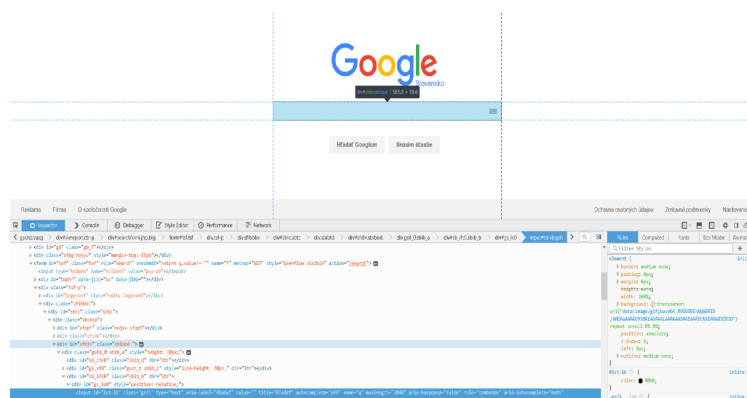
7 Tvorba testu pre webovú aplikáciu

Štruktúra testu a to, čo v sebe obsahuje, sa až tak moc nelíši od štruktúry a obsahu testu pre windowsovú aplikáciu. Preto môžeme ďalej vychádzať z informácií, ktoré sme si ukázali v kapitole číslo 6. Pri ukážke testovania webovej aplikácie sa pozrieme na tvorbu testov z dvoch pohľadov, a to imperatívny a neimperatívny prístup. K tomu využijeme dve technológie, ktoré nám ako white framework, uľahčia prácu s testovaním a ovládaním internetového prehliadača. Ide o Selenium a Canopy. Canopy zastupuje neimperatívny prístup a je postavené na Seleniu[20].

Pri testovaní windowsovej aplikácie sme využívali ďalší program UIVerify, aby sme dokázali lokalizovať jednotlivé elementy aplikácie. Pri webe, budeme potrebovať len webový prehliadač. Aký, na tom v podstate nezáleží. Selenium aj Canopy dokážu pracovať s najpoužívanejšími ako sú Internet explorer, Firefox a Chrome.

7.1 Hľadanie elementov vo webovej aplikácii

Ako je už spomínané vyššie, na túto časť nám stačí klasický webový prehliadač. Pokiaľ použijeme firefox, stačí kliknúť pravým tlačidlom myši a zvoliť možnosť preskúmať prvok (Inspect element). Následne sa nám otvorí inšpektor v ktorom si môžeme prezrieť elementy stránky a informácie ako ID, názov triedy a ďalšie, vďaka ktorým dokážeme pracovať v teste s týmito elementami. Pri prechode cez časti zobrazené v inšpektore, sa na stránke zvýraznia elementy, ku ktorým sa vzťahujú a teda sa dokážeme lepšie orientovať.



Obr. 7.1: Inšpektor

7.1.1 Obsah testu Selenium

V tejto časti si ukážeme, ako vytvoriť projekt a pripojiť k nemu Selenium a ako s ním dokážeme vytvoriť jednoduchý test.

7.1.1.1 Založenie testu s použitím Selenia

Zo začiatku postupujeme presne rovnako, ako pri tvorbe testu pre windowsovú aplikáciu, avšak keď sa dostaneme k Package Manager console, je potrebné použiť Install-Package Selenium.Support. Takto získame závislosti a môžeme začať pracovať so Seleniumom. Na začiatku je potrebné vedieť, že pre prácu s rôznymi prehliadačmi sú potrebné špecifické ovládače pre tieto prehliadače. Pre firefox nám stačí GeckoDriver, ktorý sprostredkuje komunikáciu medzi seleniumom a firefoxom. Dostupný je na stránke <https://github.com/mozilla/geckodriver/releases>. Po stiahnutí stačí k nemu v Enviroment variables nastaviť cestu a firefox je pripravený na spustenie. GeckoDriver je nutný pre fungovanie s novšími verziami firefoxu.

7.1.1.2 Usingy a inicializácia

Pokiaľ ide i o usingy a inicializáciu testu, priebeh je rovnaký, ako pri windowsovej aplikácii, avšak líšia sa príkazy a využívajú sa iné usingy.

```
using System.Collections.Generic;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using OpenQA.Selenium;
using OpenQA.Selenium.Firefox;

namespace Selenium
{
    [TestClass]
    public class UnitTest1
    {
        FirefoxDriver firefox;

        [TestInitialize]
        public void Initialize()
        {
            firefox = new FirefoxDriver();
            firefox.Navigate().GoToUrl("http://www.google.sk/");
        }
    }
}
```

Obr. 7.2: Úvod webového testu

Usingy nám znovu ukazujú, čo budeme používať pri tomto testovaní.

- Selenium: poskytuje konštrukcie, pre prácu s webovými elementami ako napríklad `IWebElement`.
- Selenium.Firefox: umožňuje použitie prehliadača firefox, pokiaľ by sme chceli použiť Internet Explorer treba pridať using, ktorý by končil `Selenium.IE`.

V globálnej premennej je uchovaná premenná `len` pre prácu s prehliadačom, ako vidíme na rozdiel od windowsovej aplikácie nepotrebujeme ďalšiu premennú na jednotlivé okná.

Pokiaľ ide o inicializáciu, v nej sa môže nachádzať príkaz na spustenie prehliadača, poprípade aj načítanie úvodnej testovacej stránky.

7.1.1.3 Telo testu

Do tela testu znova vkladáme priebeh testu. Teda čo je potrebné otvoriť, kde treba kliknúť a na záver samozrejme kontrola ktorá vyhodnotí výsledok testu.

```
[TestMethod]
public void TestMethod1()
{
    IWebElement tbSearch = firefox.FindElement(By.Id("1st-ib"));
    IWebElement btSearch = firefox.FindElement(By.Id("sblsbb"));
    tbSearch.SendKeys("pes");
    btSearch.Click();
    IList<IWebElement> allElements = firefox.FindElements(By.ClassName("r"));
    Assert.IsTrue(allElements[0].Text.Contains("Pes"));
}
```

Obr. 7.3: Telo testu pre webovú aplikáciu

Namiesto konštrukcií ako boli pri windowsovej aplikácii, napríklad button, textbox sa pri testovaní webu nachádza multifunkčný typ `IWebElement`, do ktorého môžeme vložiť tlačidlá, odkazy a mnoho ďalšieho. Nie je to teda jednoznačne určené, ako to bolo pri windowse. Pri tvorbe testu pre webovú aplikáciu treba dávať pozor na to, že stránky nie sú načítane hneď, preto mnohokrát treba využiť čakanie, avšak vždy je lepšie použiť metódu ktorá vyvolá čakanie než sa nejaký element zobrazí, a vtedy vieme, že môžeme pokračovať v teste, než natvrdo zaviesť hodnoty ako dlho sa má čakať.

- **IWebElement**: drží nám referenciu na vyhl'adaný element, ktorý je možné vyhľadať podľa rôznych kritérií ako ID, class, cssSelector, name a podobne. Tieto štruktúry majú tiež preddefinované funkcie ako click, send keys atď.
- **IList**: je kolekcia elementov, ktoré majú určitý spoločný prvok ako class, name a podobne. V takejto kolekcii sa dá následne vyhľadať, či sa tam nachádza čo sme očakávali.

Pokiaľ ide o ukončenie testu, znova môžeme použiť metódu **CleanUp** ako aj pri windowsovej aplikácii a ukončiť webový prehliadač, a teda uviesť testovacie prostredie do pôvodného stavu[21].

7.1.2 Obsah testu Canopy

Ide o neimperatívny prístup k programovaniu, čo nám už hovorí, že sa nebude používať programovací jazyk **c#** ale náhrada a to **f#**. Visual studio podporuje tvorbu testov v tomto jazyku. Ďalej si prejdeme, ako si pripraviť nový projekt práve s Canopy a jednoduchý test s použitím Canopy.

7.1.2.1 Založenie testu s použitím Canopy

Pri tvorbe nového projektu je potrebné dávať pozor na to, aký projekt voľíme. Keďže už nejde o **c#**, treba zvoliť projekt pre **f#** a vybrať, že pôjde o konzolovú aplikáciu. V novo-vytvorenom projekte môžeme obsah zmazať, pretože nebude potrebný. Teraz je potrebné implementovať Canopy, na to použijeme znova Package Manager console, do ktorej tentokrát napíšeme **Install-Package canopy**. Počas pripájania jednotlivých závislostí si môžeme všimnúť, že sa pridalo aj Selenium, to je preto, že Canopy, je vystavané na Seleniumu. Po pridaní Canopy do projektu sme pripravení začať písať test. Dôležité je uvedomiť si, že na rozdiel od doterajších testov, teraz pracujeme s konzolovou aplikáciou.

7.1.2.2 Inicializácia testu

Ako sme boli zvyknutý pri **c#** používať **usingy**, pri **f#** a Canopy sa **usingy** zmenili na **open**. Význam je v podstate rovnaký, ide len o iný zápis toho istého. Pokiaľ by sme sa snažili použiť **using**, nepodarí sa nám skompilovať test.

```

open canopy
open runner
open System

start firefox

```

Obr. 7.4: Inicializácia Canopy

- Open: rovnaká funkcia ako using pri c#
- Start: metóda, ktorá otvorí definovaný prehliadač, tak ako pri sele-
niu je potrebné mať GeckoDriver nastavený v Enviroment variab-
les, inak najnovší firefox nebude fungovať

Samozrejme aj tu môžeme použiť metódy, ktoré sa spustia pred začatím každého testu a po skončení testu, ide o metódy before a after. Ich funkcia je totožná s Initialize a CleanUp ktoré poznáme zo c#.

7.1.2.3 Telo testu

Jednotlivé príkazy sa podstatne líšia od Selenia tak isto aj samotné asserty.

```

"taking canopy for a spin" &&& fun _ ->

    url "http://www.google.sk"

    "#lst-ib" << "pes"

    click "#sblsbb"

    on "https://www.google.sk/?gws_rd=ssl#q=pes"

    read("#lst-ib") ≠ "pes"

//run all tests
run()

printfn "press [enter] to exit"
System.Console.ReadLine() |> ignore

quit()

```

Obr. 7.5: Telo testu canopy

Prvá vec, ktorú si môžeme všimnúť je označenie testu. Pozostáva z názvu testu. Tu je vhodné opísať, čo test bude vykonávať, aby sme sa pri

vyhodnotení lepšie zorientovali. Za názvom sa nachádza značka `&&&`, ide o označenie typu testu. Konkrétne toto označenie hovorí, že test je hotový a pripravený na spustenie. Test je možné zaznačiť aj značkou `&&&&`, táto má vyššiu prioritu a označuje test, ktorý je vo vývoji. Pokiaľ máme v projekte test označený ako `&&&&` a `&&&&` a spustíme testy, spustia sa len testy označené `&&&&`. Tento test sa vykonáva pomalšie a počas jeho priebehu sú vývojárovi na prehliadači označované elementy s ktorými práve pracuje[22].

- **Url:** metóda, ktorá berie cestu ktorú načíta v otvorenom webovom prehliadači.
- **On:** typ assertu ktorý skontroluje, či v danom momente je test na danej stránke.
- **Znak #asdf:** označuje, že hľadáme element, ktorého ID je asdf, na prehľadávanie pomocou akýchkoľvek typov slúžia `cssSelector`[23].
- **Read:** metóda ktorá prečíta text, ktorý sa nachádza v elemente označenom daným `cssSelectorom`.
- **Run:** spustí vykonávanie testov.
- **Quit:** ukončí vykonávanie testov.

Pokiaľ ide o zvyšné dva nespomenuté príkazy `printf` a `System.Console.ReadLine`, ide o príkazy, ktoré nám na konzole ukážu výstup ako test skončil, a umožnia stlačením akejkoľvek klávesy ukončiť túto konzolovú aplikáciu.

8 Záver

Cieľom práce bolo vytvoriť učebný materiál, ktorý slúži na oboznámenie čitateľa s oblasťou testovania a pomôže mu pri tvorbe prvých testov. Začiatok práce je venovaný metódam tvorby softvéru a jednotlivým typom testovania, počas rôznych fáz vývoja. Ďalej ukazujem, ako sa testovanie delí a v čom sa jednotlivé typy líšia a na čo treba dávať pozor. Súčasťou jednotlivých typov testov sú aj popisy chýb, aké sa často objavujú a na čo pri testovaní dávať pozor. Po rozdelení testov nasleduje ukážka tvorby automatických testov s popisom ako čo najlepšie navrhnúť a vytvoriť test.

Po prečítaní tejto práce je študent schopný rozlišovať rôzne typy testov. Je oboznámený so základnými prvkami testovania. Vie ako pristupovať k tvorbe testovacích scenárov a na čo si dať pozor aby jeho scenáre pochopil aj iný tester. Je schopný vytvoriť základne automatické testy pre web a windowsové aplikácie.

K práci sú pridané videá, ktoré sa zameriavajú na jednotlivé časti ako sú: tvorba scenáru, čo kontrolovať pri testovaní a tvorba automatických testov.

Literatúra

- [1] Tutorialspoint-Waterfall [online]. [cit. 2015-20-12]. Dostupné z:http://www.tutorialspoint.com/sdlc/sdlc_waterfall_model.htm
- [2] Tutorialspoint-Prototype [online]. [cit. 2015-20-12]. Dostupné z:http://www.tutorialspoint.com/sdlc/sdlc_software_prototyping.htm
- [3] Modely životného cyklu [online]. [cit. 2016-07-01]. Dostupné z:<http://www2.fiit.stuba.sk/~bielik/courses/psi-slov/slajdy/prototypovanie.pdf>
- [4] Tutorialspoint-Agile [online]. [cit. 2015-20-12]. Dostupné z:http://www.tutorialspoint.com/sdlc/sdlc_agile_model.htm
- [5] Software testing mentor [online]. [cit. 2015-20-12]. Dostupné z:<http://www.softwaretestingmentor.com/introduction/what-is-testing/>
- [6] PATTON, Ron. Software Testing. 2. vyd. USA: SAMS publisher, 2006, ISBN-13: 978-0-672-32798-8
- [7] OneNote [online]. [cit. 2017-08-01]. Dostupné z:<https://www.onenote.com/>
- [8] Asana [online]. [cit. 2017-08-01]. Dostupné z:<https://www.asana.com/>
- [9] JIRA [online]. [cit. 2017-08-01]. Dostupné z:<http://www.myjira.cz/index.html>
- [10] Bugzilla [online]. [cit. 2017-08-01]. Dostupné z:<https://www.bugzilla.org/>
- [11] JORGENSEN, Paul. Software testing a craftsmans Approach. 4. vyd. USA: CRC Press, 2014, ISBN: 978-1-4665-6068-0
- [12] Unit Tests, How to Write Testable Code [online]. [cit. 2016-08-01]. Dostupné z:<https://www.toptal.com/qa/how-to-write-testable-code-and-why-it-matters>

-
- [13] Code project: dependency injection [online]. [cit. 2015-20-12]. Dostupné z:<http://www.codeproject.com/Articles/615139/An-Absolute-Beginners-Tutorial-on-Dependency-Inver>
- [14] Mocks Aren't Stubs [online]. [cit. 2015-20-12]. Dostupné z:<http://martinfowler.com/articles/mocksArentStubs.html>
- [15] Moq knižnica [online]. [cit. 2015-20-12]. Dostupné z:<http://www.developerhandbook.com/unit-testing/writing-unit-tests-with-nunit-and-moq/>
- [16] White framework [online]. [cit. 2015-20-12]. Dostupné z:<https://github.com/TestStack/White>
- [17] Obrázok interakcie white framework-u s aplikáciou [online]. [cit. 2015-20-12]. Dostupné z:<http://www.codeproject.com/Articles/289028/White-An-UI-Automation-tool-for-windows-applicatio>
- [18] UIVerify [online]. [cit. 2015-20-12]. Dostupné z:<https://msdn.microsoft.com/en-us/library/jj160544%28v=vs.85%29.aspx>
- [19] WhiteStack Started [online]. [cit. 2017-08-01]. Dostupné z:<http://white.teststack.net/docs/getting-started>
- [20] Automation for web Applications [online]. [cit. 2017-08-01]. Dostupné z:http://www.seleniumhq.org/docs/01_introducing_selenium.jsp#test-automation-for-web-applications
- [21] Selenium [online]. [cit. 2015-20-12]. Dostupné z:http://docs.seleniumhq.org/docs/03_webdriver.jsp#the-5-minute-getting-started-guide
- [22] Canopy [online]. [cit. 2015-20-12]. Dostupné z:<https://lefthandedgoat.github.io/canopy/index.html>
- [23] CssSelectors [online]. [cit. 2015-20-12]. Dostupné z:http://www.w3schools.com/cssref/css_selectors.asp

9 Prílohy

Súčasťou práce sú aj nasledujúce prílohy:

- Náučné videá popisujúce získané informácie
- Demonštračné aplikácie na tvorbu automatických testov