Framework FLAUI a tvorba testovacích případů pro automatizované testování

Adam Michálek

Bakalářská práce
2020

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
Ústav informatiky a umělé inteligence

Akademický rok: 2019/2020

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Adam Michálek**
Osobní číslo:	**A17135**
Studijní program:	**B3902 Inženýrská informatika**
Studijní obor:	**Softwarové inženýrství**
Forma studia:	**Prezenční**
Téma práce:	**Framework FLAUI a tvorba testovacích případů pro automatizované testování**
Téma práce anglicky:	**The FLAUI Framework and the Trial Automation of Test Cases**

Zásady pro vypracování

1. Nastudujte a popište potřebnou technologii ve vztahu automatizovaného testování a využití frameworku FLAUI.
2. Zvolte vhodné prostředky a metody pro realizaci testů pomocí frameworku FLAUI.
3. Navrhněte a implementujte řešení pro automatizované testování pomocí frameworku FLAUI.
4. Vytvořte systém reportováni s praktickým nasazením na server.
5. Zvolte si aplikaci a řešení na ní otestujte.
6. Vhodně vyhodnoťte výsledky a rozeberte možnosti využití.

Rozsah bakalářské práce:
Rozsah příloh:
Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam doporučené literatury:

1. STEPHENS, Matt a Doug ROSENBERG. Testování softwaru řízené návrhem. Brno: Computer Press, 2011, 336 s. ISBN 9788025136072.
2. PATTON, Ron. Testování softwaru. Praha: Computer Press, 2002, xiv, 313 s. Programování. ISBN 8072266365.
3. HEROUT, Pavel. Testování pro programátory. České Budějovice: Kopp, 2016, 405 s. ISBN 9788072324811.
4. ROUDENSKÝ, Petr a Anna HAVLÍČKOVÁ. Řízení kvality softwaru: průvodce testováním. Brno: Computer Press, 2013, 208 s. ISBN 9788025138168.
5. PAGE, Alan, Ken JOHNSTON a Bj ROLLISON. Jak testuje software Microsoft. Brno: Computer Press, 2009, 384 s. ISBN 9788025128695.

Vedoucí bakalářské práce: **Ing. Petr Žáček**
Ústav informatiky a umělé inteligence

Datum zadání bakalářské práce: **28. listopadu 2019**
Termín odevzdání bakalářské práce: **15. května 2020**

doc. Mgr. Milan Adámek, Ph.D.
děkan

prof. Mgr. Roman Jašek, Ph.D.
ředitel ústavu

Ve Zlíně dne 9. prosince 2019

Prohlašuji, že

- beru na vědomí, že odevzdáním bakalářské práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že bakalářská práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně a jeden výtisk bude uložen u vedoucího práce;
- byl/a jsem seznámen/a s tím, že na moji bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – diplomovou/bakalářskou práci nebo poskytnout licenci k jejímu využití jen připouští-li tak licenční smlouva uzavřená mezi mnou a Univerzitou Tomáše Bati ve Zlíně s tím, že vyrovnání případného přiměřeného příspěvku na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše) bude rovněž předmětem této licenční smlouvy;
- beru na vědomí, že pokud bylo k vypracování bakalářské práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky bakalářské práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem bakalářské práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na bakalářské práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze bakalářské práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně, dne

Adam Michálek, v. r.
podpis diplomanta

ABSTRAKT

Cílem práce bylo navrhnout rozhraní, které zjednoduší proces vytváření UI testů, sníží duplicitu kódu a zvýší obecnost při jejich vytváření. Toto rozhraní je primárně určené pro Window form a WPF aplikace, ale je možné jeho použití v dalších technologiích, jako jsou webové stránky, nebo mobilní zařízení. Rozhraní je založeno na frameworku FlaUI, což je hlavní technologie, která je v této práci využita a na jejím základě je zkonstruováno již zmíněné rozhraní. Dále byl vytvořen funkční systém reportování výsledků testů, kde bylo použito mnoho dalších technologií. V práci je popsán význam technologií jako Nunit, Nunit console, XSLT a mnoho dalších. V neposlední řadě je v práci vytvořeno několik testovacích případů a ty jsou přepsány pomocí tohoto rozhraní do kódu a poté nasazeny na webový server. Teoretická část se věnuje problematice testování obecně, poté se zaměří na návrhové vzory a čistý kód, ze kterých se při programování vycházelo. Nakonec je shrnuta problematika FlaUI frameworku, jeho výhody, nevýhody a další informace.

Klíčová slova: UI testy, Window form, WPF, FlaUI, Nunit, Nunit konsole, XSLT, testovací případ, webový server, čistý kód

ABSTRACT

The aim of this work was to design an interface that simplifies the process of creating UI tests, decreases code duplicity and increases generality in the creation. This interface is primarily designed for Window form and WPF applications, but can be used in other technologies such as web sites or mobile devices. The interface is based on the FLAUI framework, which is the main technology that is used in this work and based on it, the interface is constructed. Furthermore, a functional system of reporting test results was created, where many other technologies were used. The thesis describes the technology challenges such as Nunit, Nunit console, XSLT and many others. Last but not least, there are several test cases created in the thesis and these are rewritten into the code using this interface and then deployed to the web server. The theoretical part deals with testing in general, then it focuses on design patterns and pure code, which was used in programming. Finally, the FlaUI framework, its advantages, disadvantages and other information are summarized.

Keywords: UI tests, Window Forms, WPF, FlaUI, Nunit, Nunit Console, XSLT, test case, web server, clean code

Poděkování

Tímto bych chtěl poděkovat vedoucímu své bakalářské práce Ing. Petru Žáčkovi a dalším lidem za jejich odbornou pomoc a cenné rady při vytváření této práce.

Prohlašuji, že odevzdaná verze bakalářské/diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

OBSAH

ÚVOD ... **11**
I TEORETICKÁ ČÁST ... **13**
1 TESTOVÁNÍ SOFTWARU ... **14**
1.1 NEJZNÁMĚJŠÍ CHYBY ... 14
1.1.1 První bug ... 14
1.1.2 Problém roku 2000 ... 15
1.1.3 Střela patriot ... 15
1.2 ZÁKLADNÍ POJMY PŘI TESTOVÁNÍ ... 16
1.2.1 Základní pojmy a definice ... 16
1.2.2 Třídy ekvivalence a analýza hraničních hodnot ... 17
1.2.3 Testování černé a bílé skříňky ... 18
1.2.4 Příčiny chyb ... 19
1.2.5 Důsledky chyb ... 20
1.3 ŽIVOTNÍ CYKLUS VÝVOJE SOFTWARU ... 21
1.3.1 Základní modely a proč je používáme ... 21
1.3.2 Vodopádový model ... 22
1.3.3 Spirálový model ... 22
1.4 MANUÁLNÍ TESTOVÁNÍ ... 23
1.4.1 Druhy testování ... 23
1.4.2 Testovací sady a testovací případy ... 24
1.5 AUTOMATIZOVANÉ TESTOVÁNÍ ... 25
1.5.1 Výhody a nevýhody ... 25
1.5.2 Typické případy použití ... 26
1.5.3 Používané technologie ... 26
2 NÁVRHOVÉ VZORY A ČISTÝ KÓD ... **28**
2.1 OBJEKTOVĚ ORIENTOVANÉ PROGRAMOVÁNÍ ... 28
2.1.1 Základní pojmy ... 28
2.1.2 Základní principy ... 29
2.1.3 Zásady při návrhu programu ... 29
2.2 ČISTÝ KÓD ... 32
2.2.1 Proč udržovat kód čistý ... 32
2.2.2 Základní pravidla čistého kódu ... 33
2.3 POUŽITÉ NÁVRHOVÉ VZORY ... 35
2.3.1 Proč používat návrhové vzory ... 35
2.3.2 Dependency injection ... 35
2.3.3 Singleton ... 36
2.3.4 Factory ... 36
3 FLAUI FRAMEWORK ... **38**
3.1 PROČ PRÁVĚ FLAUI ... 38
3.2 TEORETICKÝ ÚVOD ... 38
3.2.1 Co je FlaUI ... 38
3.2.2 Předloha a historie ... 39
3.2.3 Výhody a nevýhody FlaUI ... 39

3.3 PROGRAMOVÁ STRUKTURA FLAUI 40
3.3.1 UIA2 a UIA3 40
3.3.2 Core 40
3.3.3 UITests a TestApplication 41
3.4 POUŽÍVÁNÍ FLAUI 41
3.4.1 Základní komponenty 41
3.4.1.1 Aplikace 41
3.4.1.2 Hledání prvků a operace s nimi 41
3.4.1.3 Zachycování výsledků (capturing) 42
3.4.1.4 Opakování (retry) 43
3.4.1.5 Další 43
3.4.2 Potřebný software 43
3.4.3 Problémy s FlaUI 44
II PRAKTICKÁ ČÁST 45
4 NÁVRH ARCHITEKTURY A PLÁNOVÁNÍ 46
4.1 ZÁKLADNÍ INFORMACE 46
4.2 POTŘEBNÉ NÁSTROJE A TECHNOLOGIE 47
4.2.1 Jazyky 47
4.2.2 IDE a SW 48
4.2.3 Knihovny a frameworky 48
4.3 VÝSLEDNÝ NÁVRH 49
5 FLAUI V PRAXI 51
5.1 VOLBA SPRÁVNÉ VERZE 51
5.2 HLEDÁNÍ IDENTIFIKÁTORŮ 51
5.3 PROGRAMOVÁNÍ VE FLAUI 52
5.3.1 Zachycení okna aplikace a operace s ním 52
5.3.2 Efektivní hledání UI prvků 54
5.3.3 Operace s UI prvky 58
5.3.4 Třída Retry 59
5.3.5 Klávesy a reporting 59
6 KNIHOVNA FWHANDLER A JEJÍ SOUČÁSTI 61
6.1 HANDLER PROVIDER 61
6.2 UICONTROLLER, UIHANDLER 62
6.3 CONTROLLER 64
6.4 SYSTÉM VÝJIMEK A POMOCNÝCH VÝČTŮ 64
6.5 POMOCNÉ KNIHOVNY 65
7 SYSTÉM REPORTOVÁNÍ 67
7.1 NEZBYTNÉ KOMPONENTY A JEJICH KOOPERACE 67
7.2 XMLTRANSFORMER 68
7.3 XMLCONTROLLER 69
7.4 XSLT PROGRAMOVÁNÍ 69
7.5 VÝSLEDNÝ REPORT 71
8 NÁVRH TESTŮ PRO APLIKACI 72

8.1 NÁVRH TC PRO AUTOMATIZOVANÉ TESTOVÁNÍ 72
8.2 PŘÍPRAVA PROSTŘEDÍ PRO TESTOVÁNÍ 77
8.3 PŘEPIS JEDNOTLIVÝCH TC DO KÓDU POMOCÍ FWHANDLERU A NUNIT 79
9 NASAZENÍ TESTŮ NA SERVER 84
9.1 SERVER V PRAXI 84
9.2 ŘEŠENÍ SERVERU PRO TUTO PRÁCI 84
10 DOSAŽENÉ VÝSLEDKY A MOŽNOSTI VYUŽITÍ 88
ZÁVĚR 92
SEZNAM POUŽITÉ LITERATURY 94
SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK 97
SEZNAM OBRÁZKŮ 98
SEZNAM PŘÍLOH 100

ÚVOD

V současné době se velmi zvyšuje komplexita vytvářeného softwaru. Ať už se jedná o klasické desktopové aplikace, mobilní aplikace, webové stránky, embedded systémy, všechny sdílí jednu a tu samou potřebu a to kvalitně a důsledně otestovat danou implementaci. Důsledky nekvalitně otestovaného softwaru jsou od těch zanedbatelných, tedy ztráta peněz a času, až po ty katastrofické, jako je ztráta na životech. Tyto katastrofické scénáře jsou mnohdy umocněny dobou, ve které žijeme. Ta je charakteristická používáním výpočetní techniky ve všech oborech lidské činnosti, od zdravotnictví, po automobilový průmysl, kosmický průmysl a mnoho, mnoho dalších. Nastává tedy otázka, jak správně testovat software, abychom předešli většině nežádoucích stavů, které mohou nastat.

Mezi základní dva způsoby, jak software můžeme testovat, patří manuální a automatizované testování. Manuální testování je velmi důležité a v nejbližší době nemůže být automatizovaným zcela nahrazeno. Testuje nejen funkcionalitu aplikace, ale i uživatelskou přívětivost. Automatizované testování naopak dokáže efektivně testovat procedury, které se opakují a nevyžadují tolik lidské interakce. Tento druh testování je také náplní této práce.

Pro automatizované testy je vhodný framework (popřípadě software), který umožní efektivní, stabilní a jednoduché vytváření testů. Pro některé technologie jsou tyto frameworky velmi rozšířené, například webové stránky (robot-framework a další založené na seleniu). Problémem jsou technologie, na které požadovaný framework chybí, nebo je zastaralý, případně nefunkční. Absence takového frameworku, který je jednoduchý na použití a dokáže v něm operovat i člověk bez pokročilých znalostí programování, je důvodem vzniku této práce. Jedná se o desktopové aplikace na operačním systému Windows, vytvořené například frameworkem WPF, nebo Windows Form. Existuje několik placených programů, které si kladou za cíl zjednodušit uživateli vytváření automatizovaných testů "naklikáním". Tyto programy bývají drahé, proto se komunita snaží vytvořit vlastní nástroje pro konkrétní framework. Mezi ně patří například framework White a hlavně jeho nástupce FlaUI, který jej v mnoha ohledech předčil. I ten má však své nedostatky, hlavně co se týče obecnosti pro jednotlivá řešení a duplicity kódu. Tyto nedostatky se pokouší eliminovat vytvořená knihovna FWHandler.

První kapitola teoretické části (č.1) této práce je věnována testování jako celku, příblížíme si jednotlivé koncepty a ukážeme si, jak do nich zapadá automatizované testování a FlaUI. Další kapitola (č.2) je věnována nejen návrhovým vzorům a čistému kódu, ale také obecně programování, protože vytváření velké částí této práce vyžadovalo poměrně rozsáhlé znalosti v této oblasti. V poslední kapitole (č.3) teoretické části je rozebrán FlaUI framework, hlavně jeho výhody, důvod výběru, jednotlivé komponenty a podobně.

V první kapitole praktické části (č.4) je navržena architektura FWHandleru včetně podružných komponent. Čtenář se dozví, jak do sebe tyto součásti zapadají a jaký je jejich účel. Jedná se o vstupní bod, ze kterého poté můžeme přejít do kterékoliv části práce. Další kapitola (č.5) se věnuje využití FlaUI v této BP. Obsahuje praktické ukázky a zdůvodňuje, proč je vše implementováno tak, jak je. Následující kapitola (č.6) se věnuje samotné knihovně FWHandler a její implementaci. Velmi důležitou součástí projektu je systém reportování, který je popsán v kapitole č.7. Je zde představena jeho kompletní implementace, včetně vizuálních ukázek výsledného reportu. V další kapitole (č.8) je vybrána aplikace Notepad, pro kterou je vytvořeno několik testovacích případů. Ty poté pomocí vytvořených nástrojů (FWHandler knihovna a Nunit) zautomatizujeme. Kapitola č.8 dále obsahuje několik zajímavých typů a rad, jak by se s knihovnou mělo pracovat a jaké jsou její možnosti. Předposlední kapitola (č.9) se věnuje nasazení vytvořených testů na webový server, čímž zautomatizujeme celý proces a minimalizujeme negativní vliv lidského faktoru. Kapitola je zakončena spuštěním všech testů na serveru a vyhodnocením výsledků. V poslední kapitole (č.10) jsou sumarizovány výsledky této práce.

I. TEORETICKÁ ČÁST

1 TESTOVÁNÍ SOFTWARU

Testování softwaru je nedílnou součástí každého vývojového procesu. Co je jeho cílem nejlépe vystihuje následující věta napsaná Pattonem: “Cílem softwarového testera je vyhledávat chyby, vyhledat je co nejdříve a zajistit jejich nápravu” [1].

Existence chyb je přirozenou součástí při vývoji softwaru. Člověk není tak dokonalý, aby mohl pojmout všechny stavy, které při vývoji mohou nastat. Ignorování existence chyb, popřípadě nedůslednost jejich odchycení při vývoji, může vést k velmi nepříjemným situacím od finančních škod, až po ohrožení života, nebo dokonce smrti [1].

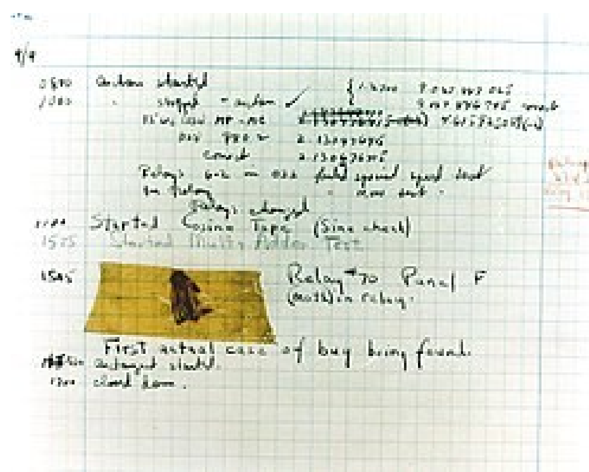
Co všechno může způsobit nedostatečné testování je ukázáno v následující kapitole č. 1.1

1.1 Nejznámější chyby

Uvedené chyby by měly pokrývat obě spektra, od zanedbatelných, po extrémní.

1.1.1 První bug

Význam počítačového bugu je pro většinu programátorů a testerů velmi známý. Jedná se o chybu, respektive nedostatek v softwarovém programu, který způsobuje neočekávané chování a nesprávné výsledky na výstupu [2].



Obr. 1. První bug [16]

Jaký je původ tohoto slova? Ještě před použitím slova bug v počítačové technice bylo oblíbené označovat tímto výrazem nějakou nedokonalost v mechanických systémech. Další teorie jdou dále do historie a tvrdí, že slovo vychází ze slova Bugge, což je základ pro slovo bubák. V počítačové technice je slovo bug připisováno k reálné příhodě, kdy počítač z 50 let, Mark 2, zničehonic přestal fungovat. Po prozkoumání každého relé v počítači se

zjistilo, že na jednom je můra. Ta způsobila zkrat a porouchala celý počítač. Od té doby se počítačová chyba označuje jako bug [3, 4].

1.1.2 Problém roku 2000

Důvodem vzniku tohoto bugu je fakt, že v historii kvůli nedostatku paměti bylo potřeba ušetřit každý byte. Proto se místo čtyř místného roku (který vyžaduje 4 bajty), začaly používat pouze poslední dvě čísla. To vedlo k nutnosti provést velké množství aktualizací a oprav s celkovými náklady sto miliard korun. Kromě této sumy se mluvilo i o katastrofičtějších scénářích, jako například výpadcích elektráren, pozemní i letecké havárie apod.. Nic z toho se naštěstí nenaplnilo. Kvůli velkému množství paměti v současné technice je nepravděpodobné, že by se problém opakoval [1, 5].

1.1.3 Střela patriot

Posledním připadem je chyba, která způsobila smrt 28 amerických vojáků. Chyba byla na straně obraného antibalistického systému, jehož úkolem je detekovat a sestřelit nepřátelské rakety. Tento systém byl vytvořen pro dlouhodobý běh a ve stavu, v jakém se nacházel, běžel už sto hodin. Po každé hodině se vnitřní hodiny zpozdily o pár milisekund, což mělo při odpálení obrovský defekt na systém. Vypálená raketa měla zpoždění třetinu sekundy, což stačilo na minutí cíle [3].

1.2 Základní pojmy při testování

V každé oblasti je nutné definovat základní pojmy, aby nedocházelo k nedorozumění a nejasnostem. Při vytváření a testování SW to platí dvojnásob. Je nezbytné si odpovědět na otázky typu co je to chyba, jak ji definujeme a jak s ní pracujeme, jaké jsou příčiny chyb a důsledky z nich vyplývající [1].

1.2.1 Základní pojmy a definice

O **chybě** mluvíme, když není splněna některá z následujících podmínek. Software by se měl chovat podle specifikace produktu (či očekávání uživatele). Neměl by dělat cokoliv navíc, nebo něco, co ve specifikaci vůbec není. Dalším případem je situace, kdy funkcionalita chybí přímo ve specifikaci softwaru a měla by zde být. Tyto pravidla jsou velmi přímočaré a neměly by být obcházena. Existují však méně formální faktory. Bude s aplikací zákazník spokojený? Ovládá se dobře? Není pomalá? Tyto a další otázky je potřeba si během procesu klást a zodpovědět na ně, v případě záporných odpovědí vyvodit důsledky, o kterých budeme mluvit později [1].

Dále je nutné rozlišovat pojem **omyl** a **selhání**. Omylem můžeme nazvat nekorektní akci uživatele, která zapříčiní chybu v programu. Selháním můžeme označit chyby v kódu, které můžou vyústit v jedno, nebo více selhání [1].

Testování lze rozdělit do několika kategorií. Patří sem statické a dynamické testování.

Statické testování je založeno na specifikaci a dokumentaci k danému softwaru. Jedná se o efektivní způsob odhalení chyb před začátkem vývoje, který vede k ušetření peněz, protože jak se dozvíme dále, čím dříve chybu odchytíme, tím levnější je její náprava [6].

Dynamické testování vyžaduje spustitelný software, který můžeme ovládat pomocí zadávání vstupů a ověřování výstupů [6].

Dalším rozdělením je testování metodou ***černé*** a ***bílé skříňky***. Zde je rozdíl ve způsobu, jakým daný software testujeme. O nich bude řeč v dalších kapitolách.

Manuální testy vymýšlí, provádí a ověřuje člověk. Jedině člověk dokáže (zatím) vyhodnotit, jestli se daná aplikace dobře ovládá, jestli je ovládání intuitivní a splňuje všechny požadavky [1].

Automatizované testy jsou vhodné pro rutinní záležitosti a téměř nevyžadují lidský faktor. Pro automatizované testy je nutné splnit několik předpokladů [1].

Více o manuálním a automatizovaném testování dále (kapitola č. 1.4 a 1.5)

Dalším rozdělením jsou testy splnění a testy selhání.

Testy splnění kontrolují minimální funkčnost daného softwaru, tedy nesnaží se nacházet určité kritické stavy, ale ověřujeme pouze základní funkčnost aplikace, tu, která je zamýšlena při běžném používání [1].

Testy selhání oproti tomu hledají určité hraniční stavy aplikace, se kterými programátor nemusel počítat [1].

Klasickým příkladem je zadání čísla do textboxu, který vyžaduje pouze písmena. Dalším dobrým příkladem je situace, kdy uživatel zadává například věk. Programátor počítal se zadáním kladného celého čísla do 120 let, uživatel omylem zadal záporné číslo, popřípadě pole zapomněl vyplnit. Všechny tyto stavy by měly pokrýt třídy ekvivalence.

1.2.2 Třídy ekvivalence a analýza hraničních hodnot

Dva velice důležité pojmy, se kterými se u testování setkáme jsou třída ekvivalence a hraniční podmínky.

Představme si aplikaci, která má miliardy možných vstupů, a ještě více možných výstupů. Jedná se o kalkulačku, popřípadě jiný komplexní software. Není v lidských silách otestovat každý vstup a výstup, který může nastat. Můžeme však všechny relevantní vstupy, které spadají do stejné skupiny, rozdělit do základních tříd a tím zúžit počet vstupů, které je potřeba testovat. Je například potřeba testovat 1+4, když víme že 1+1 funguje? Druhým příkladem je 1 + 1111111111111111. V tomto případě se už jedná o extrémní číslo a mohlo by zde dojít k přetečení. Programátor by například mohl počítat s tím, že uživatel zadá integer, ne long [8, 9].

Hraniční podmínky můžeme rozdělit jako jednotlivé intervaly a zapsat je do tabulky. Způsobů jakým je zpracovat je mnoho, zde je jeden z nich pro test textboxu pro zadání roku:

Třída ekvivalence	Očekávaný výsledek	Testovaná hodnota
Rok < 0	Nepřijato	-5
Rok == 0	Nepřijato	0
1 >= Rok <= 2020	Přijato	2020
Rok == 255	Přijato	255
Rok == 256	Přijato	256
Rok >= 10000	Nepřijato	120000

Obr. 2. Tabulka tříd ekvivalence

Na obrázku č.2 vidíme zjednodušený model třídy ekvivalence pro aplikaci přijímající jako vstup rok. Jednotlivé třídy jsou zvoleny podle logického úsudku, tedy že rok nemůže být menší než 0, nebo roven nule, musí pojmout každé číslo od 0 po současný rok a samozřejmě další v budoucnu, dejme tomu do deseti tisíc. Největší efektivitu tato technika má, když zahrneme hraniční hodnoty. Ty můžeme vidět na obrázku č.2 v podobě čísel 255 a 256. Číslo 255 v číslicové technice reprezentuje 1 byte, proto by zde měla být zahrnuta (a další na které už nezbylo místo). Důvodem je, že okolo těchto hodnot často dochází k přetečení. Hraniční hodnoty jsou tedy hodnoty, které jsou na okraji jednotlivých tříd ekvivalence. Jedná se o myšlenku, že pokud aplikace dokáže pracovat na hranici svých možností, tak za normalních okolností pravděpodobně bude pracovat taktéž dobře [1, 8].

1.2.3 Testování černé a bílé skříňky

V praxi máme několik způsobů, jakými můžeme testovat software. Ty se dělí podle toho, jak k softwaru a jeho funkcionalitám přistupujeme. Zde si probereme další rozdělení, a to na černou a bílou skříňku.

Černá skříňka znamená, že se nezajímáme o vnitřní fungování dané aplikace, ale díváme se na ní z pohledu uživatele, člověka, který má k dispozici možnost zadat určité vstupy a očekává správný výstup. Tester černé skříňky se nezajímá o vnitřní fungování aplikace a ani mu není představena. Stará se pouze o to, jak aplikace funguje jako celek, tedy splňuje-li dané specifikace či nároky a jestli očekávané chování je rovno tomu skutečnému [7].

Černou skříňku rozdělujeme na statickou a dynamickou. U statické se testují požadavky a nejasnosti ve specifikaci, které by mohly odhalit chyby v počátečních fázích vývoje a tím ušetřit velké množství peněz. U dynamického je potřeba spustitelný software [1].

Bílá skříňka znamená, že tester má k dispozici všechny informace, včetně zdrojového kódu. Ten je zároveň i předmětem testování. To dá testerovi možnost otestovat situace, které jsou jinak velmi težko dosažitelné (z hlediska černé skříňky). Tento druh testování vyžaduje alespoň nějaké znalosti programování a čistého kódu [7].

Dobrý tester v rámci testování metodou bílé skříňky pomocí svých znalostí dokáže odhadnout podezřelé situace v kódu a může podle nich upravit své další testování. Dále jeho působení může vést k zlepšení optimalizace kódu. Proto je tester s velkými znalostmi programování pro většinu velkých projektů nenahraditelný [1].

Testování bílé skříňky může být opět statické, nebo dynamické. Statické je procházení kódu a hledání podezřelých řádků, kde by mohlo dojít k chybě, popřípadě jsou jiným způsobem nebezpečné. Dále také můžou porušovat nejlepší praktiky, nebo pravidla firmy. Dynamické testování bílé skříňky je založeno na debuggingu, popřípadě vytváření podprogramů testujících určitou knihovnu, nebo funkcionalitu [1].

1.2.4 Příčiny chyb

Nejčastější příčina chyb spočívá v samotné specifikaci a návrhu, nebo v nejhorším případě jejich absencí. To, jak je aplikace navrhnutá ve svém počátku, může mít obrovký vliv na rychlost vývoje celé aplikace. Softwarové teamy často vytvoří nedokonalou, málo podrobnou specifikaci. Tu později zcela ignorují, nebo do ní přidávají věci, které jdou proti původnímu návrhu a myšlence, což je problém v případě vytváření jednotného produktu, u kterého jednotlivé komponenty a teamy na sobě závisejí [1].

Dále sem patří chyby v programování samotném. Ty jdou rozdělit na dvě hlavní kategorie:

1) Absence kontroly, syntaktické chyby (často objeví překladač) a logické chyby
2) Použití nebezpečných funkcí [10]

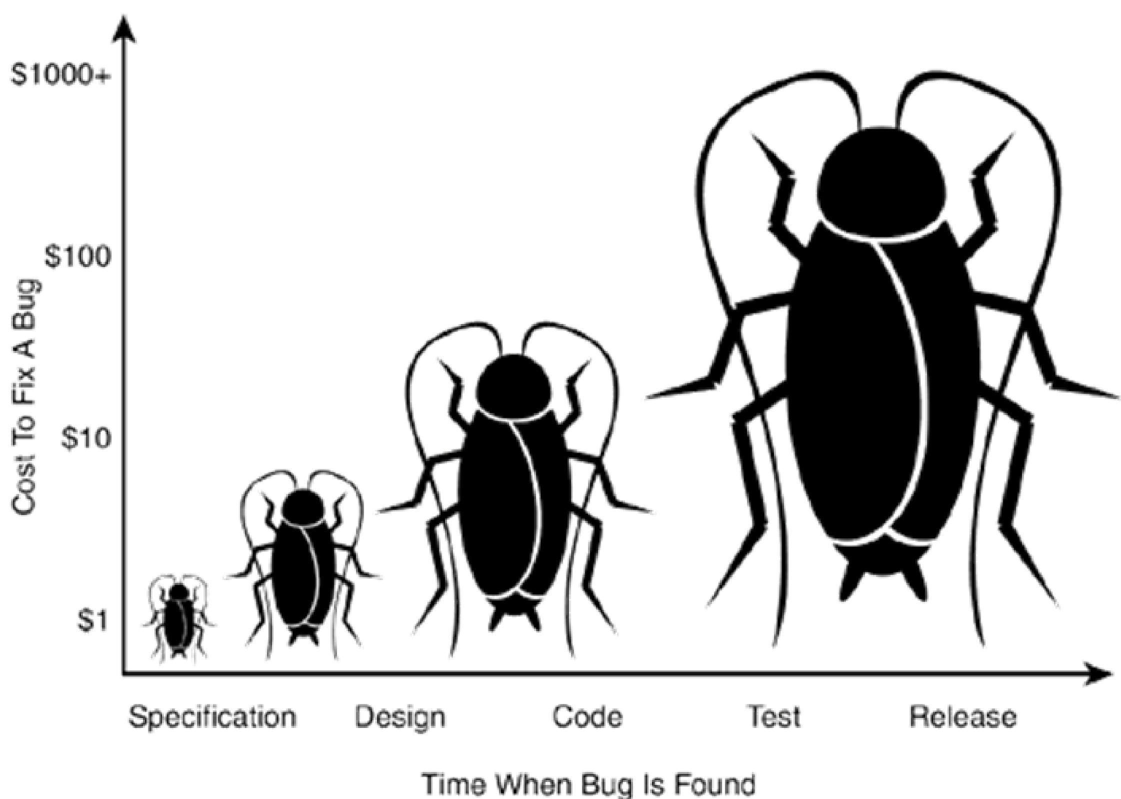
První případ je jasný. Jedná se o chyby v důsledku vytvoření špatné logiky, opomenutí nějakého chybového stavu, dělení nulou, práce s pamětí a ukazateli atd. [10].

Druhý případ je použití funkcí, které mají potenciál být nebezpečné v rukou nezkušeného programátora. Jedná se o funkce jako strcpy v jazyku C, které umožňují zkopírovat řetězec do jiného řetězce, ale už nekontrolují, jestli je pro cílové pole připraveno dostatek paměti. Ve vysokoúrovňových jazycích už se s tímto problémem nesetkáváme [10].

Příčin může být nezpočet a nemusí být ani technického rázu. Může sem patřit neshoda mezi vedoucími či jiné odlišné zvyklosti při postupu práce, ale všechny mají společný defekt, neúmyslné vytvoření chyb, které pokud se neodhalí, mají nějaké důsledky [1].

1.2.5 Důsledky chyb

Je známo, že chyby v softwaru mohou mít obrovské důsledky. My se zde budeme bavit hlavně o finančních důsledcích. Z průzkumů, které byly provedeny vyplývá, že výše nákladů má logaritmickou stupnici. To znamená, že čím později chybu objevíme, tím dražší bude její oprava a to velmi znatelně [1].



Obr. 3. Důsledek softwarových chyb [16]

Z obrázku č.3 je patrné, že cena logaritmicky narůstá podle stupně vývoje, ve kterém se produkt nachází. Najdeme-li chybu ve specifikaci, cena opravy bývá velmi malá, protože nemusí docházet k žádným opravám v kódu. Podobně je tomu při návrhu. Chyba při vývoji a testování už bývá větší, vzhledem k nutnosti zasáhnout do kódu. To může způsobit další chyby, popřípadě je chyba tak rozsáhlá a ovlivňující jiné komponenty, že se velmi těžce hledá. Nejdražší je chyba nalezená po vydání. Ta, kromě již řešených věcí může způsobit reklamaci produktu, popřípadě zdlouhavý proces od nahlášení, po nalezení a opravení

chyby, který vyžaduje mnohonásobně více peněz, než kdyby se na něj přišlo při specifikaci či návrhu [1].

1.3 Životní cyklus vývoje SW

Každý vývojový proces má určitý životní cyklus, který je ovlivněn velikostí daného softwaru a jinými okolnostmi. Tento cyklus je založen na vývojovém modelu, podle kterého produkt tvoříme. Je důležité pochopit, jaké modely se v současné době používají a jak do nich zapadají jednotlivé částí, jako programování a testování.

1.3.1 Základní modely a proč je používáme

Každý software prochází při vývoji určitým cyklem. Tento cyklus má ve výsledku vytvořit dokončený produkt. Zároveň nám umožňuje dosáhnout vyšší efektivity a produktivity při vytváření softwaru. Většinou se skládá z několika kroků, jejichž pořadí je ovlivněno konkrétním modelem. Zde je výčet možných kroků:

Specifikace - Do této fáze patří sepsání základních vlastností a charakteristik systému, požadavky na systém, vzájemná integrace a propojení

Konceptuální model – skutečnost v rámci modelu

Implementační model – konkrétní návrh

Implementace – programování a tvorba jednotlivých komponentů

Testování – ověřování funkčnosti jednotlivých procedur, které systém poskytuje

Udržování systému v provozu [11]

Vytváření softwaru podle vývojového modelu bývá často přímo úměrné následné kvalitě výsledného softwaru. Zvolit správný model je proto kritická část vývoje aplikace (hlavně u větších firem a projektů, kde pracuje více lidí). Nejhorším případem je, když projekt nemá vybraný žádný model. Tento případ Patton nazývá: "Model velkého třesku"[1]. Jedná se o situaci, kdy programátoři začnou tvořit, bez žádné specifikace a výsledný produkt je více než chaotický, jako při velkém třesku. My se zde budeme bavit o dvou nejčastějších modelech při vývoji softwaru, vodopádovém a spirálovém modelu. Samozřejmě jich ale existuje nespočet, například prototypový přístup, založený na vytváření neúplných verzí softwaru, prototypů. Dále například inkrementální, RAD přístup, agilní atd.. V praxi dochází k jejich vzájemné kombinaci [1, 12].

1.3.2 Vodopádový model

Jeden z nejstarších a nejjednodušších modelů při vývoji softwaru. Je pojmenován podle posloupnosti jednotlivých fází, které protékají z nejvyššího bodu až úplně dolů, jako u vodopádu. U dobře vybraných projektů, tento model funguje velice dobře a používá se už od počátku [1, 13]

Jednotlivé požadavky po sobě následují v přirozeném pořádí, v takovém, jaké bysme očekávali. První přichází nápad na projekt, který má nějaké požadavky. Ty musíme zanalyzovat a vymyslet důmyslný návrh, který dostatečně specifikuje, co máme a nemáme dělat. Poté produkt implementujeme, otestujeme a následuje uvolnění produktu na trh a údržba. Základem je sekvenční přístup k jednotlivým fázím, další fáze příchází až ve chvíli, kdy je předchozí bezpodmínečně dokončena. Jednotlivé kroky nelze přeskakovat a ani se nemůžeme vracet zpět, což vede k tomu, že přestup do fáze je dokončen jen v případě, že je naprostá jistota bezchybnosti té předchozí (v ideálním případě) [13].

Tento model má i několik nevýhod. Mezi hlavní patří to, že se technologie a důvody vzniku daného produktu můžou rychle vyvíjet a někdy je proto potřeba změnit něco v předchozích fázích (například modernější technologie). Dalším problémem je, že testování je provedeno pouze na konci cyklu daného vývoje, proto může dojít k nalezení chyby, když už je pozdě. Je tu také otázka změn v závislosti na potřebách klientů, které se velmi často mění [1, 13].

Tento model se často používá u menších projektů, kde se nepočítá se změnami, popřípadě se kombinuje s jinými modely, aby se odstranily jeho nedostatky.

1.3.3 Spirálový model

Spirálový model, se snaží řešit většinu nedostatků vodopádového modelu. Nejedná se o model sekvenční, ale iterativní. To znamená, že zde dochází k opakování stejného cyklu, dokud produkt není hotový. Je založen na přístupu řízených rizik, tedy postup do další fáze je závislý na analýze a prozkoumání možných rizik a problému, které můžou v dalším cyklu nastat [14].

Model obsahuje i prvky vodopádového modelu, ale řeší jeho nedostatky. Prvním je, že každá fáze je zakončená testováním, tudíž je menší šance, že by došlo k přehlédnutí nějaké chyby. Dále je zřejmé, že model je více přívětivý k často se měnícím požadavkům zákazníků a počítá s tím, že v každé další fází může dojít ke změně. Pravidelně jsou také

předávány výsledky klientovi, který na ně může dynamicky reagovat. Model je určený hlavně pro větší projekty, ale nemusí to být pravidlem [1, 14].

1.4 Manuální testování

Přestože se tato práce věnuje spíše automatizovanému testování, je nutné zde zmínit i testování manuální, bez kterého bysme se v současné době neobešli a do určité míry pravděpodobně nikdy neobejdeme.

Co to tedy je manuální testování?

Jedná se o ověření funkčnosti softwaru vyzkoušením většiny jeho funkcionalit tak, jak by to prováděl samotný uživatel. Testování by mělo odhalit nejen chyby v samotném softwaru, ale i nedokonalosti v ovládání programu, případně jiné defekty, které se jinak těžko odhalují [1, 15].

Tester, který provádí a vytváří manuální testy se neobejde bez dobře zapsaných požadavků a specifikací pro daný software, pomocí kterých vyhodnocuje co je, a co není chyba. Bez těchto požadavků by bylo velmi náročné vytvářet vhodné testovací případy. Hlavním cílem manuálního testování je dostat sofware do fáze, ve které se nachází co nejméně chyb [15].

Máme několik úrovní testování, o kterých je potřeba se zmínit.

1.4.1 Druhy testování

Jednotkové testy (Unit testy)

Provádí je sami vývojáři. Odhalí typické defekty a selhání [16].

Integrační testování

Provádí vývojáři i testeři, kteří mají opět přístup ke kódu. Patří sem testy struktury a vazeb, používají se i případy použití (TC), o kterých se budeme bavit níže. Opět může odhalit různé defekty a selhání [16].

Systémové testování

U systémového testování testeři nemají přístup ke kódu (černá skříňka). Zde se provádí manuální testování. Základem jsou specifikace, případy použití, rizika aj. Cílem je nalézt nekorektní funkcionalitu, případně další nežádoucí vlastnosti (nedostatečný výkon, špatně ovladatelné UI, atd.) [16].

Akceptační testování

Testujícím je hlavně zákazník a koncový uživatel. Testuje se metodou černé skříňky, přesněji uživatelské požadavky, případy použití a business procesy. Cílem je nalézt nesplněná akceptační kritéria, odchylky od standardů, popřípadě jiné nedokonalosti (z hlediska použitelnosti, bezpečnosti, udržovatelnosti) [16].

1.4.2 Testovací sady a testovací případy

Důležitým pojmem u testování jsou testovací případy (TC) a testovací sady (TS). TC se vytvářejí podle dokumentace daného softwaru a jejich cílem je popsat akci (kroky) s určitou částí softwaru a jejími očekávanými výsledky. Vytvářejí se jak pro manuální, tak i pro automatizové testy. TC se sepisují do dokumentu a každý TC musí splňovat určité požadavky. V zásadě obsahuje řadu vstupních hodnot, kroky, co s nimi udělat a následně očekávaný výsledek. Pokud je tento výsledek jiný než očekávaný, TC selže (fail), v opačném případě projde (pass). Co by měl obsahovat TC [17] ?

1) jednoznačný identifikátor (id),
2) datum vytvoření a autora,
3) název (stručný, ale výstižný),
4) stručný přehled o TC (sumář),
5) vstupní podmínky (preconditions),
6) výstupní podmínky (postconditions),
7) jednotlivé kroky (měly by být dostatečně deskriptivní),
8) očekávaný výsledek (velmi důležité),
9) screenshot, trasovatelnost, aj. [16]

Do TS se poté sdružují TC, které mají něco společného. Například testují společný prvek (horizontální menu). Stejně jako TC, i TS by měla obsahovat základní vlastnosti, jako id, název, její popis, účel, atd..

Kvalitní TC by měl obsahovat dostatečné množství informací a měl by být natolik deskriptivní, aby kdokoliv hned pochopil, co dělá [17].

Další možností, jak testovat software, jsou automatizované testy, které jsou probrány v následující kapitole (č. 1.5).

1.5 Automatizované testování

Jedná se o druh testování, při kterém se určité procedury automatizují (skripty) a tím dochází k výraznému ušetření času. Automatizované testování přináší i mnoho dalších výhod, kromě již zmíněné úspory času, je to rozpočet a zvýšení kvality finálního produktu. Automatizovat můžeme kromě samotných testů i celý proces testování softwaru (tomu je věnována kapitola č.9 v praktické části). Do tohoto procesu můžeme zařadit samotnou instalaci softwaru, jeho konfiguraci, provedení jednotlivých testů, vytvoření a správa výsledků testů a mnoho dalších procedur [18].

Vzniká zde otázka, kdy používat automatizované testy a kdy manuální? V případě velkého množství manuálních testovacích případů, které jsou repetetivní, je vhodnější použít automatizaci hned z několika důvodů. Prvním z nich je rychlost provedení. Ta se vůbec nemůže rovnat s rychlostí člověka. Dále efektivita. Nebylo by lepší člověka zaměstnávat kreativnějšími činnostmi (např. vytváření TC), než na rutinní provádění těch samých TC pořád dokola? Za zmínku určitě stojí i lidský faktor. Lidé se velmi často mýlí, když dělají podobné, repetetivní úkoly a dříve nebo později udělají chybu. Automat tuto chybu nikdy neudělá (pokud ho člověk vytvořil dobře). V nejideálnějším případě by se testy měly sami spustit a každý den podat zainteresovaným osobám výsledek v podobě nějakého reportu, bez zásahu člověka. Kategorie rozdělení pro automatizované testy jsou stejné jako pro manuální, protože velmi často automatizují provádění právě manuálních testů [1, 18].

1.5.1 Výhody a nevýhody

Vždy závisí na konkrétním projektu, týmu a softwaru, který se má testovat. Obecně jich ale pár můžeme najít. Výhody byly vyjmenovány výše, souvisí hlavně s úsporou času, efektivitou a odstraněním lidského faktoru, dále například spolehlivost výsledků, konzistence, frekvence testování, aj. Jsou i nějaké nevýhody?

Nevýhodou může být nutnost neustále údržby těchto testů. Jakákoliv cílená změna musí být rovněž změněna i v testech, protože jinak na ni zareagují a vyhodnotí test negativně. Odstranění lidského faktoru samozřejmě má i druhou stranu mince. Tou je, že nebudou objeveny všechny chyby, které by mohl objevit člověk. Ani sebevíc chytře naprogramovaný test v nejbližší době kompletně nezastoupí člověka, který umí zareagovat i na jiné situace, než na ty ve scénáři. S tím by však měl autor testů počítat [1, 18].

1.5.2 Typické případy použití

Častými případy použití jsou testy regresní (opětovné ověřování funkcí), smoke testy (rychlé testy, jejichž účelem je ověřit, jestli aplikace funguje správně a je připravena do další fáze testování), dále nefunkční a komparační testy. Testy by měly pokrývat část aplikace, která se často nemění. Vyplatí se je vytvářet u aplikace s velkou životností a s dlouhým vývojem (více verzí aplikace). Měly by se pokrývat části, kde je větší šance výskytu chyb [18].

Používají se například na webové stránky, různé druhy desktopových aplikací (WPF, WF), mobilních aplikací, rozsah je velmi velký.

1.5.3 Používané technologie

Je jich poměrně velké množsví. Rozdělují se hlavně podle cílené platformy a podle toho, jestli jsou zdarma (opensource), nebo placené. Vlastností pro rozdělení je ale mnohem více.

Začneme technologiemi, které souvisí s touto prací.

TestStack White

Slouží pro automatizaci aplikací vytvořených pomocí frameworku WF, WPF, Win32, Silverlight, Swt. Jeho základem je api, skrývající komplexitu knihoven UI automation library od Microsoftu a zajištuje automatizaci voláním jednoduchých příkazů. V současné době je už je zastaralé a neauktualizuje se [19].

FlaUI

Jeho účel je velmi podobný jako TestStack White (TSW), ale snaží se řešit spousty chyb zanesených v TSW, které vznikly jak časem, tak horší kvalitou kódu v TSW. Frameworku se kompletně věnuje třetí kapitola této práce.

Robot framework

Open source framework určený pro akceptační testy. Má velmi univerzální použití a jeho hlavní výhodou je snadná čitelnost testovacích skriptů (používá se python). Má velmi bohatý ekosystém a spousty pomocných knihoven a ovladačů k různým platformám [20].

Další

Mezi další (včetně těch, které používají UI k návrhu testů) patří Selenium, Robotium, Apiu, TestComplete a mnoho dalších.

2 NÁVRHOVÉ VZORY A ČISTÝ KÓD

Programování ve FlaUI, vytváření pokročilých UI testů a práce na rozhraní, které budou náplní praktické části, vyžadují znalost pokročilých metod návrhu kódu a práci s ním. Zároveň bylo nutné udržet kvalitu kódu na vyšší úrovni, proto byla snaha dodržet zásady čistého kódu. Všechny tyto techniky a myšlenky budou vysvětleny v této kapitole.

2.1 Objektově orientované programování

Jedná se o moderní metodiku vývoje softwaru, která vytváří nový náhled na program, jeho funkci a komunikaci mezi jednotlivými komponenty. Jedná se tedy o jiný způsob myšlení, než známe z procedurálního programování (například jazyk C). Základem je přemýšlení nad programem tak, jako ho vidí člověk (programátor), ne počítač. To často vede k větší čitelnosti programu i pro mnoho dalších lidí, ne jenom konkrétního programátora [21].

Na OOP se můžeme dívat i jako na zvýšení abstrakce nad programem. Je založeno více na objektech, se kterými programátor manipuluje, než na logice potřebné k této manipulaci. Této abstrakce bylo dosaženo několika základními vlastnostmi OOP o kterých budou následující řádky [22].

2.1.1 Základní pojmy

Základem OOP je ***objekt***. Jedná se o ucelenou jednotku obsahující nějaký vnitřní stav, tzn. atributy (proměnné, i další objekty), vnitřní chování (metody) a protokol zpráv (veřejné rozhraní). Základem filozofie OOP je, že všechno je objektem (často i základní proměnné jako int) a skutečně, ve většině jazyků pracujeme pouze s objekty [23].

Dalším pojmeme je ***třída***. Jedná se o schéma (přepis), které popisuje vnitřní strukturu a logiku objektu včetně jeho vnějšího rozhraní. Pomocí tříd můžeme vytvořit objekty a s nimi poté provádět určité operace, jako volat různé metody, vkládat je jako parametry jiných objektů, modifikovat je, apod. [23].

Instance třídy je datová struktura vytvořená podle konkrétní třídy, která se nachází v paměti. Instance se liší umístěním v paměti (adresou) a vnitřním stavem, který je nastaven pomocí konstruktoru, případně jiných operací [23].

OOP je dále postaveno na několika základních principech, které jsou pro něj charakteristické. Tyto principy mají za úkol zvýšit bezpečnost při programování, redukovat duplicitní kód a zvýšit čitelnost.

2.1.2 Základní principy

Mezi základní principy OOP patří zapouzdření, abstrakce, dědičnost, polymorfismus a dále například kompozice a delegování.

Zapouzdření je založeno na skrývání implementace a stavu každého objektu poskytnutím pouze veřejného rozhraní, které by mělo zajistit snadné používání a znemožnění sabotovat jeho funkcionalitu zvenčí, jiným objektem. Tento postup skrývání dat poskytuje větší bezpečnost obecně, ale také i před poškozením a znehodnocením dat [22].

V praxi to znamená, že objekt může z jiného objektu volat pouze seznam veřejných proměnných a metod. Pokud by chtěl změnit úmyslně, nebo nedopatřením, určitou privátní proměnnou, nepovede se mu to (nejsou brány v potaz případy s protected a dědičností).

Abstrakce je založena na myšlence zobrazení pouze relevatních informací o objektu, těch které jsou potřeba k jeho funkcionalitě. Abstrakce se například docílí skrytím implementačního kódu a vytvořením jednoduchých metod, které při jejich zavolání provedou to, co potřebujeme, bez znalosti jejich vnitřní funkcionality [22].

Pokud je mezi objekty nějaký vztah (hierarchický např.), můžeme použít ***dědičnost*** a snížit množství duplicitního kódu. Pomocí dědičnosti může objekt zdědit vlastnosti z jiného objektu [22, 24].

Klasickým příkladem dědičnosti je rodič a potomek. Vytvoříme-li třídu rodič s nějakými vlastnostmi, tak třída potomek je od něj bude dědit a bude je moci použít. Další situace by mohla být třída Savec a její potomci Pes a Kočka.

Posledním důležitým bodem je ***polymorfismus*** (mnoho forem z latiny). Je založen na tom, že instance třídy v závislosti na situaci (zavolání stejné metody) reagují různě, tedy tyto instance poskytují stejnou službu, ale každá instance se zachová jinak. Důvodem je opět snížení duplicity kódu [25].

2.1.3 Základy při návrhu programu

Těchto zásad je obrovské množství a stačily by na celou práci, proto jsou zde zmíněny jen ty nejdůležitější. Patří sem:

1) programování proti rozhraní,
2) skrývání implementace,
3) skládání vs dědičnost,

4) soudržnost,
5) předcházení duplicitnímu kódu [26]

Rozhraní objektu ovlivňuje, které jeho metody jsou viditelné zvenku. Jedná se o množinu informací (signatura a kontrakt), pomocí kterých entita sděluje, jak je možné s ní komunikovat a co o ní okolí ví. Do signatury dat (proměnné) patří název a typ. Do signatury metod opět název a typ, ale dále i návratová hodnota, popřípadě typy parametrů atd.. Do kontraktu patří další zásady, které ale nelze překladačem nijak ověřit. Sem patří různé povinné výsledky metod popřípadě jejich vzájemná konzistence [26, 27].

V současnosti většina OOP jazyků poskytuje k těmto účelům interface, nebo abstraktní třídu. Většina tříd by měla obsahovat vlastní interface, který usnadní práci s objektem a skryje jeho implementaci.

Skrývání implementace

Každý program se skládá z rozhraní a implementace. Implementace popisuje to, jak je daný program naprogramován, jak uvnitř doopravdy funguje. Rozhraní tedy obsahuje výčet informací, které o sobě třída zveřejní. Dobře naprogramovaná třída by měla skrýt implementaci a poskytnou kvalitní rozhraní, kterým ji budeme ovládat. Špatné skrytí implementace, popřípadě zveřejnění nesprávné části do rozhraní může mít nepříznivé důledky, co se poruch a ovladatelnosti třídy týče [26].

Skládání a dědičnost

Dědičnost by se měla používat pouze v případech, na které se opravdu hodí (vztah JE, pes je savec). Když je něco specifickým případem něčeho, pravděpodobně se jedná o vhodného kandidáta pro dědičnost (pes je specifickým případem savce). Důvodem, proč může být dědičnost nevhodná je skutečnost, že nedovoluje úplně skrýt implementační detaily a narušuje zapouzdření. Navíc se často stáva, že je použita bez rozmyslu kvůli jejím výhodám (snížení dumplicity kódu). Tato snaha ale při nevhodných případech ztroskotá (například bod nemůže být rodičem obdelníku atd.) [26].

Kompozice umožňuje, aby jiný objekt obsahoval více objektů, což poskytne vyšší kontrolu nad skrytím implementace.

Soudržnost je založena na myšlence, že každá třída, metoda a procedura dělá pouze jednu věc a dělá ji dobře. Jeden celek lze často rozdělit na více částí, které dělají jednu věc a po jejich spojení máme výsledný program, který dodržuje zásadu soudržnosti. Důsledek

nedodržení soudržnosti bývá větší chybovost programů, které se špatně testují a ještě hůře reagují na změny ve vývoji. Nejlepším řešením jsou velmi krátké metody, které se navzájem volají. Pokud má metoda více jak 10 řádků, je potřeba se zamyslet, jestli neděláme něco špatně, respektive jestli neexistuje lepší řešení [26].

Předcházení duplicitnímu kódu

Základním hnacím motorem většiny nových technik v OOP je předcházení duplicitnímu kódu. Ten je nežádoucí, vede ke zbytečným chybám, které vznikají často kvůli nepozornosti a nedodržování výše zmíněných principů [26].

2.2 Čistý kód

V každém projektu by měla být snaha vytvářet kód, který dodržuje zásady čistého kódu. Důvodů existuje nespočet. Jedním z hlavních je fakt, že kód nikdy nezanikne, protože požadavky na různé projekty mohou být tak detailní, že je nelze abstrahovat. Specifikace těchto požadavků v takovém detailu se nazývá programování a protože nikdy nezanikne, má tedy smysl vytvářet a udržovat zásady čistého kódu [28].

Co je to tedy čistý kód?

Nejlepší odpovědí jsou názory lidí, kteří nějakým způsobem přispěli k tomuto oboru a kteří se v životě setkali s obrovským množstvím kódu (Bjarne Stroustrup, Grady Booch apod.). Odpověď je následující:

Čistý kód by měl bez problému přečíst a upravovat i programátor, který jej neprogramoval. Kód by měl být elegantní a účinný a jeho logika přímočará, aby se chyby neměly kde skrývat. Měl by obsahovat smysluplná jména, minimální počet závislostí, plno břitkých abstrakcí a přímých toků řízení. Všechny procedury v něm provádějí pouze jednu věc, kterou dělají dobře, bez žádných vedlejších účinků. Má jednotkové a akceptační testy. Měl by být dobře zdokumentovaný, protože ne vždy lze kódem vyjádřit vše [28].

Některé tyto pravidla budou rozebrány i v této kapitole, ale je jich opět obrovské množství, proto budou obsaženy pouze ty nejdůležitější.

2.2.1 Proč udržovat kód čistý

Důvodem může být prostě skutečnost, že existence špatného kódu, je jako špatně postavený dům, který neobstojí při první bouři. Může vést k chybám, špatné čitelnosti, která opět zvyšuje chybovost a to může vést k pádům programu a k nespokojeným zákazníkům. Mnoho firem tak kvůli absenci čistého kódu zkrachovalo.

Jak už bylo řečeno, špatný kód se hůře čte. Cílem není vytvořit kód, který bude číst počítač, ale který bude číst člověk. Tento špatný kód může částečně způsobit rozsáhlé zpomalení vývoje (klidně i několik let). Změna kódu na jednom místě způsobí změnu na třech dalších. Tento nepořáděk se ve velmi spletitých systémech poté špatně čistí (je to téměř nemožné). Produktivita teamu se snižuje, náklady rostou kvůli nutnosti příjmout další zaměstnance a důsledek si domyslíme [28].

2.2.2 Základní pravidla čistého kódu

Smysluplná a samopopisující jména

Software se skládá z velkého množství jmen, které vytvořil programátor. Správně pojmenování čehokoliv (třídy, metody, proměnné, souboru) může být velmi důležitým kritériem při čtení kódu [28].

Jak toho dosáhnout?

Každé jméno by mělo vysvětlovat svůj význam. Dobrý výběr zabere čas, ale v konečném důsledku ho ušetří. Správná proměnná by měla vysvětlit důvod své existence, co děla a jak se používá [28].

int t; // Špatný název

int timeOfSumCompute; // Správný název

K pojmenování by se měly využívat nějaké konvence, které můžou být ovlivněny místními zvyklostmi na pracovišti, popřípadě obecně uznávanými standardmi (camel case atd.).

Jména by zároveň neměla podávat špatné informace (dezinformace). Názvy bychom měly vyjadřovat přesně, elegantně a pokud možno, jednoduše. Dále je podstatné, aby jména, která vytvoříme, šla vyslovit člověkem. Která proměnná se lépe čte, tato: *int genorndnum*, nebo tato: *genOfRandNum*. Jména, která vytvoříme, by dále měla být lehce vyhledatelná v našem IDE, protože v rozsáhlejších programech nebudeme všechny názvy hledat po jednom, snadno by nám něco uniklo. Poslední je umisťování duplicitních informací do názvů. Příklad: *int NumberOfInt* [28].

Funkce a jejich tvorba

Nejdůležitější věcí při vytváření funkcí je jejich velikost. Funkce by měla být malá (maximálně 10 řádků), výjimečně je potřeba dělat funkci delší. Menší funkce se na první pohled lépe čte, lépe se v ní hledá chyba a dohromady poté celý program dává větší smysl [28].

Jak už bylo řečeno v úvodu, funkce by měla dělat pouze jednu věc a měla by ji dělat dobře. Funkce samozřejmě v realitě musí vykonávat více věcí, ty by ale měla provádět na jedné úrovni abstrakce. Pokud tyto kroky vykonává o úroveň níže vzhledem k jejímu názvu, dělá pouze jednu věc. Další věci jsou argumenty funkce. Ideálním počtem argumentů u funkce je nula a opět platí pravidlo, čím méně tím lépe. Jen ve výjimečných případech funkce

potřebuje více jak tři argumenty. Řešením může být například zabalení argumentů do objektu a předáním ho jako argument. Dalším problémem je náročnost testování funkce s více parametry (stoupá mnohonásobně s každým parametrem) [28].

Komentáře

Nejlepší odpovědí na komentáře a čistý kód je následující citát od Briana Kernighana (tvůrce knihy o jazyku C): "Nedávejte komentáře do špatného kódu, přepište jej" [29].

Komentáře jsou nutným zlem a často přispívají k zaneřádění našich tříd a modulů. Kód by měl být sám o sobě dost expresivní, aby programátorovi sdělil, co dělá. Pokud narazíme na situaci, u které si nejsme jisti, jestli je dostatečně pochopitelná, nejlepší je se snažit změnit kód k lepšímu, ne hledat zkratku v podobě komentářů. Pokud musíme psát komentáře, tak píšeme takové, které jsou informativní, které varují před důsledky, popřípadě To-do komentáře. Špatnými jsou ty, které říkají něco, co není pravda, jsou nadbytečné, popřípadě pár dní po napsání už nejsou aktuální [28].

A další…

Korektní zpracování chyb, formátování, jednotkové testy, souběžnost atd.

2.3 Použité návrhové vzory

V programování se často nacházejí problémy, které se opakují. Většina z těchto problému může být řešena doporučeným postupem, návrhovým vzorem [26].

2.3.1 Proč používat návrhové vzory

Výhod použití návrhových vzorů je velké množství. Prvním z nich je vyšší rychlost při vývoji. Řešení na konkrétní problém nebudu muset vymýšlet sám, ale použiji variantu, kterou vymyslel někdo jiný a pravděpodobně jejímu vymýšlení věnoval mnohem více času. Návrhové vzory často také počítají s dalšími rozšířeními, které jsou u těchto problému typické, čímž dojde k ušetření dalšího času. Další výhodou je čitelnost kódu a obecně komunikace mezi programátory. Když se při vysvětlování odvolávám na konkrétní návrhový vzor, snáze mě ostatní pochopí, než když budu mluvit o nějakém vlastním řešení [26].

Návrhových vzorů je velké množství a rozdělují se do více skupin (podle jejich použití). Patří sem vzory určené pro tvorbu objektů, vzory pro strukturu programu, vzory pro chování programu atd. My se budeme zaobírat dependency injection, singleton a factory. Ty byly použity v praktické části.

2.3.2 Dependency injection

Jedná se o techniku, kdy jeden objekt vkládá závislost do jiného. Co v programování můžeme označit za závislost? Jedná se o proces, při kterém jedna třída používá funkcionalitu té druhé. Je na ní tedy závislá a nemohla by bez ní správně fungovat. Důvodem použití DI je fakt, že požadavky na různé objekty se můžou časem měnit a DI umožňuje změnit tyto požadavky za běhu programu (není potřeba znovu zkompilovat program). Jedná se tedy o určitého prostředníka, který pomůže zkonstruovat třídu podle našich požadavků. Máme tři typy [30]:

1) vložení přes konstruktor (constructor injection),
2) vložení přes setter (setter injection),
3) vložení přes rozhraní (interface injection),

Za co je tedy DI zodpovědná? Vytváření objektů, zjištění která třída je vyžaduje a poskytnutí třídy objektům.

2.3.3 Singleton

Jedná o velmi jednoduchý návrhový vzor, kdy můžeme vytvořit pouze jedinou instanci dané třídy.

Při programování nastanou situace, kdy je nevhodné, aby šlo vytvořit více instancí dané třídy. Příkladem můžou být šachy. Když si představíme hru šachy, tak na hracím poli se nachází více figurek, ale pouze jedna šachovnice. Programátor by s tím měl počítat a využít návrhového vzoru Singleton. Pokud tedy v programu v jakékoliv situaci budeme používat šachovnici, bude jistota, že používáme pouze jeden objekt. [26].

Jak lze singleton vytvořit?

Jako první uděláme konstruktor privátní, takže třídu nebudeme moci klasicky vytvořit (například přes operátor new). Poté vytvoříme statickou metodu, která v případě, že objekt ještě vytvořen nebyl, tak učiní, v případě že ano, vrátí stávající [31].

Singleton se doporučuje používat méně často, a když už, tak prozřetelně, protože činí provádění jednotkových testů náročnější.

2.3.4 Factory

Factory vzor poskytuje vyšší možnosti při vytváření tříd než klasický konstruktor. Jedná se o metodu, která jako svoji návratovou hodnotu vrací instanci dané třídy. Příkladem použití vezměme situaci, kdy máme třídu, u které můžeme při vytvoření instance zvolit z velkého množství možností (rozměry, titulek, popis, apod.). Tuto pomyslnou třídu poté v kódu vytvoříme desetkrát, takže zabírá ohromné množství místa. Pro tuto situaci byl vynalezen návrhový vzor Factory, kdy pouze zavoláme metodu s příslušným názvem a ta nám vytvoří tuto instanci. Použití konstruktoru může mít i jiné nevýhody (ve většině případů je jeho použití na místě), například konstruktor vždy vytvoří novou instanci, ať se nám to hodí, nebo ne [26, 32].

Jak tedy Factory funguje?

Dá se používat ve dvou provedeních. Jednoduchá tovární metoda a klasická tovární metoda. Jednoduchá tovární metoda funguje následovně:

V naší třídě vytvoříme statickou metodu, jejíchž návratový typ může být samotná třída, nebo rozhraní. Poté můžeme pomocí podmínek ovlivnit, jaká instance se vytvoří, jaké budou její parametry atd. Nakonec pomocí operátoru new a příslušných parametrů vrátíme

instanci, se kterou můžeme dále pracovat. Ta je zároveň zkonstruovaná podle našich požadavků a velmi sníží množství následného kódu [32].

3 FLAUI FRAMEWORK

V poslední kapitole teoretické části se budeme věnovat FLAUI frameworku, jeho předchůdcích, historii, současnému stavu, architektuře a používání v souladu s autorovými myšlenkami toho, jak by se framework měl používat. Budou zde rozebrány základní myšlenky a poznatky, které byly aplikovány do praxe vytvořením knihovny FWHandler. Ta je rozebrána v praktické části.

3.1 Proč právě FLAUI

Důvod výběru FlaUI je zapříčiněn více faktory. Hlavním z nich je platforma, na které se testy budou provádět, což v tomto případě byly WPF a WF aplikace, které jsou určené pro operační systém Windows. Podle autora je možné z části testovat pomocí FlaUI i jiné technologie, jako internetové stránky a velmi omezeně i Java UI frameworky [33].

Dalším důvodem je určitě fakt, že celý systém je zdarma a open source. Je tedy možné ho používat, upravovat a vylepšovat podle libosti. FlaUI nahradil v této době už zastaralý framework Teststack white a přinesl lepší a spolehlivější řešení, které v současné době dominuje nekomerčním řešením pro automatizované testování.

V neposlední řadě je potřeba vyzdvihnout to, jak dobře je FlaUI zdokumentovaný. FlaUI se nachází na Githubu, kde je poskytnuté celé řešení a také jsou v sekci wiki popsány hlavní části toho, jak se framework používá. V samotném projektu, který můžeme naklonovat, se nachází kromě samotného FlaUI, spousta užitečných příkladů a návrhů na řešení, ze kterých tato práce částečně čerpá. Samotná aktivita autora a ochota opravovat chyby, popřípadě přidávat nová řešení v závislosti na technologickém vývoji daných frameworků, které má testovat, je taktéž poměrně dobrá a dokud to tak zůstane, framework má před sebou zajímavou budoucnost.

3.2 Teoretický úvod

3.2.1 Co to je FlaUI

V první řadě je nutné si říci, co to FlaUI je. Jedná se o knihovnu, jejichž úkolem je poskytnout podporu pro automatizované UI testy Windows aplikací. Je založena na knihovně od Microsoftu s názvem UI automation libraries. Jde o nádstavbu, která má za úkol snížit komplexitu Windows knihoven a poskytnou další dodatečnou funkcionalitu, kterou by někdo mohl potřebovat [33].

3.2.2 Předloha a historie

FlaUI vychází ze dvou knihoven, které byly ještě do nedávna populární. Jedná se o TestStack White (TSW) a UIAComWrapper, ze kterých se autor inspiroval, ale přetvořil je od základu do FlaUI, aby se vyhnul zastaralému kódu [33].

Hlavní nevýhodou TSW je jeho zastaralá architektura a hlavně fakt, že už není aktualizován a jeho vývoj je zastaven, tudíž nemůže reagovat na změny ve vývoji, které postupem času nastávají. TSW je také znám tím, že v současném stavu má problémy s UIA3 knihovnou od Microsoftu, která podporuje nejnovější prvky pro UI testy (WPF, window store) [33].

Všechny tyto frameworky, včetně FlaUI, jsou založeny na třech knihovnách.

1) MSA (velice zastaralé, používalo se na CodedUI),
2) UIA2 (nepodporuje nejnovější prvky, hlavně WPF, nebo Windowstore apps),
3) UIA3 (podporuje nejnovější prvky, ale má oproti UIA2 problémy s WF aplikacemi) [33]

Vytvoření spolehlivé architektury okolo knihoven UIA2 a UIA3 je hlavním přínosen FlaUI oproti ostatním frameworkům, zejména TSW.

3.2.3 Výhody a nevýhody FlaUI

Výhoda

Výhoda FlaUI spočívá především ve způsobu, jakým pracuje s knihovnami UIA2 a UIA3. Jedná se hlavně o poskytnutí jednoduchého a poměrně spolehlivého rozhraní. Pomocí něj voláme jednotlivé příkazy a vnořujeme se hlouběji do stromu elementů. S nimi provádíme určité operace a zaznamenáváme výsledky. FlaUI je velice rozsáhlé a poskytuje velmi mnoho způsobů, jakým dosáhnout požadovaného výsledku [33].

Dále za zmínku stojí možnost volby mezi rozhraními UIA2 a UIA3, kdy každé z nich se hodí na něco jiného a obě jsou stejně dobře zpracovány a aktualizovány [33].

Výhod existuje samozřejmě více, od rychlosti, po příjemný způsob zápisu (pro ty, kteří mají zkušenost v programování).

Nevýhody

FlaUI má samozřejmě i nějaké nevýhody. Ty ale spíše plynou ze samotných knihoven UIA2 a UIA3, nebo obecně z tohoto druhu testování. Tyto problémy jsou řešeny v poslední

kapitole třetí části o FlaUI (č.3.4.3), ze které posléze vyplývají další rozhodnutí, ovlivňující praktickou část této práce [33].

Nevýhodou, kterou je potřeba zmínit je to, že na FlaUI v tuto chvíli pracuje pouze jeden člověk, který ji také drží při životě. Jakmile by nedocházelo k dalším aktualizacím, FlaUI by mohl následovat stejný osud, jako TSW, nebo knihovny předtím. O tomto riziku autor sám ví a počítá s tím, že pokud on někdy nebude schopen pokračovat, měl by převzít práci někdo jiný [33].

3.3 Programová struktura FlaUI

Programová struktura FlaUI je poměrně rozsáhlá, proto zde budou vysvětleny hlavně části, které jsou nějakým způsobem podstatné. Účelem této kapitoly je přiblížit, jak FlaUI uvnitř funguje. Části, kterým se budeme věnovat:

- FlaUI.UI2 a 3,
- FlaUI.Core,
- FlaUI.Core.UITests,
- FlaUI.Core.UnitTests,
- testApplications.

3.3.1 UIA2 a UIA3

Jedná se o podstatné součásti FlaUI, rozdělené do dvou knihoven. Pro WPF aplikace by se měla používat UIA3, pro WF UIA2. Rozdíly ve funkčnosti těchto dvou knihoven jsou nepatrné (kompatibilita již se zmíněnými frameworky) a proto se budeme věnovat pouze UIA3 [33].

Tyto dvě komponenty provádí propojení s Microsoft knihovnami a obsahují mnoho podpůrných funkcionalit, které umožňují automatizaci při vykonávání potřebných úkonů. Po vytvoření instance třídy automation a umístění ji jako parametr do metody GetMainWindow říkáme, že otevřené okno bude automatizováno.

3.3.2 Core

Core obsahuje většinu funkcionality, kterou používáme při standardní práci s FlaUI. Ať už se jedná o hledání prvků, operace s prvky a jiné speciální operace, všechno je zastoupeno zde. Jako první se zaměříme na složku AutomationElement, která obsahuje všechny dostupné prvky a operace k nim. Každý ovládací prvek dědí od třídy AutomationElements,

která obsahuje všechnu společnou funkcionalitu. Jedná se poměrně o komplikovanou hierarchii. Dále se v Core nachází složka input, která má za úkol zpracovávat všechny vstupní operace z myši a klávesnice, jež můžeme použít a nasimulovat tak reálné situace ovládání aplikace uživatelem. Další složkou jsou Identifiers, která obsahuje třídy pro zpracování různých druhů identifikátorů. Kromě těchto zmíněných se v Core nachází ještě Capturing pro zachycování výsledků (v podobě obrázku nebo záznamu) a další pomocné metody, jako TreeWalker pro lepší procházení stromu. Obsahu je zde velmi mnoho a byl z velké části prozkoumán, aby mohl být co nejefektivněji využitý [33].

3.3.3 UITests a TestApplication

Další podstatná část FlaUI je UITests, která obsahuje praktickou ukázku ke každé výše zmíněné funkcionalitě a zamýšlené operace při konkrétních situacích. Tato složka byla často využívána při hledání řešení na obecné problémy. Poslední součástí je TestApplication, která se skládá z testů pro dvě konkrétní aplikace, jedna vytvořená pomocí WPF frameworku a druhá pomocí WF. Z těchto dvou souborů bylo rovněž čerpáno.

3.4 Používání FlaUI

Nyní se můžeme podívat na to, jak by se FlaUI mělo používat. V praktické části poté budou tyto poznatky aplikovány na realné případy.

3.4.1 Základní komponenty

3.4.1.1 Aplikace

Prvotním komponentem je samotná ***aplikace (application)***, která je zároveň vstupním bodem testování. Abychom mohly volat potřebné příkazy na aplikaci, je nutné ji prvně odchytit, čehož lze dosáhnout více způsoby (Attach, Launch, AttachOrLaunch, LaunchStoreApp) [33].

S odchycenou aplikací pak můžeme provádět potřebné operace.

3.4.1.2 Hledání prvků a operace s nimi

Nejdůležitější operací je hledání UI prvků a FlaUI nabízí několik možností, jak toho dosáhnout. Hledání probíhá ve stromě prvků dané aplikace a tyto metody se týkají efektivního procházení tohoto stromu. Jaké máme tedy způsoby pro hledání prvků [33]?

FindFirst a *FindAll* slouží pro hledání s maximální kontrolou. Jako parametr kromě identifikátoru, předáváme i TreeScope pro přiblížení části stromu pro hledání a ConditionBase, která popisuje, co je hledáno. Více však byly využívány metody Find*FirstChild*, *FindAllChildren*, *FindFirstDescendant*, *FindAllDescendants*, protože poskytují jednoduchý a většinou spolehlivý způsob hledání [33].

Příklady použití hledání:

var firstChild = parent.FindFirstDescendant(cf => cf.ByAutomationId("id"));

Pomocí nějakého rodiče (napřiklad okna) najdeme prvního potomka s příslušným Id, které vložíme jako parametr v podobě lambda funkce. Tento základní koncept dokonale vystihuje práci ve FlaUI. Po tom, co element vyhledáme, s ním můžeme provést určité operace. To můžeme vidět na další ukázce, která je vzata a upravena z dokumentace FlaUI a ukazuje, jak autor zamýšlel práci s FlaUI [33].

```
1    var app = FlaUI.Core.Application.Launch("application.exe");
2    using (var automation = new UIA3Automation())
3    {
4    var window = app.GetMainWindow(automation);
5    var button1 = window.FindFirstDescendant(cf => cf.ByText("1"))?.AsButton();
6    button1?.Invoke();
7    }
```

3.4.1.3 Zachycení výsledků (capturing)

V případě pádu testu se hodí, když máme nějakou vizuální zpětnou vazbu. FlaUI poskytuje dva hlavní způsoby, jak zachytávat výsledky. Prvním je screenshot a druhým je video [33].

K použití je potřeba statické třídy Capture, poskytující mnoho statických metod, jako Screen (pro celou obrazovku), Element (pro konkrétní oblast, nebo prvek), ElementRectangle a Rectangle (pro konkrétní obdelníkovou oblast) [33].

Metoda vrací daný obrázek, který můžeme posléze uložit na disk, popřípadě na server. Použití této metody je velmi jednoduché a bylo použito v praktické části pro zachycování výsledků do HTML souboru.

3.4.1.4 Opakování (Retry)

Důležitým faktorem u UI testů je to, že můžou kdykoliv selhat i z důvodů, které nesouvisí s aplikační logikou. Systém opakování je proto kritickým prvkem [33].

Opět se jedná o statickou třídu, která poskytuje několik statických metod. Jako parametr této funkce je lambda funkce a do jejího těla umístíme kód, který chceme, aby se opakoval. Tento koncept byl hojně využit v praktické části. Zde vidíme ukázku:

```
1    FlaUI.Core.Tools.Retry.WhileException(() =>
2    {
3    // Kód
4    }TimeSpan.FromMilliseconds(timing));
```

Metod je opravdu velké množství. Patří se *WhileTrue*, *WhileFalse*, *WhileNull*, *WhileNotNull*, *WhileEmpty*, a konečne *WhileException* [33].

3.4.1.5 Další

Jak už bylo řečeno, FlaUI je velmi rozsáhlé a spoustu funkcionality nebylo v pozdější části použito, protože byly autorem přidány teprve nedávno, byly v experimentální fázi, nebo se prostě nehodily. Ať tak či tak, zde je další funkcionalita, která může být v konkrétních případech užitečná.

Caching se používá v případě, kdy máme v aplikaci mnoho statických prvků, obsahujících nějaké informace, které potřebujeme. Příkladem může být obrovská mřížka s velkým množstvím dat. U ní potřebujeme získat hodnotu každé buňky [33].

TreeWalker je dosažitelný z AutomationBase objektu, který poskyuje TreeWalker Factory obsahující potřebné metody. Ty slouží pro upřesnění elementu, který se má hledat, například podle podmínky (hledej jen elementy, které mají určitou vlastnost nastavenou na true atd.) [33].

Eventy jsou nedávno přidaná funkcionalita, které dělá přesně to, co bysme od ní očekávali. Sdružuje metody, které se provedou po spuštění nějaké události. Patří sem například PropertyChangedEvent, StructureChangedEvent atd. [33].

3.4.2 Potřebný software

Ke spuštění FlaUI je potřeba Visual studio a jazyk C#, protože všechna infrastuktura okolo, včetně Microsoft knihoven, je vytvořena v C#. Kromě těchto nezbytných věcí,

potřebuje FlaUI ještě minimálně jeden program. Ten musí umět zobrazit strom elementů dané aplikace a musí dokázat zobrazit její identifikátory a všechny potřebné atributy k danému elementu. Bez této aplikace bysme nebyli schopni provést FlaUI příkaz, protože jeho správný chod logicky závisí na identifikátoru daného elementu, který musíme najít a provést s ním potřebné operace. Autor FlaUI nabízí takový program s názvem FlaUInspect. Další možné programy jsou Spy, Inspect, UIVerify.

3.4.3 Problémy s FlaUI

Poslední kapitola bude věnována výčtu hlavních problému s FlaUI, které jsou známy a pojmenovány autorem a na které je potřeba si v dalších kapitolách dávat pozor.

Špatné odchytávání hlavního okna aplikace

Okno, se kterým pracujeme, se může měnit za běhu programu, například při testování menu může dojít ke změne okna, na kterém mají testy pokračovat, ale program si myslí, že pracujeme ještě s původním oknem. FlaUI částečně poskytuje řešení tohoto problému [33].

Problém s autorizací a access denied hláška

Spoustu operačních systému může mít problém s automatizovanými testy, proto všechny aplikace a IDE je potřeba spouštět jako administrátor.

To samé platí pro access denied hlášku (přístup zamítnut), která se může objevit v konzoli po spuštění testů. Řešení je vždy ve spuštění všech částí jako administrátor [33].

Další problémy:

- 32 bitový proces nemůže přistoupit k modulu 64 bitového procesu,
- nelze najít nějaký element z jakéhokoliv důvodu,
- problémy WPF vs WF aplikací,
- chybý jednoznačné identifikátory nebo podobné problémy [33],
- mnoho dalších…

Většina těchto problémů je řešena v praktické části a často udávají směr, kterým se další vývoj pomocí FlaUI vyvíjel.

II. PRAKTICKÁ ČÁST

4. NÁVRH ARCHITEKTURY

Pro celý proces bylo potřeba vymyslet důmyslnou architekturu. Ta by měla splňovat několik základních předpokladů. Patří sem jednoduchost, rozšiřitelnost, robustnost a v nejhorším případě i nahraditelnost vlajkového frameworku (FlaUI) pro případ, kdyby přestal být podporován, nebo by se objevila lepší verze. Na tyto a další vlastnosti muselo být přihlíženo hlavně na začátku návrhu, kdy se nevědělo, jestli je tato forma vůbec žádoucí a efektivní. Architektura byla v průběhu vytváření dále upravována a zlepšována, aby se zajistila její použitelnost v praxi. V následujích řádcích je výsledná architektura prozkoumána.

4.1 Základní informace

Architektura projektu se rozděluje na tři nezávislé části.

1) FWHandler – Jedná se nádstavbu pro FlaUI, která v základě dělá to, že všechnu její důležitou funkcionalitu zjednoduší do podoby jednoduchých příkazů, které posléze voláme ve všech automatizovaných TC. Zároveň by měla umět reagovat na nestandardní situace, které můžou při testování specifických aplikací nastat.
2) XmlToHtmlHandler – Jedná se důmyslný systém reportování, který po skončení testů z XML reportu vygeneruje HTML report a poté jej zašle zainteresovaným osobám. Systém dokáže působit nezávisle a při vhodném nastavení velmi efektivně spolupracuje s FWHandlerem a dokáže vytváření reportů velmi zjednodušit, což bude rozebráno níže
3) UITests – Samotné UI testy vkládáme do třídy s použitím Nunit knihovny (Nunit atributy, které ohraničí třídu a metodu) a s pomocí FWHandleru, který poskytuje sadu příkazů, které voláme

Neméně důležitou součástí jsou nástroje, které byly naprogramovány jako součást celkového balíčku a které mají za úkol zpříjemnit práci testera, popřídadě mu pomoct se situacemi, které by mohly nastat.

Při návrhu architektury byl velký důraz kladen na korektní zpracování chyby, tedy aby ani část informace, která by mohla pomoct k nalezení problému, nebyla zahozena. Ať je tedy nalezena chyba na jakékoliv vrstvě, tak putuje do místa, kde je zpracována a uložena do XML reportu, kde je poté přetransformována do HTML podoby, lépe čitelné člověkem. Tato spolupráce mezi jednotlivými komponenty je pro celé rozhraní velmi charakteristická.

Vždy byla snaha o to, aby jednotlivé komponenty na sobě byly nezávislé a fungovali sami o sobě.

4.2 Potřebné nástroje a technologie

Pro dosažení požadovaného cíle bylo zapotřebí mnoho technologií. Velká část je tvořena a udržována samotnou komunitou a těší se obrovské popularitě, proto také byly zvoleny. Jaké technologie a nástroje tedy byly použity? Jejich výběr bude zdůvodněn v jednotlivých sekcích.

Jazyky: C#, XSLT, HTML, CSS, PHP

IDE a software: Visual studio, Vmware, VisualUI-Verify, Inspect

Knihovny a frameworky: FlaUI, Nunit, Nunit console runner, Nunit test adapter

Verzovací systémy a jiné: Gitlab, Bootstrap, Live server

K těmto základním věcem musíme připočítat veškeré knihovny, které jsou zdarma k použití ve frameworku .NET a další technologie, které byly použity jen velmi okrajově, proto zde nebudou zmíněny.

4.2.1 Jazyky

Jazyky, které byly použity, jsou poměrně rozmanité. Většina kódu však byla vytvořena v jazyku C#. Volba zde byla jednoznačná, vzhledem k tomu, pro jakou platformu a projekty, měla být knihovna vytvořena. Dalším zásadním důvodem je fakt, že FlaUI spolu s ostatními použitými nástroji je naprogramován právě v C#. Kromě těchto zřejmých důvodů byly podstatné následující. Jazyk C# je velmi populární, robustní a rychlý. Většina standardních problému již byla na internetových fórech vyřešena, proto je jejich nalezení velmi rychlé. Dále se musel vybrat jazyk, který bude srozumitelný i pro osoby, které nejsou zvyklé programovat. Proto byl ve své zjednodušené formě, kterou poskytla knihovna FWHandler, ideálním kandidátem na tvorbu testů pomocí jednoduchých příkazů.

Dalším použitým jazykem je XSLT, který slouží pro převod XML na HTML. Tento jazyk je dobrý k tomu, aby se textový výstup, který poskytuje XML, převedl na člověkem čitelný text, v podobě naformátovaného HTML souboru. Ten je dále nadesignován pomocí CSS. Jednotlivé řádky v těchto jazycích se nacházejí v XSLT souboru, který provede požadovaný efekt.

Poslední za zmínku stojí například PHP, pomocí kterého byl na závěr vytvořen webový lokální server pro spuštění testů.

4.2.2 IDE a SW

IDE (integrated development enviroment) bylo vybráno Visual studio 2017. Hlavním důvodem je, že většina využívaných technologií je v něm lehce dostupná (Nuget packages). Důvodů je ale mnohem více. Visual studio obsahuje například velmi propracovaný debugger, který byl při ladění celého rozhraní kritický.

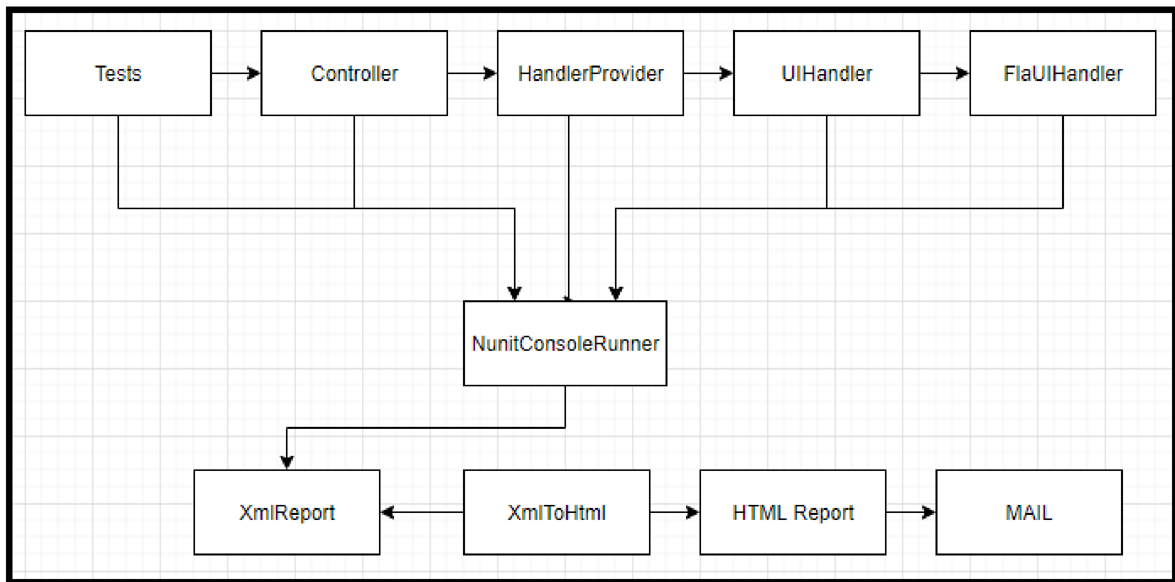
VisualUIVerify a Inspect jsou opět velmi důležité programy, bez kterých by nešly vytvářet UI testy pro WPF a WF aplikace. Každý program má určitou stromovou strukturu všech UI elementů, které obsahuje v daném okně. Každý UI element stromové struktury obsahuje nějaký identifikační prvek, například jméno, automationID aj. (v ideálním případě). Tyto programy daný strom zobrazují a dovolují uživateli zjistit konkrétní identifikátor. Příkaz, který je zavolaný v testech, na základě tohoto identifikátoru dokáže rozpoznat o jaký prvek se jedná a kde se nachází a rovněž započne požadovanou akci, například na nalezené tlačítko klikne.

4.2.3 Knihovny a frameworky

Hlavním frameworkem je samozřejmě FlaUI, které tvoří mozek celé operace. Dalé sem patří frameworky z rodiny Nunit. Základem je samotné Nunit, které slouží pro vytváření testů na .NET platformě. Obsahuje obrovské množství funkcionality samo o sobě, které ve spojení s FWHandler vytváří koherentní rozhraní s velkou škálou možností. Další v řadě je Nunit test adapter, který uživateli umožní spouštět testy přímo ve Visual studio a tím dojde k ulehčení práce, hlavně při počátečním testování. Poslední z rodiny je Nunit console runner. Obsahuje konzoli, ve které udáme nejen cestu k .dll souboru, obsahujícího testy, ale také cestu vygenerovaného výsledku.

4.3 Výsledný návrh

Po prvotním plánování a výběru technologií byl zpracován výsledný návrh, který byl posléze naprogramován. Na následujícím zjednodušeném schématu můžeme vidět celkovou architekturu.



Obr. 4. Schéma popisující architekturu systému a její provázanost

Architektura na obrázku č.4 nezachycuje všechny detaily, ale udává informaci, jak jsou jednotlivé třídy a komponenty na sobě závislé a jak spolu komunikují. V základě to funguje tak, že pomocí FWHandleru (který poskytuje jednoduché příkazy) vytvoříme test, který spustíme pomocí Nunit console runneru jednoduchým příkazem (manuálně, nebo pomocí serveru). Výsledkem testu je XML report, který je pomocí XmlToHtmlHandleru přetransformován na čitelný HTML report a odeslán příslušným osobám. Celý proces je dále zautomatizován a spuštěn pomocí serveru.

Nejdůležitější součásti je FWHandler knihovna (znázorněna na horní části schématu, obr. č. 4)

FWHandler funguje následovně. Vytvoříme nějaký test s použitím jednotlivých příkazů z Controller tříd, které jsou analogické ke každému UI prvku a můžeme s nimi v konkrétní aplikaci operovat. Tato třída vznikla hlavně z důvodu dalšího zjednodušení příkazů, které by bez ní zabíraly více místa a znepřehledňovaly by test (jeho podstatu). Další důležitou třídou je HandlerProvider. Tato třída, založená na návrhovém vzoru factory, vytváří instance pro všechny UI prvky, které následně voláme z Controlleru. Při konstrukci těchto

tříd dále vkládáme nový odkaz na FlaUIHandler. Dále sem patří UIHandler s jehož pomocí už se téměř blížíme k volání příkazů samotného FlaUI.

UIHandler je soubor tříd, který stejně jako Controller, obsahuje výčet jednotlivých ovládacích prvků. V těchto třídách se sdružují jednotlivé metody, které ke každému prvku můžeme volat. Tyto metody volají přímo FlaUIHandler. Ten už pracuje s FlaUI. Pro každý prvek UIHandleru existuje rozhraní, které sdružuje všechny metody. Důvodem jeho vzniku je nejen "best practise", ale také použití dependecy injection v třídě UIHandler, kdy přes konstruktor vkládáme závislost na FlaUIHandler a ukládáme ji do proměnné s typem rozhraní (interface). Tím se zúží možnost metod, které můžeme v dané třídě pro daný prvek volat.

FlaUIHandler je třída, která dědí od rozhraní IFWController a to dědí od všech dostupných rozhraní. Proto se v této třídě nacházejí všechny metody pro každý prvek a je tedy nejdůležitější. Komunikuje přímo s FlaUI a snaží se řešit většinu jeho nedostatků.

Všechny komponenty na obrázku č.4 jsou podrobně řešeny ve zbytku práce.

5 FLAUI V PRAXI

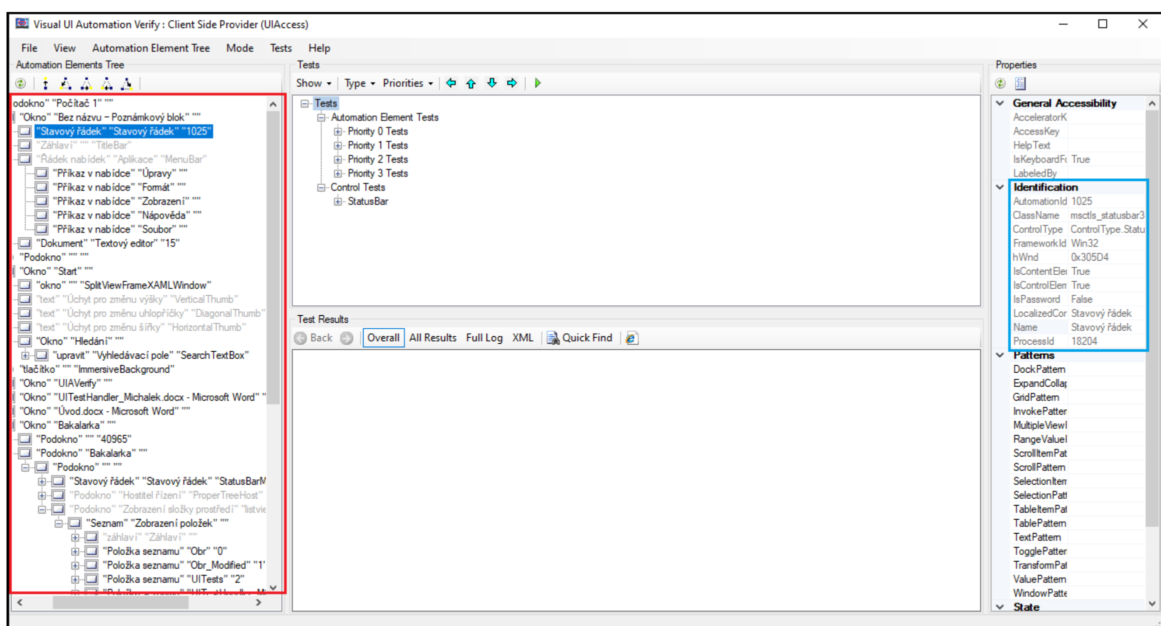
Tato kapitola pojednává o praktickém používání FlaUI. Způsobem zde uvedeným s ním bylo pracováno s většinou UI prvků, které by se v programu mohly vyskytovat. Všechny metody jsou uvedeny ve třídě FlaUIHandler, která je dále rozšiřitelná pro další metody.

Při práci s FlaUI vyplynulo na povrch určité množství problému, které byly způsobeny nejen samotným FlaUI, ale i testovaným programem. Tyto problémy bylo na obou stranách potřeba vyřešit a pojmout do finální implementace. Programování ve FlaUI je specifické v tom, že musíme brát ohled na strom prvků dané aplikace, který může být velmi vnořený, dynamicky generovaný, popřípadě nefunkční. Všem těmto situacím, jak dále uvidíme, byla snaha předejít.

5.1 Volba správné verze

FlaUI obsahuje dvě základní verze, UIA2 a UIA3. Každá verze má svoje výhody a nevýhody pro danou cílovou aplikaci (přesněji framework, na kterém je vytvořena). Obecně platí, že pro WF aplikace by se měl používat UIA2, na WPF UIA3. Dalšími rozdíly jsou různé změny v kompatibilitě pro UI prvky. U nově naprogramovaných aplikací jsou však změny naprosto minimální. Z důvodu ukázkových testů, které budou probíhat na Notepadu (WF), došlo ke zvolení verze 2, kdykoliv se však dá nahradit.

5.2 Hledání identifikátorů



Obr. 5. Ukázka programu Visual UI Verify s označenými oblastmi

Velmi důležitým programem je Visual UI Verify (popřípadě Inspect). Jedná se poměrně o jednoduchý program, který má pro naše účely jediný úkol. Nalézt identifikátor pro hledaný prvek. K jeho zobrazení dojde tak, že se namíří myší na hledaný prvek a stiskne se CTRL klávesa. Tím se provede obnovení obrazovky a změní se stromová struktura vlevo (viz červená barva na obr. č.5). Každá aplikace má určitou stromovou struktura, na které jsou poskládány všechny ovládací prvky. Příkladem je aplikace poznámkový blok obsahující nějaké základní okno (například to po spuštění). Na tomto okně je umístěna horní lišta. Vlevo nahoře na liště je umístěna ikona, vedle ní nadpis a na druhé straně tlačítka pro ukončení, zvětšení a vložení na lištu. Tato stromová struktura může být někdy velmi rozlehlá hlavně u komplexních programů. To samo o sobě často způsobuje problémy. Mezi nejčastější patří nevhodné překrývání prvků, tudíž je nějaká část ve stromě „neviditelná“, nebo aplikace prvky dynamicky generuje, takže nemají jednoznačný identifikátor atd..

Vpravo na obrázku č.5 vidíme sekci označenou modrou barvou. Jedná se o již zmíněné identifikátory, pomocí kterých přistupujeme k prvku. Nejdůležitější jsou automationID a name. S těmi poté pracujeme 99% času.

5.3 Programování ve FlaUI

Jak už bylo řečeno, vytváření FlaUIHandleru bylo ovlivněno stávajícími problémy FlaUI, které jsou například přílišná složitost v koplikovanějších situacích a nedostatečná obecnost.

5.3.1 Zachycení okna aplikace a operace s ním

Počítač obsahuje souhrn všech spuštěných úloh, které právě běží. Tyto úlohy mají různé vlastnosti, jako jméno, relativní a absolutní cestu, dobu běhu, id atd. Nás zajímá hlavně jméno, podle kterého se nám podaří identifikovat aplikaci a “zachytit ji”, abychom na ni mohli volat příslušné metody. Před tím, než provedeme ve FlaUI jakýkoliv příkaz, je třeba inicilizovat UI2Automation objekt, který se vkládá do každé metody, ve které voláme okno aplikace. Většinou ho inicializujeme pouze jednou, v metodě, která spustí aplikaci, jako vidíme na obr. č. 6 níže.

```
public void AppInicialize(string wayToApp, int timing = 0)
{
    automation = new UIA2Automation();

    FlaUI.Core.Tools.Retry.WhileException(() =>
    {
        app = FlaUI.Core.Application.Launch(wayToApp);
        window = app.GetMainWindow(automation);
    }, TimeSpan.FromMilliseconds(timing));
}
```

Obr. 6. Inicializace FlaUI a zachycení okna aplikace

Na obrázku č.6 vidíme metodu, která zahájí každý test (nebo testovací sadu). Tato metoda příjíma dva parametry, název procesu (testovaná aplikace) a čas (timing). Čas, spolu s důvodem toho, proč je celý kód obalen v lambda funkci, budeme řešit později. Uvnitř lambda funkce se nachází proměnná *app*, do které uložíme nalezenou aplikaci. Ta je nalezena pomocí metody Launch s parametrem dané cesty. Poté je do proměnné *window* uloženo samotné okno hledané aplikace. S oknem už můžeme mimo jiné provádět základní činnosti, jako s každým jiným prvkem. Tato velmi jednoduchá metoda spustí aplikaci a čeká na další příkazy.

S proměnnou app můžeme provést další operace, mezi nejdůležitější patří:

app.Close() – ukončení

app.Kill() – násilné ukončení procesu

app.Name, app.ProcessID – získání identifikátorů za běhu

Dalším problémem byla skutečnost, že u většiny aplikací může dojít k otevření dalšího okna, které je součástí dané aplikace, ale má jiné jméno. FlaUI tuto skutečnost často ignoruje a prvek se snaží najít v původním okně (kde samozřejmě není). V důsledku toho vznikla následující univerzální metoda.

```
public void AttachNewWindow(string newWindow,string lookedName, int timing = 0)
{
    automation = new UIA2Automation();
    Process correctProcess = null;
    FlaUI.Core.Tools.Retry.WhileException(() =>
    {
        var localByName = Process.GetProcessesByName(newWindow);
        foreach(Process name in localByName)
        {
            if(name.MainWindowTitle == lookedName)
            {
                correctProcess = name;
            }
        }
        app = FlaUI.Core.Application.Attach(correctProcess);
        FlaUI.Core.AutomationElements.Window window = app.GetMainWindow(automation);
        if (window == null)
        {
            throw new Exception();
        }
    }, TimeSpan.FromMilliseconds(timing));
}
```

Obr. 7. Zachycení okna aplikace odlišným způsobem (za běhu)

Tato metoda je ještě rozšířena o řešení problému, který se vyskytoval poměrně často a to ten, že některé aplikace mají stejný název. Řešení bylo vyhledání každého procesu se stejným jménem a nalezením toho, který se liší v nadpise (nebo jinde, podle potřeby)

Pokud se dále nachází aplikace, které mají stejný název procesu i stejný nadpis, bylo vytvořeno několik dalších přetížení. Ty by tuto situaci měly vyřešit, patří sem:

AttachNewWindow(string name, int timing = 0);

AttachNewWindow();

Další metody pro samotné okno:

GetWindowName()

GetHeight(), GetWidth()

5.3.2 Efektivní hledání UI prvků

V rozhraní FlaUIHandler byla snaha přijít na nějaký jednotnější a stabilnější způsob pro hledání elementů. Proto byla vyzkoušena většina způsobů hledání v různých situacích. Nejefektivnější se ukázalo použití metody FindFirstDescendants (závisí však na situaci). Tato metoda projde všechny potomky daného rodiče a hledá konkrétní výsledek pomocí identifikátoru. Má své nevýhody, hlavně u aplikací s rozlehlejším stromem. S metodou Retry, která bude vysvětlena dále, tvoří nejstabilnější kombinaci, proto se všeobecně

doporučuje na většinu operací. Na ty operace, které naopak potřebují rychlost, nebo se nachází až na dně stromu, byla vyvinuta jiná metoda. Představme si, že máme okno aplikace (id = window1), v okně se nachází další okno (id = window2), obsahující dvě tlačítka (id = btn1 a btn2). Pro nalezení *btn1* by kód vypadal následovně:

```
lookedElement = window.FindFirstDescendant(cf => cf.ByAutomationId(„btn1“));
```

Hledáme element v okně (window) pomocí zvolené metody a jako parametr vložíme identifikátor v podobě lambda funkce.

Nyní autor testu požaduje (ať už z jakéhokoliv důvodu, například chybějící identifikátory tlačítek), aby se neprocházel celý strom a došlo se přesnými cestami k hledanému prvku. Řešení vidíme v následující ukázce.

```
lookedElement = window.FindFirstDescendant(cf => cf.ByAutomationId(„window1“));

newElement = lookedElement.FindFirstDescendant(cf => cf.ByAutomationId(„window2“));

btn = newElement.FindFirstDescendant(cf => cf.ByAutomationId(„btn1“));
```

Je zřejmé, že prakticky zúžujeme oblast, než se dostaneme ke správnému výsledku. Postupy jdou libovolně kombinovat a ze zkušenosti řeší 90% problémů, na které může autor narazit. Může vzniknout otázka, k čemu tedy jsou všechny ostatní metody pro hledání prvků. Odpovědí je, že některé řeší další problémy při hledání a jiné jsou duplicitní, jen pro možnost výběru. Každopádně v rozhraní jsou zahrnuty pouze FindFirstDescendant, FindAllDescendant, FindFirstChild, FindAllChildren.

```
private FlaUI.Core.AutomationElements.Button ButtonFindBy(string name, int repeatingTime = 0)
{
    FlaUI.Core.AutomationElements.Button button = null;
    FlaUI.Core.AutomationElements.Infrastructure.AutomationElement element = lookedElement;
    lookedElement = null;

    FlaUI.Core.Tools.Retry.WhileException(() =>
    {
        var temp = element ?? app.GetMainWindow(automation);

        button = temp.FindFirstDescendant(cf => cf.ByAutomationId(name))?.AsButton();
        if (button == null)
        {
            button = temp.FindFirstDescendant(cf => cf.ByName(name))?.AsButton();
        }

        if (button is null)
            throw new ElementNotFoundException(ExceptionMessagesProvider.CreateMessage(name));

    }, TimeSpan.FromMilliseconds(repeatingTime));
    return button;
}
```

Obr. 8. Metoda určená pro hledání tlačítka

Tato metoda představuje základní myšlenku FlaUIHandleru. Jedná se o metodu, která přijímá jako parametr identifikátor a čas opakování. Z používání FWHandleru vyplynulo, že bude vhodnější, když metoda dokáže pojmout i různé jiné identifikátory, proto nezáleží na tom, jestli uživatel zadá automationID, popřípadě jméno. Jako první lokálně deklarujeme proměnnou button, a nastavíme ji na null. Poté zjistíme, jestli už neproběhlo zúžení oblasti hledaného tlačítka, tak jak to bylo vysvětleno výše. Dále v metodě Retry do proměnné *temp* uložíme buď zúženou oblast hledání, pokud nějaká existuje, nebo okno aplikace, ve kterém chceme hledat. Toto rozhodnutí zajištuje operátor "??", který se rozhoduje podle nulovosti daného stavu. Poté probíhá klasické hledání prvku, poprvé pomocí automationID (je pravděpodobnější) a posléze pomocí jména, v případě že první hledání vrátí null. Pokud je i v tomto případě výsledek null, je vhozena FWHandlerem poskytnutá výjimka, která způsobí opakování celého postupu, pokud to zbývající časový interval umožňuje. V opačném případě je tlačítko nalezeno a vráceno. Tento způsob hledání čas prověřil jako stabilní a je použit v této formě u každého UI prvku. Kvůli úplnosti byly do rozhraní přidány i metody, kdy uživatel může volat specifický identifikátor, bez operace pro rozhodnutí.

V případě, kdy UI prvek nemá žádné pojmenování, tedy není to tlačítko, checkbox, nebo jiný standardní prvek, se používá metoda *FindElementBy(...)*, která funguje uplně stejně jako button na obrázku č.8. Nejčastěji se používá ke zmíněnému zúžení oblasti hledaných prvků, nebo hledání nedefinovaného prvku (vlastního), který sdílí vlastnost se standardními UI prvky (například možnost kliknout na něj). V tomto případě nalezneme prvek, nastavíme, že má vlastnosti jako tlačítko a provedeme požadované operace.

Na obrázku č.9 můžeme vidět další variaci pro hledání.

```
public void FindAllDescendentsBy(string name, int repeatingTime = 0)
{
    FlaUI.Core.AutomationElements.Infrastructure.AutomationElement element = lookedElement;
    lookedElement = null;
    FlaUI.Core.Tools.Retry.WhileException(() =>
    {
        var temp = element ?? app.GetMainWindow(automation);
        elementArrayGeneral = temp.FindAllDescendants(cf => cf.ByAutomationId(name));

        if (elementArrayGeneral == null)
        {
            elementArrayGeneral = temp.FindAllDescendants(cf => cf.ByName(name));
        }

        if (elementArrayGeneral is null)
            throw new ElementNotFoundException(ExceptionMessagesProvider.CreateMessage(name));

    }, TimeSpan.FromMilliseconds(repeatingTime));
}

public void SelectSpecificId(int id)
{
    lookedElement = elementArrayGeneral[id];
}
```

Obr. 9. Speciální metoda určená pro hledání všech potomků se zadaným id

Tato metoda řeší situaci, kdy se ve stromě prvků nachází objemné množství informací, které nemají žadný identifikační prvek, nebo naopak mají každý stejný. Řešení existuje opět několik. Jedno z nich je nalezení všech elementů se stejným jménem, nebo všech potomků daného rodiče a přistoupení ke specifickému pomocí indexu. Opět zde vznika obrovské množství kombinací s jinými hledacími metodami a možnost řetězení.

Podobných metod je vytvořeno poměrně velké množství, aby vždy šlo vytvořit nějaké řešení a tester nemusel zasahovat do kódu. Mezi hlavní patří:

FindAllChildrenBy(…)

MenuItemFindBy(…),

CheckBoxFindBy(…),

ListFindBy(…),

ListItemFindBy(…),

TextBoxFindBy(…),

TreeFindBy(…),

TreeItemFindBy(…)

5.3.3 Operace s UI prvky

Tato kapitola se věnuje nejdůležitějším a nejzajímavějším operacím s UI prvky a jejich řešením v rozhraní FlaUIHandler. Operace s UI prvky se skládají ze dvou části. Nalezení UI prvku zavoláním metod z předešlé kapitoly (5.3.2) a zavoláním příslušné metody pro operaci. Většina UI prvků obsahuje následující metody: Click, Invoke, IsEnabled, Exists, IsChecked, IsVisible, DoubleClick, RightClick, aj. Vyjímkou je například textbox, který dále obsahuje metody pro vložení a čtení textu. Pro ukázku bude opět zvolen nejpoužívanější prvek, tlačítko (viz. obr. 10).

```
public void ButtonClickBy(string name, int repeatingTime = 0)
{
    FlaUI.Core.AutomationElements.Button button = ButtonFindBy(name, repeatingTime);
    button.Click();
}

public void ButtonDoubleClickBy(string name, int repeatingTime = 0)
{
    FlaUI.Core.AutomationElements.Button button = ButtonFindBy(name, repeatingTime);
    button.DoubleClick();
}

public void ButtonInvokeBy(string name, int repeatingTime = 0)
{
    FlaUI.Core.AutomationElements.Button button = ButtonFindBy(name, repeatingTime);
    button.Invoke();
}

public void ButtonClickByName(string name, int repeatingTime = 0)
{
    FlaUI.Core.AutomationElements.Button button = ButtonFindByName(name, repeatingTime);
    button.Click();
}

public void ButtonClickByAutomationID(string name, int repeatingTime = 0)
{
    FlaUI.Core.AutomationElements.Button button = ButtonFindByAutomationID(name, repeatingTime);
    button.Click();
}
```

Obr. 10. Ukázka možných operací s prvky

Zde vidíme metody pro kliknutí a invoke tlačíka, jak pomocí univerzálního hledání, tak pomocí specifického. Také je patrné, že vytvoření další takové metody je velmi jednoduché, pouze vyhledáme prvek a zavoláme na něj operaci. Další podstatné metody ověřují existenci, viditelnost a zapnutost tlačítka ve stromě elementů. Ty fungují podobně jako metody výše s rozdílem toho, že musíme řešit v podmínce různé stavy a v závislosti na nich vrátíme true, nebo false.

5.3.4 Třída Retry

Z obrázků u hledání je patrné, že každá metoda je zabalena v Retry tříde a zavolána pomocí lambda funkce. Tento, na první pohled zvláštní mechanismus, má velmi dobrý důvod. Jak bylo zmíněno výše, při hledání elementů můžou nastat různé komplikace. Dá se říct, že se jedná o nejrizikovější část. Jakmile se prvek najde, je už velmi malá šance, že by něco selhalo. Operace hledání je proto pojištěna systémem Retry. Ten v závislosti na parametru času (ms), který tester zvolí, bude v případě chyby opakovat celou činnost, dokud chyba nezmizí a prvek je nalezen, nebo dokud časomíra nevyprší a je vhozena výjimka. To, proč UI prvek potřebuje nějaký čas k nalezení ve stromě UI prvků, je ovlivněno mnoha faktory. Mezi základní patří vytíženost počítače, moc velká vnořenost ve stromě, chyby v aplikaci a další. Pro prvky, které jsou zadány správně a pro které vyskakuje výjimka, je nutné buď zvýšit časování, nebo zvolit jiný postup hledání prvků ve stromě.

Další situace, kde má Retry své využití, se týkají určitého čekání v aplikaci. Toto čekání může být způsobeno výkonem aplikace, čekáním na nějakou událost (například spojenou s připojením k síti), nebo jiným faktorem. Třída Retry obsahuje další různé metody, nejčastěji je využívána WhileException, tedy dokud je vhozena výjimka, tak opakuj (s ohledem na časový interval).

5.3.5 Klávesy a reporting

Tato kapitola uzavře práci s FlaUI metodami, pomocí kterých můžeme aktivovat klávesu na klávesnici, pohnout myší, popřípadě pořídit printscreen, nebo natočit video.

Součástí testování je potřeba ověřit různé klávesové zkratky či jiné operace, na které jsou uživatelé zvyklí (při práci v konkrétním SW). Na obrázku č.11 můžeme vidět jejich souhrn:

```
public void KeyBoardType(char c)
{
    FlaUI.Core.Input.Keyboard.Type(c);
}

public void KeyBoardTypeSpecialKeys(EnumProviderPressSpecialKeys keys)
{
    switch (keys)
    {
        case EnumProviderPressSpecialKeys.Enter:
            FlaUI.Core.Input.Keyboard.Type(VirtualKeyShort.ENTER);
            break;
        case EnumProviderPressSpecialKeys.Space:
            FlaUI.Core.Input.Keyboard.Type(VirtualKeyShort.SPACE);
            break;
        case EnumProviderPressSpecialKeys.Backspace:
            FlaUI.Core.Input.Keyboard.Type(VirtualKeyShort.BACK);
            break;
        case EnumProviderPressSpecialKeys.ArrowUp:
            FlaUI.Core.Input.Keyboard.Type(VirtualKeyShort.UP);
            break;
        case EnumProviderPressSpecialKeys.ArrowDown:
            FlaUI.Core.Input.Keyboard.Type(VirtualKeyShort.DOWN);
            break;
        case EnumProviderPressSpecialKeys.ArrowLeft:
            FlaUI.Core.Input.Keyboard.Type(VirtualKeyShort.LEFT);
            break;
        case EnumProviderPressSpecialKeys.ArrowRight:
            FlaUI.Core.Input.Keyboard.Type(VirtualKeyShort.RIGHT);
            break;
```

Obr. 11. Metoda pro aktivaci klávesy na klávesnici

Vidíme, že metody jsou tentokrát velmi jednoduché a intuitivní. V první pouze předáváme uživatelem zvolenou klávesu pro vykonání operace. Druhá metoda poskytuje parametr, kde uživatel může pomocí výčtu vybrat klávesu, která se poté zpracuje pomocí přepínače a vybere se korektní.

Těchto pomocných metod je celá řada. Jsou zde i takové, které FlaUI vůbec neposkytuje, jako například pro posun kolečkem myši.

Co ale FlaUI poskytuje, je možnost provést printscreen obrazovky a uložit ho na konkrétní místo. Tento postup se hojně využívá, jak uvidíme v dalších kapitolách. FlaUI umožnuje také nahrávání videí.

6 KNIHOVNA FWHANDLER A JEJÍ SOUČÁSTI

Tato kapitola se věnuje popisu všech souborů, tříd, rozhraní a komponentů, které se zde nachází. Bude zde dopodrobna rozepsána vzájemná interakce mezi komponenty a zdůvodnění, proč to tak je. Vynechána bude třída FlaUIHandler, o které pojednává předešlá kapitola (č.5).

Architektura je založena na návrhových vzorech, které jsou detailně popsány v teoretickém úvodu. Patří sem singleton, dependency injection a factory. Důvodem použití těchto konceptů byla snaha vytvoření uživatelsky přívětivého rozhraní, které si klade za cíl jednoduchost, jak pro testera, který ho bude používat, tak pro programátora, který ho bude spravovat a přidávat novou funkcionalitu.

6.1 Handler Provider

Handler provider je interní statická třída vytvořená podle návrhového vzoru factory, která má za úkol zprostředkovat instanci třídy pro každý UI prvek, který je v nabídce. Tato instance je vytvořena v podobě property, která obsahuje pouze getter (nechceme ji jakýmkoliv způsobem modifikovat v budoucnu). Jako parametr do konstruktoru vložíme odkaz na instanci FlaUIHandleru, která je vždy stejná (jedná se o singleton). Výslednou třídu můžeme vidět na obrázku č.12 níže:

```
internal static class HandlerProvider
{
    public static ButtonHandler ButtonHandler { get; } = new ButtonHandler(FlaUIHandler.Instance);
    public static MenuHandler MenuHandler { get; } = new MenuHandler();
    public static CheckBoxHandler CheckBoxHandler { get; } = new CheckBoxHandler(FlaUIHandler.Instance);
    public static TextBoxHandler TextBoxHandler { get; } = new TextBoxHandler(FlaUIHandler.Instance);
    public static AppHandler AppHandler { get; } = new AppHandler(FlaUIHandler.Instance);
    public static WindowHandler WindowHandler { get; } = new WindowHandler(FlaUIHandler.Instance);
    public static RadioButtonHandler RadioButtonHandler { get; } = new RadioButtonHandler(FlaUIHandler.Instance);
    public static ComboBoxHandler ComboBoxHandler { get;  } = new ComboBoxHandler(FlaUIHandler.Instance);
    public static InputHandler InputHandler { get; } = new InputHandler(FlaUIHandler.Instance);
    public static TreeHandler TreeHandler { get; } = new TreeHandler(FlaUIHandler.Instance);
    public static LabelHandler LabelHandler { get; } = new LabelHandler(FlaUIHandler.Instance);
    public static ListHandler ListHandler { get; } = new ListHandler(FlaUIHandler.Instance);
    public static MenuItemHandler MenuItemHandler { get; } = new MenuItemHandler(FlaUIHandler.Instance);
    public static TreeItemHandler TreeItemHandler { get; } = new TreeItemHandler(FlaUIHandler.Instance);
    public static ScrollBarsHandler ScrollBarsHandler { get; } = new ScrollBarsHandler(FlaUIHandler.Instance);
    public static ListItemHandler ListItemHandler { get; } = new ListItemHandler(FlaUIHandler.Instance);
    public static CaptureHandler CaptureHandler { get; } = new CaptureHandler(FlaUIHandler.Instance);
    public static ProgressBarHandler ProgressBarHandler { get; } = new ProgressBarHandler(FlaUIHandler.Instance);
}
```

Obr. 12. Výsledná třída HandlerProvider

Vidíme, že se v rozhraní nachází poměrně velký počet UI prvků (téměř všechny, které FlaUI podporuje). Jednotlivé property jsou umístěny ve statické třídě, proto musí být také statické. Je to z toho důvodu, abychom pro volání HandlerProvideru nemuseli zbytečně

vytvářet instanci dané třídy. Třída je dále označena jako internal. To znamená, že je přístupná pouze ve stejné assembly. Dokud do rozhraní nebyla zahrnuta Controlls nádstavba, příkazy se volaly pomocí Handler Provideru. Původní postup by byl následující:

HandlerProvider.ButtonHandler.Click("…")

Tento zápis byl posléze zkrácen třídou Controlls.

6.2 UIController, UIHandler

Tyto dvě vrsty zprostředkovávají volání metod z FlaUIHandleru. Příklad si ukážeme na textboxu, protože výborně ilistruje užitečnost tohoto rozhraní. Jako první se podíváme na UIController. Ten obsahuje všechna rozhraní (interface) pro daný element a v něm všechny metody, které můžeme volat. Do tohoto rozhraní poté v UIHandleru pomocí dependency injection vkládáme závislost na FlaUIHandler, a tím pádem zúžíme množství metod, které z něj můžeme z UIHandleru volat. Zde můžeme vidět výčet několika rozhraní:

IFWButtons, IFWApps, IFWTextBox, IFWList, IFWWindow…

Na obrázku č.13 vidíme příklad takového rozhraní:

```
public interface IFWTextBoxs
{
    //TextBox metody
    void TextBoxClickByName(string name, int repeatingTime = 0);
    void TextBoxClickByAutomationID(string name, int repeatingTime = 0);
    void TextBoxClickBy(string name, int repeatingTime = 0);
    void InsertToTextBox(string text);
    string ReadFromTextBox();
    void TextBoxFind(string value, int repeatingTime = 0);
    void TextBoxDoubleClickBy(string name, int repeatingTime = 0);
    bool TextBoxExist(string name, int repeatingTime = 0);
    bool TextBoxIsVisible(string name, int repeatingTime = 0);
    void TextBoxFindBy(int id);
}
```

Obr. 13. Rozhraní pro TextBox element

Jedná se o klasické rozhraní, které je vytvořeno pro všechny elementy. Můžeme si všimnout metod, které souvisejí s textboxem.

Na obrázku č.14 následuje UIHandler. Ten obsahuje třídy reprezentující jednotlivé UI elementy, které uvnitř vytváří proměnnou s typem příslušného rozhraní a v konstruktoru očekává instanci na FlaUIHandler.

```
public class TextBoxHandler
{
    IFWTextBoxs controller;
    public TextBoxHandler(IFWTextBoxs controller)
    {
        this.controller = controller;
    }

    private string text;
    public string Text
    {
        get
        {
            text = controller.ReadFromTextBox();
            return text;
        }
        set
        {
            text = value;
            controller.InsertToTextBox(text);
        }
    }

    public TextBoxHandler Click(string name, int repeatingTime = 0)
    {
        controller.TextBoxClickBy(name, repeatingTime);
        return new TextBoxHandler(controller);
    }
```

Obr. 14. Implementace třídy TextBoxHandler

Zde vidíme výsledek. V konstruktoru dojde k požadovanému vložení závislosti a následuje vytvoření metod, ve kterých voláme metody z controlleru a následně z FlaUIHandleru. Kvůli poslední nádstavbě s názvem Controller, je nutné vrátit instanci na danou třídu. Dále si můžeme všimnout speciálního případu pro práci s textboxem, kdy pomocí getteru a setteru vkládáme, nebo čteme z textboxu. Vzniká zde otázka, proč u textboxu poskytovat metodu Click, když do něj většinou potřebujeme pouze zapsat nebo číst. Některé textboxy před možností číst (nebo vložit), vyžadují prvnotní kliknutí.

Kdybychom tedy chtěli zavolat metodu pro zapsání do textboxu pomocí dosavadních informací, postupovali bychom následovně:

HandlerProvider.TextBox.Find(“id”).Text = “”;

6.3 Controller

Z předchozí ukázky vyplývá, že volání už je funkční, ale pořád poněkud dlouhé, kostrbaté a vedoucí k nepřehlednosti, zejména u větších testovacích případů. Proto byla vytvořena poslední nádstavba, která zlepšuje uživatelský zážitek. Dělá to velmi jednoduše. Volá předchozí příkaz v metodě s mnohem jednodušším názvem a nijak neovlivňuje funkcionalitu nebo možnost řetězení příkazů. Výsledný příkaz, tedy ten, s jakým bude pracovat tester, vypadá následovně:

Button.Click(“id”);

Oproti případům s FlaUI, kdy vytvoření takového příkazu zabralo dva až tři řádky, tedy pod podmínkou, že nenastane žádna komplikovaná situace, je to pokrok. Často je zmiňováno řetězení příkazů. To vypadá následovně.

Button.Click(“id1”).Click(„id2“).Click(„id3“);

Můžeme vidět, že tento jednoduchý a intuitivní příkaz, který klikne po sobě na tři různé tlačítka, dokáže nahradit až šest řádků FlaUI kódu. Tento postup však poskytuje i mnoho dalších výhod, což se dozvíme dále.

6.4 Systém výjimek a pomocných výčtů

Jak už bylo řečeno, důležitou vlastností FWHandleru je korektní zpracovávání chyb a výjimek, které se můžou objevit. Většinu těchto stavů bez problému odchytí Nunit console runner a následně je zapíše do XML souboru. Někdy ale výstup může být nejasný, z toho důvodu vznikl soubor GuessErrorProvider, který obsahuje souhrn nejčastějších chyb, které můžou v průběhu testu nastat a jdou předpovědět. Jsou zapsané v statické property a lze je volat z každého místa ve FWHandleru. Hláška může vypadat následovně.

Chyba: *!bError:b!: There was no error in assertion, which means: !br" + "1) Some UI cound not be found or clickable (try add waiting) !br" + "2) Some other error, for example, in transformation !br" + "For more info look at the STACK TRACE ! !br*

Jedná se o chybu, která říká, že nebyl problém s programem jako takovým, ale spíše s nalezením elementu ve stromě. Zajímavostí jsou znaky označené vykřičníkem a nějakým písmenem. Ty jsou převedeny na klasické hranaté závorky, jaké známe z HTML. Kvůli jazyku XSLT a jeho formátování, musí být označeny tak, jak je výše uvedeno.

Předpovězených chyb je připraveno přibližně dvacet, zahrnují většinu negativních stavů, které můžou nastat při práci s UI prvkem.

Dále rozhraní obsahuje jednu vlastnoručně vytvořenou výjimku ElementNotFound Exception a velice primitivní třídu, která ji zpracovává v případě, že nastane.

Další podstatnou složkou je EnumProvider, který obsahuje všechny potřebné výčty, se kterými uživatel pracuje, jako například výčet kláves, velikost scrollování a časování. U časování se chvíli zastavíme. Jak už bylo vysvětleno, u každého elementu můžeme zadat volitelný parametr pro časování, tedy jak dlouho bude hledán ve stromě prvků, než vyskočí výjimka. Číslo, které můžeme zadávat, je v milisekundách. Při testování rozhraní vyšly najevo časy, které jsou nejefektivnější k danému úkolu, nebo prostředí a nedochází ke zbytečnému čekání, které by mohlo test velmi zpomalit (když jsou testy 2, je to zanedbatelné, ale když je jich 120, tak toto číslo vyšplhá o desítky minut, což může být na serveru s vyšším provozem problém). Tyto časy jsou:

```
public enum EnumProviderTryTiming
{
    FastTry = 1000,
    StandartTry = 10000,
    SlowTry = 30000,
    VerySlowTry = 60000
};
```

Obr. 15. Enumerátor s přednastavenými hodnotami pro čekání

U většiny problémovějších prvků stačí jedna sekunda. Platí, že pokud stabilita pokulhává a testy bez důvodu padají, je vhodné dát všude FastTry. Druhou možností je 10 sekund. Ta je standardně používána, když se hledá ve větším množství dat, například v listu nebo ve stromě (UI element, není myšlen strom všech elementů aplikace), popřípadě při kratším čekání. Druhá polovina časů je vhodná pro čekání v programu na nějakou událost, popřípadě na velmi pomalé servery. Pokud je čekání delší, definuje si ho uživatel sám, v případě, že je časté, vloží ho do výčtu.

6.5 Pomocné knihovny

FWHandler vyžadoval několik přídavných knihoven na různé situace. Patří sem FileHandler, Retry a Inline logger.

FileHandler slouží pro zjednodušení některých komplikovanějších operací se soubory, patří sem práce se zipy, čtení specifické části textového souboru v závislosti na nějakém řádku, nebo obecně predikátu, smazání určitých řádků, čtení prvního řádku, otevření nejnovějšího souboru v souborové hierarchii, kopie všech souborů atd..

Retry umožňuje stejnou fukcionalitu jako retry ve FlaUI, ale mimo rozhraní FlaUI, tedy je ho možné použít v testech bez nutnosti importovat knihovny FlaUI a FWHandler.

InlineLogger je původní a pořád funkční návrh logování výsledků, který je založen na vlastnoručním získávání výsledků do proměnných a na následném vytvoření HTML souboru. Tento postup se ukázal jako nevhodný pro konkrétní testovanou situaci, ale v jiných projektech by mohl být naprosto funkční.

7 SYSTÉM REPORTOVÁNÍ

Hned od počátku bylo jasné, že rozhraní bude muset obsahovat efektivní systém reportování. Bez něj by programátor jen těžko našel a dokázal opravit chybu. Cílem bylo podat programátorovi co nejvíce informací v co nejatraktivnější podobě, aby se pouze podíval do emailu a během pár sekund věděl, kde se nachází chyba, proč vznikla a jak ji dokáže nasimulovat a odhalit. Tento systém byl pojmenován XmlToHtmlHandler a není součástí FWHandleru, tudíž funguje i samostatně, po zádání cesty k XML souboru. Jako v každé oblasti, kde systémy byly vyvíjeny k blízké spolupráci, i zde existuje konfigurace, která nejlépe pracuje s FWHandlerem. Ta zde bude zčásti nastíněna.

Před popisem jednotlivých komponentů je nutné objasnit, jak funguje v celém systému koloběh chyby. Ta může vzniknout ze čtyř důvodů:

1) Selhání přímou cestou, respektive něco je v testovém programu špatně (bug),
2) selhání při nenalezení UI prvku ve stromě (bug programu, nebo stromu UI prvků),
3) selhání samotného FlaUI nebo FWHandleru,
4) nedefinované selhání.

Jednotlivé chyby jsou zpracovány pomocí Nunit console runneru s jehož spoluprácí jsou spuštěny testy v souboru .dll. Nunit skutečně zpracuje většinu chyb, které vzniknou (jeden ze čtyř důvodů výše) a systematickým způsobem je zapíše do XML souboru (ke každému testu zvlášť chybovou hlášku atd.). Tento výstupní soubor je základem pro XmlToHtmlHandler, jak uvidíme v následujících řádcích.

7.1 Nezbytné komponenty a jejich kooperace

Mezi nezbytné komponenty systému patří Nunit, který poskytuje prostředí pro jednotlivé testovací případy a sady a zároveň obsahuje potřebnou funkcionalitu v podobě assertů. Rovněž sem patří již zmíněny Nunit console runner, který sbírá data a generuje XML report. Dalším komponentem je XmlTransformer, který převede XML soubor na HTML na základě XSLT souboru. Následuje HtmlReplace, který provádí dodatečné úpravy v designu potom, co se provede transformace. Tyto úpravy nelze zahrnout do XSLT souboru, proto musela být vytvořena dodatečná třída (patří sem úprava znaků a kódování). Poté je zde XmlController, který sdružuje a spouští veškerou zmíněnou funkcionalitu ve správném pořadí, provádí korektní poslání mailu a řeší velké množsví hraničních případů. Následuje třída obsahující Main, aby bylo možné vybuildovat spustitený .exe soubor.

Tento soubor můžeme spustit s mnoha přídavnými parametry. Tyto parametry většinou udávají cesty, odkud se má číst XML soubor, kam se má vygenerovat HTML soubor a spousty dalších volitelných parametrů. Operace s tímto velkým množstvím parametrů byly zjednodušeny pomocí McMaster.Extension.CommandLineUtils knihovny, která umožňuje pomocí atributů nadefinovat parametry do jednotlivých proměnných. Ty poté voláme na příslušném místě.

7.2 XmlTransformer

Na obrázku č.16 můžeme vidět podobu transformátoru.

```
public static void Transform(string xmlPath, string outputPath)
{
    var xmlDoc = new XPathDocument(xmlPath);
    var xslTrans = new XslCompiledTransform();
    var xmlResolver = new XmlUrlResolver();

    // Load the transformation from resources
    var programType = typeof(Program);

    string file = @"UITestBin\NUnitXslTransform.xslt";
    using (var xmlReader = XmlReader.Create(file))
    {
        xslTrans.Load(xmlReader, new XsltSettings {EnableDocumentFunction = true}, xmlResolver);
    }
    using (var writer = new XmlTextWriter(outputPath, null))
    {
        xslTrans.Transform(xmlDoc, writer);
    }
}
```

Obr. 16. Metoda Transform pro převedení XML na HTML

Jedná se o statickou metodu, která příjímá v argumentech cestu k XML souboru a výstupní cestu pro HTML soubor. Jako první načte XML soubor a připraví proměnné pro kompilaci. Do proměnné file je uložena cesta k XSLT souboru. Ten říká, která data z XML se mají vzít a kam se mají umístit do HTML. Poté načteme XML soubor s námi vytvořeným XSLT souborem a spustíme transformaci, která během okamžiku vyprodukuje HTML soubor.

7.3 XmlController

Jako první pomocí properties uložíme argumenty do proměnných. Tento postup vypadá následovně:

[Option(ShortName = "InputXml")]

public string InputXml { get; }

Poté, co máme všechny potřebné cesty, spustíme transformaci a HtmlReplace, který upraví kódování na příslušných místech. Poté se kopírují cesty podle potřeby uživatele, například pro server. Tento postup není nijak složitý, pouze pomocí cyklu foreach procházíme celou souborovu hierarchii, která obsahuje XML soubor, HTML soubor a složky obsahující obrázky v případě chyby. Tuto strukturu kopírujeme na jiné místo.

Následuje práce s mailem, opět velice jednoduchá. Zde pouze inicializujeme třídu pro odesílání mailu, předáme mu adresu, port a různé další bezpečnostní nastavení. Posléze mail odešleme. V tomto místě samozřejmě dále musíme specifikovat, komu mail chceme poslat, v případě chyby, nebo naopak.

7.4 XSLT programování

XSLT jazyk má za úkol rozparsovat XML soubor, který má formát stromové struktury, vzít z něho relevantní data a umístít je do struktury HTML, na korektní pozici, nejlépe do tabulky, nebo jiného, dynamicky se rozšiřujícího elementu. Tento poslední poznatek je velmi důležitý. Vzhledem k tomu, že nevíme, kolik dat v reportu může být, musí se struktura dynamicky rozšiřovat. K tomu byla použita tabulka, upravená pomocí CSS stylů.

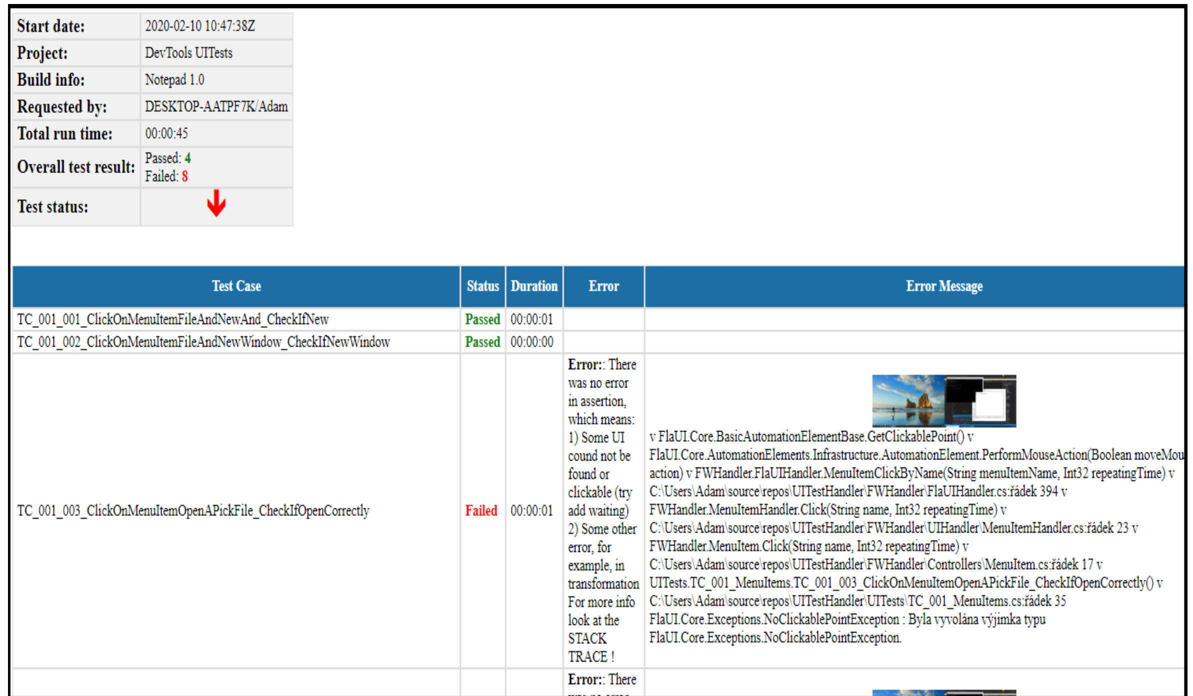
```
<td>
  <xsl:variable name="seconds" select="@duration" />
  <xsl:value-of select="format-number(floor($seconds div 3600), '00')" />
  <xsl:value-of select="format-number(floor($seconds div 60) mod 60, ':00')"/>
  <xsl:value-of select="format-number($seconds mod 60, ':00')"/>
</td>
<td>
  <xsl:if test="@result='Failed'">
    <xsl:if test="not(contains(failure/message,'Error:'))">
      <xsl:value-of select="substring-before(output,'Screen:')" />
    </xsl:if>
    <xsl:if test="contains(failure/message,'Error:')">
      <xsl:value-of select="failure/message" />
    </xsl:if>
  </xsl:if>
</td>
<td>
  <xsl:if test="@result='Failed'">
    <a>
      <xsl:attribute name="href">
        <xsl:value-of select="substring-after(output,&quot;Screen:&quot;)" />
      </xsl:attribute>
```

Obr. 17. Ukázka XSLT souboru

Na obrázku č.17 můžeme vidět část této struktury v jazyce XSLT. Vidíme, že se jedná o standardní HTML, s výjimkou některý tagů. Pomocí těchto tagů, nejčastěji s názvem xsl, dokážeme se znalostí XML vyfiltrovat přesně ty data, která potřebujeme. Tyto data filtrujeme buď pomocí jednoduchých cest, například failure/message (nalezne kořen failure a v něm vnořený message), nebo pomocí složitějších cest, které se vyhodnocují v závislosti na zadané podmínce. V prvním příkladu pracujeme s formátem času. Vyhledáme ho a pomocí jednoduchých instrukcí (modulo dělení) upravíme do čitelné podoby. Pod ním je příklad, kdy v závislosti na tom, jestli test prošel, nebo ne, vkládáme jiný druh dat (červený kříž, zelená pomlčka). Tento postup je aplikován všude v dokumentu.

7.5 Výsledný report

Po zavolání .exe souboru s příslušnými parametry, dostaneme finální HTML soubor.



Start date:	2020-02-10 10:47:38Z
Project:	DevTools UITests
Build info:	Notepad 1.0
Requested by:	DESKTOP-AATPF7K/Adam
Total run time:	00:00:45
Overall test result:	Passed: 4 Failed: 8
Test status:	

Test Case	Status	Duration	Error	Error Message
TC_001_001_ClickOnMenuItemFileAndNewAnd_CheckIfNew	Passed	00:00:01		
TC_001_002_ClickOnMenuItemFileAndNewWindow_CheckIfNewWindow	Passed	00:00:00		
TC_001_003_ClickOnMenuItemOpenAPickFile_CheckIfOpenCorrectly	Failed	00:00:01	**Error:**: There was no error in assertion, which means: 1) Some UI cound not be found or clickable (try add waiting) 2) Some other error, for example, in transformation For more info look at the STACK TRACE !	v FlaUI.Core.BasicAutomationElementBase.GetClickablePoint() v FlaUI.Core.AutomationElements.Infrastructure.AutomationElement.PerformMouseAction(Boolean moveMou action) v FWHandler.FlaUIHandler.MenuItemClickByName(String menuItemName, Int32 repeatingTime) v C:\Users\Adam\source\repos\UITestHandler\FWHandler\FlaUIHandler.cs:řádek 394 v FWHandler.MenuItemHandler.Click(String name, Int32 repeatingTime) v C:\Users\Adam\source\repos\UITestHandler\FWHandler\UIHandler\MenuItemHandler.cs:řádek 23 v FWHandler.MenuItem.Click(String name, Int32 repeatingTime) v C:\Users\Adam\source\repos\UITestHandler\FWHandler\Controllers\MenuItem.cs:řádek 17 v UITests.TC_001_MenuItems.TC_001_003_ClickOnMenuItemOpenAPickFile_CheckIfOpenCorrectly() v C:\Users\Adam\source\repos\UITestHandler\UITests\TC_001_MenuItems.cs:řádek 35 FlaUI.Core.Exceptions.NoClickablePointException : Byla vyvolána výjimka typu FlaUI.Core.Exceptions.NoClickablePointException.
			Error:: There	

Obr. 18. Finální HTML report

Chyba je vytvořena úmyslně, pro zobrazení plného rozsahu reportu. Vidíme že obsauje hlavičku se základními informace, jako celkový čas, výsledek testu atd.

Tělo obsahuje jednotlivé testy s jejich celým názvem, statusem a dobou trvání. Dále se tam nachází odhadnutá chyba poskytnutá rozhraním FWHandler a chyba pocházející přímo ze stack tracu (někdy jsou podobné). Nad chybou je klikatelný obrázek, který vizuálně ukáže, co přesně se přihodilo. Z dostupných informací by mělo být zřejmé, proč a kde chyba nastala, což bylo cílem reportu.

8 NÁVRH TESTŮ PRO APLIKACI

V této kapitole si vybereme aplikaci a otestujeme na ní řešení. Kvůli jednoduchosti byla vybrána aplikace Notepad, na ni vytvoříme 8 TC. Tyto TC poté budou přepsány do kódu, vytvořeného s kombinací FWHandler a Nunit. Návrh těchto testů by se měl řídit pravidly zmíněnými v teoretické části, přesto z důvodu vytvoření vlastního testovacího prostředí, jsou určité nejlepší techniky, které byly při práci vypozorovány a budou zde zmíněny. Dále bude vytvořeno prostředí okolo testů, které snižuje jejich duplicitu, komplexitu a umožní se testerovy soustředit pouze na testy.

Notepad byl vybrán také kvůli tomu, že se nachází v každém počítači a je tedy snadno dostupný.

8.1 Návrh TC pro automatizované testování

TC by měly částečně počítat s tím, že později budou přepsány do kódu. Jedná se o velmi jednoduché ukázky, které zároveň mají představit co nejvíce funkcí při pozdější implementaci.

TC jsou sepsány do jednotlivých tabulek, které obsahují veškeré informace. Začneme první TS, která se věnuje horizontálnímu menu.

TS_001	MenuItems
TC_001_001	ClickOnMenuItemFileAndNewFile_CheckIfNewOpen
Autor	Adam Michálek
Datum	2.4.2020
Verze	1903
Popis	Testovací případ, který testuje funkcionalitu horizontálního menu File, specificky vytvoření nové poznámky
Precondition	1) Windows 7 a vyšší 2) Spuštěná aplikace ve standartním nastavení (po instalaci)
Kroky	1) Vložíme do hlavní části pro vkládání poznámek text: Test Text -> text je korektně vložen 2) Klikneme myší na menuitem s názvem Soubor -> dojde k "vysunutí" vertikálního menu a v něm výčet dalších možností 3) Klikneme na první možnost s názvem nový -> objeví se dialog, který nás informuje, jestli chceme poznámku zahodit, nebo uložit -> klikneme zahodit, otevřela se prázdná poznámka
Vstupní data	None
Očekávaný výsledek	Výsledky jednotlivých kroků jsou podle očekávání, notepad reaguje korektně na otevření nové poznámky
Post-condition	1) Vypnout aplikaci 2) Zaznamenat screenshoty

Obr. 19. První TC

Vidíme, že TC je vytvořen tak, aby šel velmi lehce nasimulovat jak manuálně, tak v kódu. První TC testuje funkcionalitu horizontálního menu Notepadu, přesněji část pro vytvoření nové poznámky s tím, že se právě nacházíme v rozepsané poznámce, která není uložena.

TS_001	MenuItems
TC_001_002	ClickOnMenuItemOpenAndPickFile_CheckIfOpenCorrectly
Autor	Adam Michálek
Datum	2.4.2020
Verze	1903
Popis	Testovací případ, který testuje funkcionalitu horizontálního menu File a otevírání nového .txt souboru s nějaký obsahem
Precondition	1) Windows 7 a vyšší 2) Spuštěná aplikace ve standartním nastavení (po instalaci) 3) Připravený .txt soubor s nějakým obsahem, který budeme ověřovat
Kroky	1) Klikneme myší na menuitem s názvem Soubor -> dojde k "vysunutí" vertikálního menu a v něm výčet dalších možností 2) Klikneme na menuitem s názvem otevřít -> otevřel se klasický windows dialog pro výběr souboru 3) V comboboxu v pravo dole nastavíme možnost na indexu č.1, Textové dokumenty (*.txt) -> došlo ke změně v comboboxu 4) Do textboxu Název souboru vložíme cestu k našemu .txt souboru -> text je vložen 5) Klikneme na tlačítko Otevřít -> Soubor se otevřel i s příslušným textem se správným formátováním na očekávaném místě
Vstupní data	TextFile.txt
Očekávaný výsledek	Výsledky jednotlivých kroků jsou podle očekávání, notepad reaguje korektně na přidání existující poznámky
Post-condition	1) Vypnout aplikaci 2) Zaznamenat screenshoty

Obr. 20. Druhý TC

Druhý TC se opět věnuje horizontálnímu menu a to části otevření souboru s existující poznámkou. Po otevření tohoto souboru (předpřipraveného) by se v hlavní editační části měl zobrazit očekávaný text.

TS_001	MenuItems
TC_001_003	InsertTextThenClickOnMenuItemExitAndSave_CheckIfExitedAndSaved
Autor	Adam Michálek
Datum	2.4.2020
Verze	1903
Popis	Testovací případ, který testuje funkcionalitu horizontálního menu File a jeho podmenu Exit, které ukončí aplikaci
Precondition	1) Windows 7 a vyšší 2) Spuštěná aplikace ve standartním nastavení (po instalaci)
Kroky	1) Vložíme do hlavní části pro vkládání poznámek text: Test Text -> text je korektně vložen 2) Klikneme myší na menuitem s názvem Soubor -> dojde k "vysunutí" vertikálního menu a v něm výčet dalších možností 3) Klikneme na menuitem s názvem Exit -> Aplikace se prostřednictvím dialogu zeptá, jestli má obsah smazat, nebo uložit 4) Vybereme uložit -> otevře se klasický windows dialog pro uložení 5) Do textboxu Název souboru vložíme cestu, kam se má soubor uložit -> text je vložen 6) Klikneme na tlačítko uložit -> Soubor je uložen, program je zavřen
Vstupní data	Test Text
Očekávaný výsledek	Výsledky jednotlivých kroků jsou podle očekávání, notepad reaguje korektně na ukočení aplikace při neuloženém obsahu
Post-condition	1) Vypnout aplikaci 2) Zaznamenat screenshoty 3 Zkontrolovat, jestli obsah uloženého souboru je shodný s obsahem, který jsme tam skutečně vložili

Obr. 21. Třetí TC

Třetí TC rozklikne horizontální menu a pokusí se zavřít soubor s neuloženou poznámkou. V případě vyskočení dialogu "uložit" se o to pokusí a poté zkontroluje, jestli se doopravdy uložil.

To byly TC pro první sadu. Samozřejmě že by bylo možné jich vymyslet mnohem větší množství, ale na ukázku je to dostačující. Druhá testovací sada se bude věnovat přímo editoru a operacemi, které s ním můžeme provádět.

TS_002	MainEditor
TC_002_001	WriteSomethingToEditor_PressStandartKeys_CheckReaction
Autor	Adam Michálek
Datum	15.3.2020
Verze	1903
Popis	Testovací případ, který otestuje funkcionalitu editoru, hlavně jeho reakci na stisk vybraných kláves, jako enter, space, tab
Precondition	1) Windows 7 a vyšší 2) Spuštěná aplikace ve standardním nastavení (po instalaci) 3) Funkční klávesnici 4) Připravený výstupní řetězec pro porovnání
Kroky	1) Vložíme do hlavní editační části předpřipravený text -> Test Text 2) Zmáčkneme levou šipku -> kurzor je posunut doleva 3) Zmáčkneme delete -> znak před kurzorem je smazán 4) Zmáčkneme backspace -> Znak za kurzorem je smazán 5) Zmáčkneme space -> je vložena mezera 6) Porovnáme výsledný řetězec s očekávaným -> rovnají se
Vstupní data	Test Text
Očekávaný výsledek	Výsledky jednotlivých kroků jsou podle očekávání, notepad reaguje korektně na stisk speciálních kláves
Post-condition	1) Vypnout aplikaci 2) Zaznamenat screenshoty

Obr. 22. Čtvrtý TC

Čtvrtý TC testuje funkcionalitu různých kláves na hlavní editační část a sleduje jeho reakci. Poté ji uloží a porovná výstup s předpokládaným výstupem, který by se nikdy neměl změnit.

TS_002	MainEditor
TC_002_002	WriteSomethingToEditor_TryToFindIt_CheckIfFoundedCorrectNumberOfTimes
Autor	Adam Michálek
Datum	18.3.2020
Verze	1903
Popis	Testovací případ, který otestuje funkcionalitu hledání řetězců v hlavním textovém poli
Precondition	1) Windows 7 a vyšší 2) Spuštěná aplikace ve standardním nastavení (po instalaci) 3) Funkční klávesnici 4) Připravený výstupní řetězec pro hledání
Kroky	1) Vložíme do hlavní editační části předpřipravený text -> text se zobrazil 2) V horizontálním menu klikneme na Úpravy a poté Najít -> vyskočí okno pro hledání 3) Do textboxu pro hledání vložíme text, který chceme vyhledat -> text se zobrazí v textboxu 4) Klikáme na tlačítko najít další -> po posledním výskytu se zobrazil dialog informující, že další nelze nalézt 5) Klikneme OK -> Dialog zmizel, můžeme pokračovat v hledání
Vstupní data	ahoj\nahoj\ncau\ncau\nahoj , cau
Očekávaný výsledek	Výsledky jednotlivých kroků jsou podle očekávání, notepad reaguje korektně na hledání podřetězců v řetězci
Post-condition	1) Vypnout aplikaci 2) Zaznamenat screenshoty

Obr. 23. Pátý TC

Tento TC testuje funkcionalitu hledání řetězců v hlavní editační oblasti. Do ní je vložen text, který můžeme vidět na obrázku č.23 a poté se tento text pokoušíme najít tolikrát, kolikrát je to možné.

Dále za pozornost stojí TS č.4, obsahující různé ovládací operace pro zjednodušení práce v Notepadu. První TC můžeme vidět na obráku č.24

TS_004	ControlOperations
TC_004_003	WriteText_Settings_CopyPaste_Check
Autor	Adam Michálek
Datum	10.4.2020
Verze	1903
Popis	Testovací případ, který testuje funkcionalitu kopírování a vkládání
Precondition	1) Windows 7 a vyšší 2) Spuštěná aplikace ve standardním nastavení (po instalaci) 3) Funkční klávesnici 4) Připravený výstupní řetězec pro ověření správnosti
Kroky	1) Vložíme do hlavní editační části předpřipravený text (který chceme zkopírovat) -> text se zobrazil 2) V horizontálním menu klikneme na Úpravy a poté Vybrat vše -> Text v hlavní editační oblasti se označí 3) V horizontálním menu klikneme na Úpravy a poté Kopírovat -> Text zůstal označený 4) V horizontálním menu klikneme na Úpravy a poté Odstranit -> Text z hlavní editační oblasti zmizel 5) V horizontálním menu klikneme na Úpravy a poté Vložit -> Zkopírovaný text z předešlého kroku se zobrazil v okně 6) Krok č.5 zopakujeme 5x, po každém zkopírování zmáčkneme enter -> Text se korektně objevuje v okně na novém řádku
Vstupní data	Test Text
Očekávaný výsledek	Výsledky jednotlivých kroků jsou podle očekávání, kopírování a vkládání funguje správně, výstupní řetězec se rovná očekávanému
Post-condition	1) Vypnout aplikaci 2) Zaznamenat screenshoty

Obr. 24. Šestý TC

Hlavním cílem TC_004_003 je testování kopírování a vkládání pomocí horizontálního menu. Vkládání je zde provedeno v cyklu, aby byl vytvořen zajímavější řetězec.

TS_004	ControlOperations
TC_004_005	Settings_FindRow_Check
Autor	Adam Michálek
Datum	11.4.2020
Verze	1903
Popis	Testovací případ, který testuje funkcionalitu pro automatický přesun kurzoru na požadovaný řádek
Precondition	1) Windows 7 a vyšší 2) Spuštěná aplikace ve standardním nastavení (po instalaci) 3) Funkční klávesnici 4) Připravený vstupní řetězec
Kroky	1) Vložíme do hlavní editační části předpřipravený text (musí být na více řádků) -> text se zobrazil 2) V horizontálním menu klikneme na Úpravy a poté Přejít -> Objeví se dialog, obsahující textbox pro vložení požadovaného řádku (jeho výchozí hodnota je nastavena na poslední řádek) 3) V tomto textboxu změníme hodnotu na 3 (řádek 3) -> hodnota v texboxu se změnila 4) Klikneme na tlačítko Přejít -> Kurzor se objevil na požadovaném řádku 5) Kurzor ponecháme na řádku a zmáčkneme klávesu 'a' -> objevilo se písmeno 'a' před textem na řádku 6) Výstupní řetězec porovnáme s požadovaným -> rovnají se
Vstupní data	Test TextTest TextTest TextTest TextTest Text, 3, a
Očekávaný výsledek	Výsledky jednotlivých kroků jsou podle očekávání, hledání řádku funguje korektně
Post-condition	1) Vypnout aplikaci 2) Zaznamenat screenshoty

Obr. 25. Sedmý TC

V TC na obrázku č.25 pomocí dialogu přesuneme kurzor na požadovaný řádek. Na ten poté vložíme libovolný znak. Výsledný řetězec poté porovnáme s požadovaným, pokud se rovnají, přesun kurzoru byl proveden na korektní řádek.

TS_004	ControlOperations
TC_004_006	Settings_GenerateTime_Check
Autor	Adam Michálek
Datum	11.4.2020
Verze	1903
Popis	Testovací případ, který testuje funkcionalitu stisku tlačítka pro generování datumu a času do hlavní editační oblasti
Precondition	1) Windows 7 a vyšší 2) Spuštěná aplikace ve standardním nastavení (po instalaci) 3) Funkční klávesnici
Kroky	1) V horizontálním menu klikneme na Úpravy a poté Čas a datum -> do hlavní editační oblasti se vygeneruje datum a čas ve formátu H:M:S dd/yy/MM 2) Jako první zaměříme pozornost na čas a porovnáme ho s reálným časem -> časy se shodují 3) Poté zaměříme pozornost na datum a opět porovnáme s reálným datumem -> datumy se shodují 3) Jako poslední se zaměříme na celkový formát času a datumu a porovnáme jej s očekávaným -> formáty se shodují
Vstupní data	H:M:S dd/yy/MM
Očekávaný výsledek	Výsledky jednotlivých kroků jsou podle očekávání, notepad generuje správný čas a datum ve se správném formátu
Post-condition	1) Vypnout aplikaci 2) Zaznamenat screenshoty

Obr. 26. Osmý TC

Poslední TC se věnuje funkcionalitě generování času a datumu do hlavní editační oblasti. Po vygenerování dochází k ověření toho, jestli se čas a datum rovnají s očekávanými (reálnými).

Počet TC, které by šly vytvořit k Notepadu je obrovský. V tomto případě jich bylo vytvořeno 15. Rozhodl jsem se kvůli stručnosti vybrat následujících osm, které v dalších kapitolách (8.3) budou přepsány do kódu a ukáží pravý potenciál používání FWHandleru a obecně FlaUI.

8.2 Příprava prostředí pro testování

Před implementací testů je nutné navrhnout testovací prostředí, které bude sdružovat souborovou hierarchii. Tam se budou umisťovat všechny testy, identifikátory, cesty, jména, preconditiony a postconditiony. Zároveň v tomto prostředí budou platit určité konvence, aby se zajistila jeho jednotnost. Patří sem jména všech složek, souborů a testů. Dále komentáře u každého testu, jak se budou psát, jejich rozsáhlost atd.. Druh konvencí vždy závisí na pravidlech daného prostředí (např. firma).

Pro naše testy bylo vytvořeno prostředí, které se co nejvíce snaží eliminovat duplicitu a zajišťuje, aby se tester mohl soustředit pouze na vytváření testů, ne na věci okolo. Zároveň vytváří jednoduchou konvenci názvů a umisťování souborů.

Všechny soubory, které přímo souvisejí s vytvářením testů, by se měly nacházet v UITests souboru (v c# se jedná o knihovnu). Rozvržení tohoto souboru by mělo být následující. Měl by obsahovat předpřipravené třídy, do kterých se budou ukládat všechny znakové řetězce, se kterými uživatel bude pracovat. Shromažďování těchto řetězců na jedno místo je velmi důležitým zvykem, díky kterému se sníží duplicitní řetězce rozházené po všech testech a když je bude chtít uživatel změnit, provede to na jednom místě. Do těchto souborů by měl minimálně patřit PathProvider, pro všechny cesty a ValueProvider, pro všechny hodnoty. Jména ani rozdělení nemusí být ideální, v případě větších aplikací je žádoucí další rozdělení, například podle části aplikace, ve které se identifikátory nachází.

Dále soubor obsahuje všechny testovací sady (jednotlivé třídy), které obsahují jednotlivé testy (metody). Testy by vždy měly mít nějakou precondition, která bude buď globální (pro všechny testy), nebo lokální pro jednotlivé TC (případně pro jednotlivé TS). Všechny konstrukce, které mají za úkol snížit duplicitu testů, by měly být v precondition, nebo ve speciální metodě v dané TS.

Nunit framework

Nunit je knihovna, která poskytuje prostředí pro FWHandler příkazy (a pro jakékoliv jiné testy). Toto prostředí ve Visual studiu nejlépe funguje s Nunit test adapterem, který detekuje veškeré testy a je samozřejmě kompatibilní s Nunit Console runnerem, na kterém je založen systém reportování. Nunit sám o sobě poskytuje obrovskou škálu příkazů, které dokáží znatelně usnadnit práci vytváření testů. Tyto příkazy by se měly používat v každé situaci, kde jsou vhodné. Důvodem je jejich otestovanost a stabilita.

Jak postupujeme při vytváření testů?

Základem je samozřejmě vytvoření třídy, kterou bychom měli vhodně pojmenovat. V tomto případě byla vybrána následující konvence:

TS_001_MenuItem_File

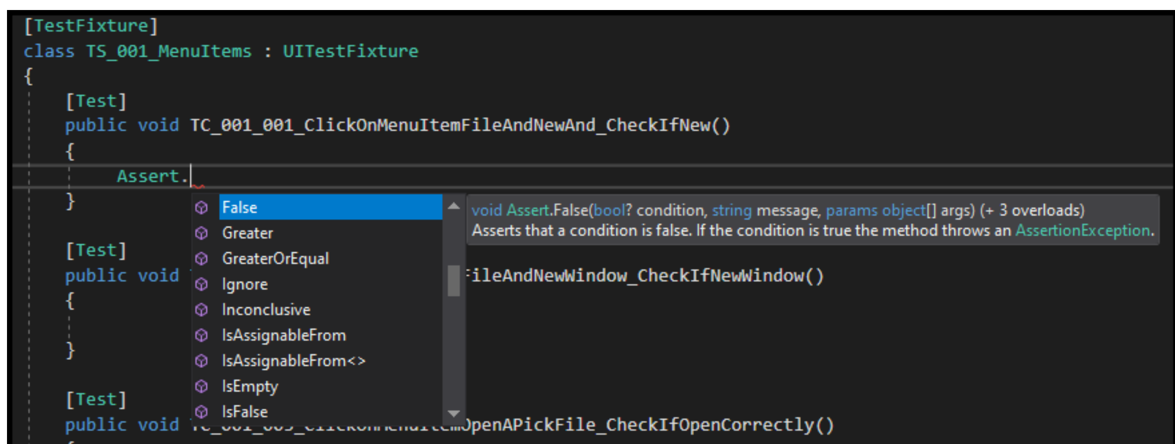
Třída představuje jednu testovací sadu. Proto identifikátor TS na začátku. Poté následuje číselný identifikátor, jehož velikost závisí na odhadovaném množství TS. Následuje textová forma, která by měla specifikovat oblast, ve které testy budou působit. Vše oddělujeme podtržítkem.

Do třídy přidáme potřebné reference.

using FWHandler;

using Nunit.Framework

Nyní můžeme přidat veškerou požadovanou funkcionalitu:



Obr. 27. Kostra pro jednotlivé testy

Na obrázku č.27 je zachyceno vše potřebné, co můžeme k vytváření testů potřebovat. Jako první označíme třídu atributem [TestFixture], což znamená, že bude obsahovat více testů. Poté vytvoříme metody s atributem [Test]. Při pojmenovávání jednotlivých testů opět platí následující konvence. Začneme TC, jako testcase, následuje číselný identifikátor testovací sady a poté číselný identifikátor testu. Následuje popis činnosti TC. Platí, že pokud je extrémně složitý, vložíme komentář. Jinak dle výběru popíšeme činnost co nejlépe (co, kde, proč). V metodě se nachází ukázka nejdůležitější funkce, kterou Nunit obsahuje, Assert. V něm je umístěno obrovské množství metod pro vyhodnocení úspěchu, nebo neúspěchu testu. Tyto metody se v testech často používají, protože při selhání generují

velké množství užitečných informací o tom, proč test selhal zrovna na tomto místě. Tyto informace, jak už víme, zpracuje Nunit console runner pro report.

Dále vidíme, že naše třída dědí od třídy UITestFixture. Ta obsahuje precondition, postcondition a všechny duplicitní metody, které používáme napříč testovacími sadami. Každá třída by měla dědit od této. Další třída, která je potřeba, je globální precondition. Často je potřeba udělat nějaké operace ještě před prvním spuštěním testovací sady. To může být vygenerování souborové hierarchie, do které se budou umisťovat výsledky apod.

8.3 Přepis jednotlivých TC do kódu pomocí FWHandleru a Nunit

Testy vytvořené FWHandlerem ve spojení s Nunit mají často velmi jednotný postup při vytváření.

Jako první se musíme dostat do oblasti, ve které testujeme (například do okna pro nastavení). Tento postup, zvláště když se opakuje, umístíme do speciální metody, popřípadě do precondition. Jak už víme z teoretického úvodu, tak test by měl nejlépe dělat pouze jednu věc, abychom věděli, co selhalo. To je ještě znásobeno u našich testů, protože kdyby jeden prováděl dvě činnosti a selhal by, Nunit console runner by logicky zachytil stack trace pouze jednoho. Platí tedy, že je 1 TC = 1 funkcionalita aplikace.

Druhou části by měl být samotný obsah testu. Býva nejrozsáhlejší a v případě potřeby opět rozdělen na menší metody pro eliminaci duplicity. Samotný test by se měl skládat z navození testované situace a ověření, jestli tato situace proběhla v pořádku.

Test by měl téměř vždy skončit uklizením po sobě, tedy navrácením programu do své původní podoby (stavu), ze které můžou vyjít ostatní testy. Nedoporučuje se vytvářet testy, které na sobě nějakým způsobem závisí, protože to vede dřív nebo později, k chybám testů.

Poslední by měla být postcondition, které opět provede úklid, popřípadě zpracuje relevantní informace (chybové printscreeny) a ukončí program.

Nyní můžeme přejít k implementaci samotných testů. Ty budou přepsány z vymyšlených TC z kapitoly 8.1.

```
[Test] // Tento test vloží do hlavní editační části text a poté se pokusí otevřít novou poznámku
public void TC_001_001_ClickOnMenuItemFileAndNewFile_CheckIfNewOpen()
{
    InitUiTest(); // Interní inicializace testu -> build aplikace, jméno metody do reportu atd...
    TextBox.Find("15").Text = ValueProvider.Notepad_TestString;
    MenuItem.Click("Soubor").Click("Nový");
    Button.Click("CommandButton_7", waitStandart);
    Assert.IsTrue(TextBox.Find("15").Text == "", GuesErrorProvider.CompareTwoValuesTextBoxError);
    MenuItem.Click("Soubor").Click("Nový");
}
```

Obr. 28. TC1 přepsaný do kódu

Jako první zavoláme InitUiTest() metodu. Ta provádí nějaké interní operace a připravuje metodu na možnost chyby (vezme její jméno a pojmenuje podle něj výstupní printscreen). Následují jednoduché FWHandler příkazy. Vždy se zaměříme na element, se kterým chceme provést operaci. Nalezneme pro něj identifikátor a zavoláme příslušnou metodu, do které vložíme identifikátor. Můžeme vidět, že kód je velmi čitelný a stačí pro něj základní angličtina. Je zde také patrný poměr funkcionalita a délka kódu. Ve FlaUI by byl rozsah trojnásobný.

Pod InitUITest metodou nalezneme textbox, přistoupíme k jeho property text a vložíme do něj náš řetězec. Dále vidíme práci s Menuitem a zároveň zřetězení, kdy menuitem pro kliknutí na další prvek nemusíme volat opětovně, stačí přidat požadovanou metodu. Tak to můžeme provádět, dokud je potřeba. Uvnitř metody IsTrue (když je podmínka false, je vhozena výjimka) hledáme textbox s identifikátorem “15” a přistupujeme k jeho obsahu. Následně testujeme, jestli je prázdný. Pokud ne, do stack tracu se zapíše chybová hláška z již vysvětleného GuessErrorProvideru, který v tomto místě s velkou přesností předpovídá, co se stalo.

```
[Test] // Otevře existující popsanou .txt poznámku a porovnájí s očekávaným výstupem
public void TC_001_002_ClickOnMenuItemOpenAndPickFile_CheckIfOpenCorrectly()
{
    InitUiTest();
    MenuItem.Click("Soubor").Click("Otevřít...");
    TextBox.Find("1148").Text = PathProvider.Notepad_PathToApp;
    ComboBox.Select("1136", 0); // ComboBox.Select("1136", jmeno);
    TextBox.Find("1136").Text = ValueProvider.Notepad_DocumentType;
    Input.KeyBoardSpecialType(EnumProviderPressSpecialKeys.Enter);
    Assert.IsTrue(TextBox.Find("15").Text == ValueProvider.Notepad_TestString,
        GuesErrorProvider.CompareTwoValuesTextBoxError);
    TextBox.Find("15").Text = "";
}
```

Obr. 29. TC2 přepsaný do kódu

Zde, kromě již zmíněných operací, vidíme práci s comboboxem (rovnou dvě varianty) a poté práci se vstupem z klávesnice, kdy do parametru metody vložíme danou klávesu pomocí výčtu.

```
[Test] // Zapíše do hlavní editační oblasti, uloží soubor a zkontroluje, jestli se obsah souboru shoduje se zapsaným
public void TC_001_004_InsertTextThenClickOnMenuItemExitAndSave_CheckIfExitedAndSaved()
{
    InitUiTest();
    TextBox.Find("15").Text = ValueProvider.Notepad_TestString;
    MenuItem.Click("Soubor", waitSlow).Click("Ukončit", waitSlow);
    Button.Click("CommandButton_6", waitStandart);
    TextBox.Find("1001").Text = PathProvider.Notepad_TestFile;
    Input.KeyBoardSpecialType(EnumProviderPressSpecialKeys.Enter);
    bool isOff = false;
    try
    {
        App.AppAttachNewWindow("notepad", waitSlow);
        isOff = false;
    }
    catch (Exception ex) { isOff = true; }

    Assert.False(isOff == false, GuesErrorProvider.AppExitError);
    string fileString = File.Exists(PathProvider.Notepad_TestFile) == true ?
        File.ReadAllText(PathProvider.Notepad_TestFile) : "";
    Assert.IsTrue(fileString == ValueProvider.Notepad_TestString, GuesErrorProvider.CompareTwoValuesFileError);
}
```

Obr. 30. TC3 přepsaný do kódu

Trochu složitější TC, který ukazuje práci s procesy, specificky testování jejich existence. Jako první vypneme aplikaci (uložením souboru a ukončením) a poté se ji v bloku Try snažíme určitý časový interval znovu najít. Pokud nalezen nebyl, aplikace je skutečně vypnutá (a nezůstala někde v pozadí běžet). Dále vidíme, že test obsahuje dva asserty, jeden ověřuje, jestli je aplikace opravdu vypnutá pomocí boolovské proměnné. Druhý testuje obsah souboru, který jsme uložili (jestli se shoduje s očekávaným).

```
[Test] // Test který ověřuje funkci speciálních kláves na hlavní editor
public void TC_002_002_WriteSomethingToEditor_PressStandartKeys_CheckReaction()
{
    InitUiTest();
    TextBox.Click("15");
    TextBox.Find("15").Text = ValueProvider.Notepad_TestString;
    Input.KeyBoardSpecialType(EnumProviderPressSpecialKeys.ArrowLeft)
         .KeyBoardSpecialType(EnumProviderPressSpecialKeys.Delete)
         .KeyBoardSpecialType(EnumProviderPressSpecialKeys.Backspace)
         .KeyBoardSpecialType(EnumProviderPressSpecialKeys.Space);
    Assert.IsTrue(TextBox.Find("15").Text == ValueProvider.Notepad_TestString_Modified,
        GuesErrorProvider.CompareTwoValuesTextBoxError);
    CleanUp();
}
public void CleanUp()
{
    MenuItem.Click("Soubor").Click("Ukončit");
    Button.Click("CommandButton_7", waitStandart);
}
```

Obr. 31. TC4 přepsaný do kódu

Zde můžeme vidět metodu CleanUp, provádějící navrácení programu do výchozího stavu. Rozdělování TC na metody je velmi žádoucí, zvláště když se poté používají ve více TC.

```
[Test] // Vloží do hlavní editační oblasti řetězec (vícekrát) a pak se ho pokouší najít (vícekrát)
public void TC_002_004_WriteSomethingToEditor_TryToFindIt_CheckIfFoundedCorrectNumberOfTimes()
{
    InitUiTest();
    int numberOfLookedString = 3;

    TextBox.Find("15").Text = ValueProvider.Notepad_TestToFind;
    MenuItem.Click("Úpravy").Click("Najít...");
    TextBox.Find("1152").Text = "cau";
    for(int i = 0; i < numberOfLookedString; i++)
    {
        Button.Click("1", waitFast);
    }
    Assert.IsTrue(Button.IsVisible("OK", waitFast));
    Window.Find("Poznámkový blok").Button.Click("2", waitFast); // stejné jako toto Button.Click("2")
    CleanUp();
    // Windows.Find("").Find("").Find("").Button...
    // Windows.Find("").FindDescendants("").SelectSpecificId(2).FindChildrens("").Click("");
}
```

Obr. 32. TC5 přepsaný do kódu

V posledním TC této TS vidíme používání cyklu For pro opakované kliknutí na tlačítko. Poté následuje nestandardní způsob kliknutí na tlačítko (pod příkazem assert). Jako první vyhledáme území, ve kterém se tlačítko nachází (rodiče), a poté v něm hledáme tlačítko (potomka). V tomto případě je to zbytečné, ale je to ukázka toho, jak se hledají hůře dostupné elementy. Jak můžeme vidět v komentářích pod testem, hledání můžeme do sebe vnořovat opakovaně různými způsoby, což nám nechává mnoho možností, jak svobodně procházet strom elementů, takže uživatel není limitován jednou možností. Ty se však používají velmi zřídka.

```
[Test] // Vloží text do hlavní editační části a pomocí menu Úpravy ho zkopiruje a vloží
public void TC_004_003_WriteText_Settings_CopyPaste_Check()
{
    InitUiTest();
    TextBox.Find("15").Text = ValueProvider.Notepad_TestString;
    MenuItem.Click("Úpravy").Click("Vybrat vše");
    MenuItem.Click("Úpravy").Click("Kopírovat");
    MenuItem.Click("Úpravy").Click("Odstranit");
    for (int i = 0; i < 5; i++)
    {
        MenuItem.Click("Úpravy").Click("Vložit");
        Input.KeyBoardSpecialType(EnumProviderPressSpecialKeys.Enter);
    }
    Assert.IsTrue(TextBox.Find("15").Text == ValueProvider.Notepad_CopyPasteTest,
        GuesErrorProvider.CompareTwoValuesTextBoxError);
}
```

Obr. 33. TC6 přepsaný do kódu

Na obrázku č.33 vidíme TC pro ověření kopírování a vkládání. Proces je velmi podobný jako u předchozích TC. Provedeme inicializaci, vložíme text pro zkopírování a provedeme další operace s MenuItem třídou. Poté pomocí cyklu vložíme text pětkrát a každý průchod zakončíme klávesou enter (nový řádek). TC je zakončen porovnáním výsledného výstupu s očekávaným.

```
[Test] // Přesuneme kurzor na řádek (pomocí Menu) a ověříme, jestli se doopravdy přesunul
public void TC_004_005_Settings_FindRow_Check()
{
    InitUiTest();
    TextBox.Find("15").Text = ValueProvider.Notepad_CopyPasteTest;
    MenuItem.Click("Úpravy").Click("Přejít...");
    TextBox.Find("Číslo řádku:", waitStandart).Text = "3";
    Button.Click("Přejít", waitStandart);
    Input.KeyBoardType('a');
    Assert.IsTrue(TextBox.Find("15").Text == ValueProvider.Notepad_CopyPasteTest_Modified,
      GuesErrorProvider.CompareTwoValuesTextBoxError);
}
```

Obr. 34. TC7 přepsaný do kódu

Na obrázku č.34 vidíme TC pro ověření funkčnosti přesunu kurzoru na konkrétní řádek pomocí menu. Tento TC je zajímavý ve způsobu, jakým provádí kontrolu. Jakmile dojde k přesunu na řádek, vloží před text, který se na řádku nachází, písmeno. Výsledný řetězec spolu s písmenem porovnáme s očekávaným (kde písmeno bude vloženo). Pokud se rovnají, je jisté, že přesun kurzoru funguje korektně.

```
[Test] // Vygeneruje pomocí menu datum ve standardním formátu, a ověří jeho správnost
public void TC_004_006_Settings_GenerateTime_Check()
{
    InitUiTest();
    MenuItem.Click("Úpravy").Click("Čas a datum");
    string notePad = TextBox.Find("15").Text;
    string[] splited = notePad.Split(' ');
    string notePadDate = splited[1];
    string notePadHour = splited[0].Substring(0, 2);
    string date = DateTime.Now.Date.ToString("dd/MM/yyyy");
    string hour = DateTime.Now.Hour.ToString();

    Assert.That(date == notePadDate, GuesErrorProvider.CompareTwoValuesTextBoxError);
    Assert.That(hour == notePadHour, GuesErrorProvider.CompareTwoValuesTextBoxError);
}
```

Obr. 35. TC8 přepsaný do kódu

V posledním TC vygenerujeme čas a datum do editoru. Ten poté vezmeme a rozparsujeme do požadovaného formátu. Následně získáme reálný čas a datum pomocí třídy DateTime. V posledním kroku porovnáme získané časy pomocí metody That. Ta je výhodná v tom, že v případě pádu testu dokáže získat velké množství informací pro XmlToHtmlHandler.

9 NASAZENÍ TESTŮ NA SERVER

Tato kapitola se věnuje samotnému nasazení testů z předchozí kapitoly (č.8) na server. Bude zde popsáno, jak by to mělo být provedeno v praxi a jaké technologie jsou pro to vhodné. Závěrem bude krátka a jednoduchá ukázka řešení pro tuto práci a pro menší projekty.

Má vůbec význam nasadit testy na server? Nestačilo by je pouze lokálně spustit? Odpověď závisí na velikosti projektu, obecně je ale pohodlnější, když nám jednou za den dojdou výsledky testu a nemusíme řešit jejich spuštění. Zároveň server dokáže, po správném nastavení, provést korektně všechny potřebné vstupní podmínky pro inicializaci testového prostředí a posléze provede i úklid. To vše ušetří čas a odstraní lidský faktor, který může chybovat. Proto i jednoduchý, dobře nastavený server je lepší než žádný.

9.1 Server v praxi

V praxi by byl využit Virtuální privátní server, na kterém by bylo uloženo testovací prostředí (připravené testerem). Toto prostředí by mělo připravenou složkovou hierarchii, potřebný software, ovladače, pluginy a jiné věci.

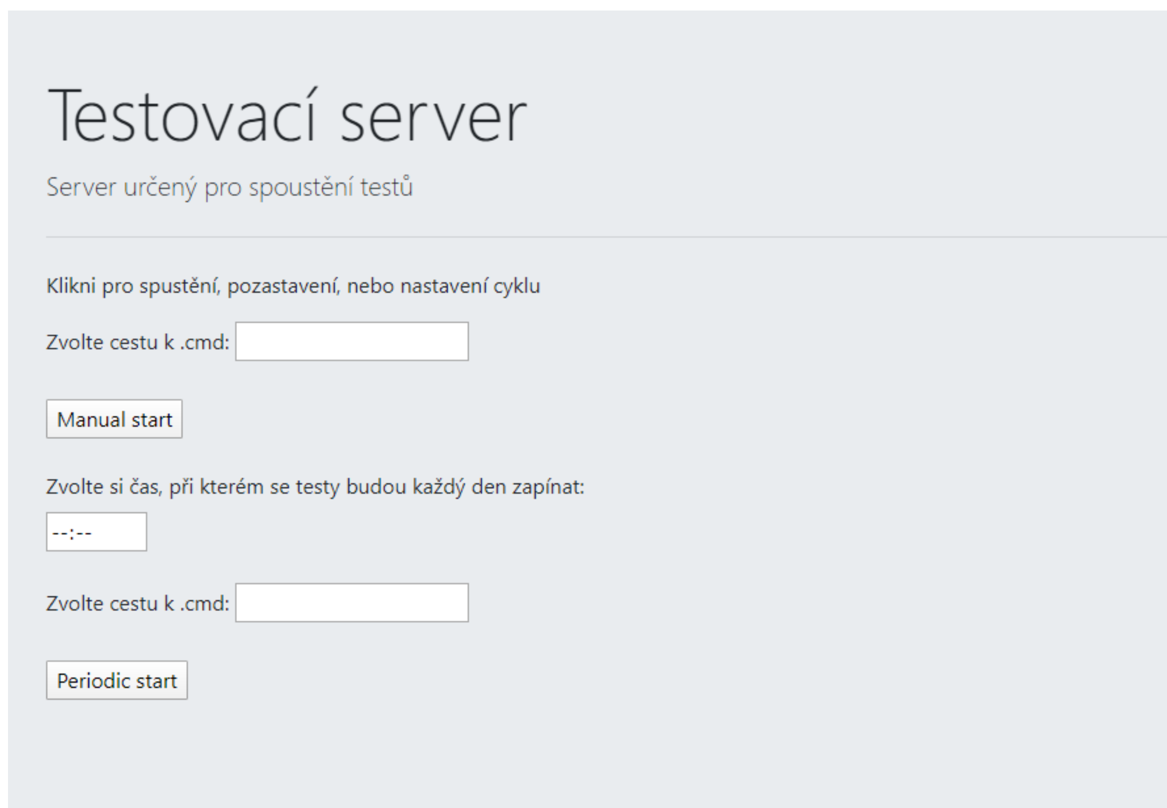
Důležitou součástí celého systému by byla služba Jenkins, která slouží pro automatizování serveru. Jenkins umí poměrně dobře spolupracovat s gitovými systémy. Proto by bylo po domluvě nastaveno, že se celý proces spustí například po pushnutí nového buildu testované aplikace, nebo po pushnutí testera na konkrétní branch (aktualizace, nebo přídaní testů), atd. Úkolem Jenkinsu je dále zautomatizovat celý proces vykonávání testů na serveru, od spuštění testů, přes úklid (a všechny další operace). Tyto kroky se nastavují v jednotlivých stages pomocí Jenkins jazyka (pipelines) [34].

Toto řešení je velmi výhodné, ale hodně časově náročné, proto se hodí hlavně u větších a dlouhotrvajících projektů.

9.2 Řešení serveru pro tuto práci

Naše řešení bude jednodušší a založené na nástrojích, které jsou dostupné zdarma. Využijeme lokální webový server. Pomocí něj se testy budou spouštět. Toto řešení je o trochu méně efektivní než to předchozí, ale splní svůj účel dokonale a pomůže nám zautomatizovat téměř celý proces.

Základem bude webové rozhraní vytvořené pomocí HTML, PHP a Bootstrapu. Bude mít za úkol spustit celý testovací proces po přidání cesty k .cmd commandu. Dále budeme moci ovládat, jak často se testy budou spouštět (volba času). Výsledek práce můžeme vidět na obrázku č.36 níže.



Obr. 36. Webový server pro ovládání testů

Vidíme, že se jedná o velmi jednoduché a hlavně rozšiřitelné rozhraní. Můžeme si samozřejmě zvolit i jiné ovládací prvky, jako cestu k virtuálnímu prostředí, ze kterého se testy budou spouštět, jiné časové intervaly, popřípadě podmínky, kdy se testy mají spouštět.

Dalším důležitým krokem, který je částečně udělán v kódu a částečně se musí provést manuálně je vytvoření složkové a souborové hierarchie:

Pro případné chyby

V tomto případě bude složka vytvořena automaticky našim rozhraním a je pojmenována podle času spuštění testů. Uvnitř je výsledný XML a HTML soubor (pro debugging) a složka Error_Picture, do které v případě chyby budou uloženy printscreeny.

Pro spuštění testů, řetězce pro porovnávání, pomocné soubory

Asi nejdůležitější složka, pojmenovaná UITestEntry, obsahuje všechny prostředky, pomocí kterých můžeme spustit testy mimo Visual studio příkazovým řádkem. Obsahuje spustitelnou Nunit test consoly, pomocí které, jak uvidíme, zahájíme všechny testy (a která generuje důležitý XML výstup). Dále složka result, do které se uloží XML a HTML. Následuje SW pro hledání UI elementů v aplikaci (UITestVerify). Poté jsou zde testovací textové soubory pro porovnávání výsledků a také .cmd soubor, který zahájí všechny testy a posléze spustí .exe soubor našeho XmlToHtmlTransformeru, který rovněž musíme mít umístěný ve složce. V té se pak dále nachází i celé vybuildované řešení, tedy FWHandler knihovna, testy a všechny ostatní komponenty.

Výsledný příkaz vypadá následovně:

```
C:\UITestEntry\bin\net35\nunit3-console.exe C:\UITestEntry\UITestBin\UITests.dll --result=C:\UITestEntry\Result\TestResult.xml
C:\UITestEntry\UITestBin\XmlToHtmlHandler.exe
                        -Input Xml C:\UITestEntry\Result\TestResult.xml
                        -OutputHtml C:\UITestEntry\Result\TestResult.html
                        -FileVersion x -ServerPath x
                        -ResultDir C:\UITestEntry\Result
                        -LocalPath C:\UITestResult\
                        -PathToHtmlResultLocal C:\UITestEntry\Result\TestResult.html
                        -MailPath smtp.gmail.com
                        -MailFrom xx -Password xx
```

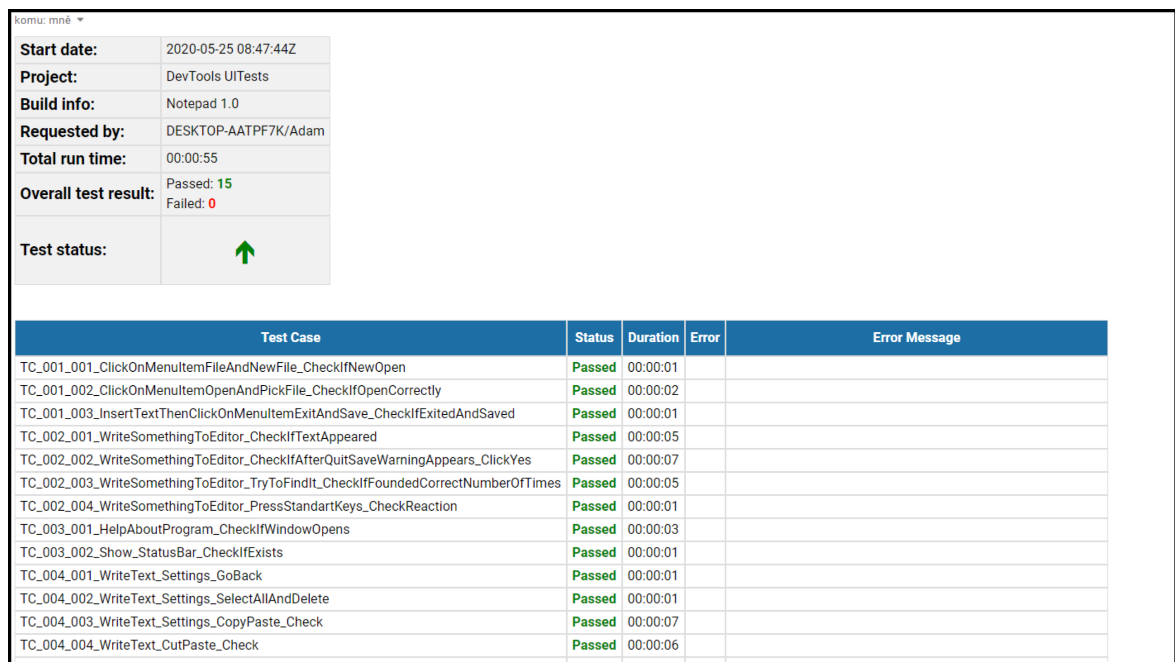
Obr. 37. Spouštěcí příkaz

První příkaz specifikuje cestu k Nunit consoly (spouštěcí bod), prvním parametrem jsou samotné testy v .dll souboru. Druhým parametrem je výstupní XML soubor.

Druhý příkaz specifikuje cestu k XmlToHtmlHandleru. Zde už je obrovské množství parametrů (mnoho jich je volitelných). Důležitá je cesta k vygenerovanému XML souboru z předchozího příkazu, výstup pro HTML soubor a maily, pro zaslání výstupu příslušným osobám.

Složková hierarchie a funkční příkaz jsou velmi důležité pro testovací prostředí, ovládané serverem. Protože nebyl použit jenkins, bylo vytvořeno více podobných příkazů, které provádí podpůrné operace.

Server a vše potřebné je připraveno, nyní vybuildujeme celý projekt, vložíme cestu a klikneme na tlačítko Manul start z obr. č. 36. Po skončení testů došel tento email:



komu: mně

Start date:	2020-05-25 08:47:44Z
Project:	DevTools UITests
Build info:	Notepad 1.0
Requested by:	DESKTOP-AATPF7K/Adam
Total run time:	00:00:55
Overall test result:	Passed: 15 Failed: 0
Test status:	

Test Case	Status	Duration	Error	Error Message
TC_001_001_ClickOnMenuItemFileAndNewFile_CheckIfNewOpen	Passed	00:00:01		
TC_001_002_ClickOnMenuItemOpenAndPickFile_CheckIfOpenCorrectly	Passed	00:00:02		
TC_001_003_InsertTextThenClickOnMenuItemExitAndSave_CheckIfExitedAndSaved	Passed	00:00:01		
TC_002_001_WriteSomethingToEditor_CheckIfTextAppeared	Passed	00:00:05		
TC_002_002_WriteSomethingToEditor_CheckIfAfterQuitSaveWarningAppears_ClickYes	Passed	00:00:07		
TC_002_003_WriteSomethingToEditor_TryToFindIt_CheckIfFoundedCorrectNumberOfTimes	Passed	00:00:05		
TC_002_004_WriteSomethingToEditor_PressStandartKeys_CheckReaction	Passed	00:00:01		
TC_003_001_HelpAboutProgram_CheckIfWindowOpens	Passed	00:00:03		
TC_003_002_Show_StatusBar_CheckIfExists	Passed	00:00:01		
TC_004_001_WriteText_Settings_GoBack	Passed	00:00:01		
TC_004_002_WriteText_Settings_SelectAllAndDelete	Passed	00:00:01		
TC_004_003_WriteText_Settings_CopyPaste_Check	Passed	00:00:07		
TC_004_004_WriteText_CutPaste_Check	Passed	00:00:06		

Obr. 38. Výsledný mail

Vidíme, že všechny vytvořené testy prošly a proběhly poměrně rychle. Nyní si můžeme nastavit časový interval zahájení testů. Pokud bude server online, tak se každý den proces spustí a po skončení testů obdržíme výsledky. V případě přidání nových testů a vybuildování celého řešení, by to server vzal v potaz a spustil aktualizované testy. Proces je v tomto bodě kompletně automatizovaný a bezpečný.

10 DOSAŽENÉ VÝSLEDKY A MOŽNOSTI POUŽITÍ

Tato kapitola se věnuje zhodnocení celé práce a všech jejich výsledků. Dále rozebírá možnosti použití, od těch předpokládaných, po ty potencionální.

Výběr frameworku

Výběr správného frameworku, který bude efektivně komunikovat s Microsoft knihovnami, byl asi nejdůležitějším rozhodnutím pro tento projekt. Bylo zde k dispozici více knihoven, které umožňovaly stejnou funkcionalitu, ale obecně na tom FlaUI framework byl nejlépe, proto byl na počátku vybrán.

Vyplatil se tedy výběr FlaUI? Z dosažených výsledků konstatuji, že ano. Programování ve FlaUI probíhalo poměrně hladce a bylo intuitivní. Vyžadovalo ale hodně výzkumu ohledně stromové struktury aplikací a způsobu, kterým se UI prvky v této struktuře vyhledávájí (dále pokročilé OOP a návrhové vzory). Naštěstí FlaUI poskytlo spoustu důmyslných nástrojů, jak tyto problémy řešit i v aplikacích, které díky nevhodné architektuře, popřípadě špatnému systému identifikátorů, byly hůře testovatelné, což považuji za obrovskou výhodu.

FlaUI bylo z velké části prozkoumáno a značná část jeho komponentů využita v dalších místech, jako systém reportování.

FlaUI samo o sobě má ohromný potenciál v oblasti testování (hlavně automatizovaného) WPF, WF aplikací, ale jeho použití má mnohem větší rozsah. Jedná se o testování webu, nebo automatizaci různých procedur. Obecně pomocí FlaUI můžeme zautomatizovat cokoliv a ulehčit si tak život.

Pomocná knihovna FWHandler

I přes všechny výhody FlaUI bylo psaní testů v něm poměrně náročné pro lidi, kteří nemají zkušenost s pokročilými znalostmi programování. Při větších TC, byly přepsané testy do kódu velmi rozsáhlé a obsahovaly mnoho duplicitního kódu. Navíc testy vždy postrádaly kvůli chybám (ne na straně FlaUI) obecnost a jeden test, dělající stejnou věc, mohl vypadat jinak, v závislosti na identifikátorech aplikace, což je nevýhoda pro čitelnost testu. V důsledku toho bylo rozhodnuto implementovat knihovnu FWHandler, jejichž úkolem je tyto nedostatky z velké části vyřešit. FWHandler vytváří nádstavbu nad FlaUI a poskytuje jednoduché API. To se skládá z jednoduchých příkazů, které tester volá a tím komunikuje

přímo s FlaUI. Tato komunikace s FlaUI je samozřejmě vylepšena o funkcionalitu, která má vyřešit výše zmíněné obtíže.

Otázkou je, jestli taková knihovna byla vůbec zapotřebí. Z nynějších poznatků zatím vyplývá, že skutečně dokáže velmi zjednodušit a zpříjemnit vytváření testů. Při porovnání kódu jednoho testu z FlaUI a z FWHandleru je poznat obrovský rozdíl, který někdy čítá i dvojnásobek řádku navíc pro FlaUI. Výsledkem je, že s FlaUI musíte nejen psát více řádků kódu, ale zároveň musíte pojistit každý neobvyklý případ v každém testu, což je oproti FWHandleru obrovská nevýhoda.

Využití knihovny může být opět velké. Její další výhodou je to, že obsahuje možnost “od-pojit” stěžejní framework (FlaUI) a nahradit ho jiným. V případě nutnosti by jej tedy šlo využít i na testování webových stránek, mobilních aplikací a dalších platforem, musela by se pouze dodělat funkcionalita pro daný framework, ale funkce knihovny by zůstala nezměněna. Knihovna dokonce obsahuje i lokální logger (na reporty), aby poskytovala nějaký výstup i sama o sobě.

Systém reportování

Dalším komponentem je systém reportování. Základním cílem bylo, aby člověk, který spustil testy, vždy obdržel výstup, který bude v čitelné podobě a s jehož pomocí lehce odhalí chybu a nedostatky dané aplikace. Tímto komponentem se stal XmlToHtmlHandler. Ten v závislosti na Nunit Console Runneru (pomocí kterého jsou testy spuštěny) vygeneruje XML soubor, provede transformaci na HTML, do čitelné podoby. Tato podoba obsahuje souhrn všech sad, testů, výsledků a dalších informací. Důležitým bodem zde bylo, aby chyba, která se objeví na kterékoliv vrstvě systému, doputovala korektně na výstup a informovalo uživatele. V neposlední řadě řeší odeslání zprávy příslušným osobám.

Všechny tyto body se povedlo splnit a knihovna XmlToHtmlHandler se stala neoddělitelnou součástí celého systému. Od prvního spuštění knihovny se celková práce hledání chyb zrychlila. Jedinou nevýhodou je, že v některých velmi atypických případech může být chyba z reportu hůře čitelná (protože ji nelze předpovědět, a stack trace nepostačuje). Ve většině případů funguje korektně.

Její využití je rozsáhle, dá se použít prakticky ve všech případech, kdy se vytvářejí automatizované testy (nebo kdekoliv jinde, kde z XML vytváříme HTML report). Jediné, co se musí změnit, je výsledná podoba XSLT souboru (který čte z XML), což by nemělo

být vůbec náročné. Dá se používat sama o sobě, ale také se velmi vhodně dá zkombinovat s FWHandlerem a s FlaUI.

Celková architektura

Všechny zmíněné komponenty do sebe zapadly a tvoří ucelenou architekturu pro automatizované testy a jiné operace. Architektura funguje hlavně jako celek, ale jak už bylo řečeno, její jednotlivé součásti se velmi dobře dají používat i sami o sobě na příslušné úkoly. Hlavní využití této architektury je na UI testy, ale teoreticky má velký potenciál i pro backendové testy. Musela by se vytvořit jednoduchá knihovna, která by se vložila místo FWHandleru. Ta by zprostředkovávala provádění příkazů přes příkazový řádek a na konci by se opět vygeneroval report. Jednalo by se tedy pouze o zastoupení úlohy FWHandleru na mnohem menší úrovni a všechna ostatní funkcionalita je už přípravená.

Vytvoření ukázkových testů

Jako ukázka výsledného snažení byla vybrána aplikace Notepad. Pro ní bylo navrhnuto 15 TC, které byly poté přepsány do kódu pomocí vytvořené knihovny (8 TC přepsáno). Zároveň bylo okolo testů vybudováno prostředí, které se snaží eliminovat veškerou duplicitu a snížit komplexitu na minimum, aby se uživatel mohl soustředit pouze na vytváření testů, ne na věci okolo. Celkový postup se tedy zjednodušil na pouhé vytvoření třídy, která bude dědit od nějaké precondition a každý test musí obsahovat metodu pro inicializaci. Poté se už píší testy ve formě jednoduchých příkazů. Celkové nároky na vytváření těchto testů jsou základní znalost angličtiny a programování, což bylo jedním z cílů, který se povedlo naplnit.

Z vytvořených testů dále vyplývá, že se skutečně velmi dobře čtou. Naučit se vytvářet a číst tyto testy je velmi jednoduché a rychlé. Zároveň byl vytvořen průzkum, jak moc se jim podařilo snížit duplicitu. Vytvořil jsem test na 25 řádků ve FlaUI, který jsem poté přepsal pomocí FWHandler knihovny na 10 řádků. U FlaUI jsem musel dále řešit každou situaci zvlášť a celkově kód byla skupina dlouhých metod a lambda funkcí, kdežto u FWHandleru jsem zavolal pouze 10 příkazů a vůbec jsem nepřemýšlel nad nějakým interním fungováním stromu prvků, pouze nad testy.

Využití je rozsáhlé, od menších projektů po ty větší. Největší smysl má ale u velkých projektů, které vyvíjí dlouhodobě nějakou aplikaci s UI. Toto UI by mělo být alespoň středně velké.

Nasazení na server

Po dokončení testů bylo důležité je nasadit na nějaký server. Tyto testy z tohoto serveru poté spouštíme, buď jednorázově, nebo v přednastaveném cyklu. Server se dále stará o řešení různých příprav pro testovací prostředí, provádí úklid a všechny ostatní operace, aby zajistil, že testerovi každý den v určitou hodinu, dojde mail s výsledkem. Tento proces má obrovské množství výhod, kromě rychlosti a efektivnosti, je to odstranění lidského faktoru.

Server je použitelný na menší projekty a projekty, které nebudou vyvíjeny nějakou extrémní dobu. Na větší a dlouhodobější projekty se spíše vyplatí řešení, které bylo nastíněno v kapitole č. 9.1 s Jenkinsem a virtuálním serverem, ale pokud nároky na server nejsou moc vysoké, vždy může být použito toto řešení, které se navíc dá velmi snadno rozšířit.

ZÁVĚR

Hlavním cílem práce bylo prozkoumat možnosti automatizovaného testování a zlepšit uživatelskou přívětivost při používání FlaUI. Výsledek práce poskytuje lepší podmínky pro vytváření testů, kód není duplicitní ani rozsáhlý, byla snížena poměrně velká komplexita FlaUI a jiných částí a zjednodušena do knihovny s názvem FWHandler. Zároveň se podařilo prokázat, že FlaUI je solidní, dobře navržený framework pro automatizované testování, není však bez chyby. Mezi jeho základní chyby patří oblast, která byla problémová u předchůdce FlaUI s názvem TestStack White. Jedná se o hledání elementů ve stromě. To může být někdy nestabilní a u některých komplikovanějších situací, vyloženě složité. Je však diskutatibilní, jedná-li se o chybu na straně FlaUI, nebo dané aplikace. Mezi další, tentokrát menší problémy, patří nefunkčnost některých, málo používaných elementů. Zde je jen otázkou času, než je někdo opraví.

Rozhraní FWHandler, které bylo vytvořeno, počítá i s možností výměny FlaUI a nahrazení jiným frameworkem. Rozhraní samo o sobě splnilo svůj účel a pomocí různých návrhových vzorů s ním bylo dosaženo příjemné práce. Stejně úspěšně se podařilo dokončit i další komponenty, například systém reportování, popřípadě další knihovny pro zjednodušení práce a všechny součásti byly umístěny do jednotného koheretního systému. Tento systém lze libovolně rozšiřovat a upravovat podle situace, projektu, firmy a dalších okolností, ale v zásadě funguje následovně. Uživatel vytvoří několik testovacích sad pro danou aplikaci. Testy se budou spouštět ručně, nebo automaticky, pomocí serveru. Po spuštění se testy budou provádět a reagovat na většinu nestandardních situací či chyb. Tyto chyby poté Nunit console zapíše do XML souboru. Ten se po skončení převede na HTML pomocí XmlToHtmlHandleru, uloží do souborové hierarchie a odešle mailem příslušné osobě. Ta si report přečtě, v případě chyby na ni zareaguje a v nejlepším případě i opraví. Tím je cyklus skončen a opakuje se každý den. Výše zmíněný cyklus se ukázal jako nejlépe vyhovující a podařilo se díky němu odhalit nezanedbatelné množství chyb, nebo nedokonalostí, na které by se ručně přicházelo velmi pomalu.

Využití rozhraní v praxi je značné. Nejvíce se však hodí na větší projekty, vyvíjející aplikaci s rozsáhlým UI. V tomto případě může být celý systém použit, tak jak byl navržen. Vytvoří se několik TS pro konkrétní verzi UI, ty se pak dlouhodobě spouští a provádějí regresní testování. Tento proces má potenciál ušetřit velké množství času, zvláště když je testů dostatek a jsou dobře navrženy. Vzhledem k tomu, že jednotlivé komponenty rozhraní nejsou pevně svázány a fungují sami o sobě, mají využití i jiné části tohoto

rozhraní. Patří sem hlavně systém reportování, který může být oddělen a použit na jakýkoliv další projekt. Dalším využitím rozhraní je možnost automatizace repetetivních úkonů.

Rozhraní je navrženo tak, aby šlo velmi snadno rozšířit o další funkcionalitu. Sem může patřit například podpora pro některé nestadardní UI prvky, které nebyly zahrnuty do této práce. Dále by mohla být zakomponována knihovna, pro backendové testy. Ta by byla nahrazena za FlaUI a celý systém by fungoval stejně. Dále by knihovna mohla být rozšířena o možnost provádět některé úkony paralelně, čímž by se zvýšila nejen rychlost prováděných testů, ale vznikly by i lepší možnosti pro řešení problému neočekávaných situací v testu.

FlaUI prokázal, že je vhodnou volbou pro výběr a že v kombinaci s pomocnou knihovnou FWHandler je ideální volbou pro vytváření UI testů.

SEZNAM POUŽITÉ LITERATURY

[1] PATTON, Ron. *Testování softwaru: Software testing, 2000*. Praha: Computer Press (CP Books), 2002. ISBN 80-7226-636-5.

[2] Types of Software Errors. *Professionalqa* [online]. 2019, 11 July 2019 [cit. 2020-03-23]. Dostupné z: https://www.professionalqa.com/types-of-software-error

[3] ABEROUCH, Abdallah. The 5 Most Infamous Software Bugs in History: Technology, inovation. *Bbvaopenmind* [online]. 2015, 02 November 2015 [cit. 2020-03-22]. Dostupné z: https://www.bbvaopenmind.com/en/technology/innovation/the-5-most-infamous-software-bugs-in-history/

[4] MCFADDEN, Christopher. The Origin of the Term 'Computer Bug': The origin and history of "Computer Bugs" is a surprisingly long and fascinating one. *Interestingengineering* [online]. 2018, 13 September 2018 [cit. 2020-03-22]. Dostupné z: https://interestingengineering.com/the-origin-of-the-term-computer-bug

[5] Y2K bug: The Y2K bug was a computer flaw, or bug, that may have caused problems when dealing with dates beyond December 31, 1999. *Nationalgeographic* [online]. 2011, 21 Jan 2011 [cit. 2020-03-23]. Dostupné z: https://www.nationalgeographic.org/encyclopedia/Y2K-bug/

[6] What is Software Testing? Introduction, Definition, Basics & Types: What is Software Testing? *Guru99* [online]. [cit. 2020-03-23]. Dostupné z: https://www.guru99.com/software-testing-introduction-importance.html

[7] Černá vs. bílá skříňka. *Test: swtestovani* [online]. 2012 [cit. 2020-03-23]. Dostupné z: http://test.swtestovani.cz/index.php?option=com_content&view=article&id=23:erna-vs-bila-skika&catid=3:zaklady&Itemid=11

[8] Testování ekvivalence. *Test: swtestovani* [online]. 2012 [cit. 2020-03-23]. Dostupné z: http://test.swtestovani.cz/index.php?option=com_content&view=article&id=28:testovani-ekvivalence&catid=13:testovaci-techniky&Itemid=29

[9] KRÁLOVÁ, Iveta. Guideline: Rozdělení tříd ekvivalencí. Metodikamezitest: asp2 [online]. 2013[cit.2020.03.23].Dostupnéz:http://metodikamezitest.asp2.cz/Metodika_testovani/guidances/guidelines/rozdeleni_trid_ekvivalenci_156BBF93.html

[10] Softwarová chyba: Vývoj softwaru. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001-, 3 November 2019 [cit. 2020-03-24]. Dostupné z: https://cs.wikipedia.org/wiki/Softwarov%C3%A1_chyba

[11] Životní cyklus informačního systému: Vývoj softwaru. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA):
Wikimedia Foundation, 2001-, 17 March 2019 [cit. 2020-03-24]. Dostuné z: https://cs.wikipedia.org/wiki/%C5%BDivotn%C3%AD_cyklus_informa%C4%8Dn%C3%ADho_syst%C3%A9mu

[12] Metodika vývoje softwaru: Vývoj softwaru. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001-, 29 October 2019 [cit. 2020-03-24]. Dostupné z: https://cs.wikipedia.org/wiki/Metodika_v%C3%BDvoje_softwaru

[13] Vodopádový model: Vodopádový model životního cyklu software (The waterfall Life cycle). *Testovanisoftwaru: manualni-testovani* [online]. [cit. 2020-03-23]. Dostupné z: http://testovanisoftwaru.cz/manualni-testovani/modely-zivotniho-cyklu-softwaru/vodopadovy-model/

[14] Spirálový model. *Testovanisoftwaru: manualni-testovani* [online]. [cit. 2020-03-23].
Dostupné z: http://testovanisoftwaru.cz/manualni-testovani/modely-zivotniho-cyklu-softwaru/spiralovy-model/

[15] BARTLETT, Jake. What is Dynamic Testing? Types, Techniques & Example: What Is Manual Testing ? *Blog.testlodge* [online]. Cardiff Road, Llandaff, Cardiff, CF5 2DP: TestLodge, 9 May 2016 [cit. 2020-03-24]. Dostupné z: https://blog.testlodge.com/what-is-manual-testing/

[16] Žáček P. Testování softwaru – Úvod do problematiky. Presentation presented at: [Univerzita Tomáše Bati ve Zlíně, fakulta informačních technologíí, 2019, Zlín, Czechia].

[17] Test Case: Testovací případ. *Testovanisoftwaru: test-case* [online]. [cit. 2020-03-24]. Dostupné z: http://testovanisoftwaru.cz/dokumentace-v-testovani/test-case/

[18] AUTOMATION TESTING Tutorial: What is, Process, Benefits & Tools: What is Automation Testing? *Guru99: automation-testing* [online]. guru 99 2020 [cit. 2020-03-25]. Dostupné z: https://www.guru99.com/automation-testing.html

[19] SINGH, Vivek. How white works? *Teststackwhite: latest* [online]. [cit. 2020-03-25]. Dostupné z: https://teststackwhite.readthedocs.io/en/latest/

[20] Robot Framework. *Robotframework* [online]. Nokia Networks, 2008 [cit. 2020-03-25]. Dostupné z: https://robotframework.org/

[21] ČÁPKA, David. Úvod do objektově orientovaného programování v C#: Objektově orientované programování. *Itnetwork: c-sharp-tutorial-uvod-do-objektove-orientovaneho-programovani* [online]. Tým ITnetwork, 24 May 2012 [cit. 2020-03-25]. Dostupné z: https://www.itnetwork.cz/csharp/oop/c-sharp-tutorial-uvod-do-objektove-orientovaneho-programovani

[22] ROUSE, Margaret. Object-oriented programming (OOP): The vital guide to modern programming languages and their uses. *Searchapparchitecture.techtarget: object-oriented-programming-OOP* [online]. TechTarget at 617-431-9200: TechTarget, 2019, January 2020 [cit. 2020-03-25]. Dostupné z: https://searchapparchitecture.techtarget.com/definition/object-oriented-programming-OOP

[23] HORDĚJČUK, Vojta. Objektově orientované programování: Informatika. *Voho: oop* [online]. 2008 [cit. 2020-03-25]. Dostupné z: http://voho.eu/wiki/oop/

[24] AGARWAL, Harsh. Inheritance in C++. *Geeksforgeeks: inheritance-in-c* [online]. Sector-136, Noida, Uttar Pradesh - 201305: geeksforgeeks [cit. 2020-03-25]. Dostupné z: https://www.geeksforgeeks.org/inheritance-in-c/

[25] C/C++: Polymorfismus - dokončení. *Builder: polymorfismus-dokonceni* [online]. Praha 2, Nové Město, Hálkova 1406/2: IDIF s.r.o, 26 February 2001 [cit. 2020-03-25]. Dostupné z: http://www.builder.cz/rubriky/c/c--/polymorfismus-dokonceni-155701cz

[26] *Návrhové vzory: [33 vzorových postupů pro objektové programování]*. 2007. Brno: Computer Press, 2007. ISBN 978-80-251-1582-4.

[27] KNESL, Jiří. Má čistý kód smysl? *Knesl: ma-cisty-kod-smysl* [online]. 2 December 2011 [cit. 2020-03-25]. Dostupné z: http://www.knesl.com/ma-cisty-kod-smysl

[28] MARTIN, Robert C. *Čistý kód: [návrhové vzory, refaktorování, testování a další techniky agilního programování]*. 2009. Brno: Computer Press, 2009. ISBN 978-80-251-2285-3.

[29] KERNIGHAN, Brian. Brian W. Kernighan: Quotes. *Goodreads: Brian_W_Kernighan* [online]. [cit. 2020-03-25]. Dostupné z: https://www.goodreads.com/author/quotes/153350.Brian_W_Kernighan

[30] KARIA, Bhavya. A quick intro to Dependency Injection: what it is, and when to use it: Tech. *Freecodecamp* [online]. freeCodeCamp, 2014, 18 October 2018 [cit. 2020-03-25]. Dostupné z: https://www.freecodecamp.org/news/a-quick-intro-to-dependency-injection-what-it-is-and-when-to-use-it-7578c84fa88f/

[31] RAYAPATI, Pavan Gopal. Singleton Class in Java. *Geeksforgeeks: singleton-class-java* [online]. geeksforgeeks, Sector-136,Noida [cit. 2020-03-25]. Dostupné z: https://www.geeksforgeeks.org/singleton-class-java/

[32] VALKOVIČ, Patrik. Factory (tovární metoda): Vzory pro vytváření. Itnetwork [online]. itnetwork, 24 November 2015 [cit. 2020-03-25]. Dostupné z: https://www.itnetwork.cz/navrh/navrhove-vzory/gof/gof-vzory-pro-vytvareni/factory

[33] Roemer. Introduction. *Github* [online]. 13 April 2016 [cit. 2020-03-25]. Dostupné z: https://github.com/FlaUI/FlaUI

[34] HELLER, Martin. What is Jenkins? The CI server explained. *Infoworld* [online]. IDG Communications, 9 March 2020 [cit. 2020-03-25]. Dostupné z: https://www.infoworld.com/article/3239666/what-is-jenkins-the-ci-server-explained.html

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

API	Application programming interface
CSS	Cascading Style Sheets
DI	Dipendency injection
HTML	Hypertext Markup Language
IDE	Integrated development environment
OOP	Object oriented programing
PHP	Hypertext Preprocessor
TC	Test case
TS	Test suit
TSW	TestStack White
UI	User interface
WPF	Windows Presentation Foundation
WF	Windows form
XML	Extensible Markup Language
XSLT	Extensible Stylesheet Language Transformations

SEZNAM OBRÁZKŮ

Obr. 1. První bug……………………………………………………………………..*14*

Obr. 2. Tabulka tříd ekvivalence …………………………………………………......*18*

Obr. 3. Důsledek softwarových chyb.…………………………………..……………*20*

Obr. 4. Schéma popisující architekturu systému a její provázanost……………..……..*49*

Obr. 5. Ukázka programu Visual UI Verify s označenými oblastmi.……………………..*51*

Obr. 6. Inicializace FlaUI a zachycení okna aplikace…………….…………………….*53*

Obr. 7. Zachycení okna aplikace odlišným způsobem (za běhu)………...………………..*54*

Obr. 8. Metoda určená pro hledání tlačítka………...……………………..…………….*55*

Obr. 9. Speciální metoda určená pro hledání všech potomků se zadaným id….………….*57*

Obr. 10. Ukázka možných operací s prvky…………………………………….……….*58*

Obr. 11. Metoda pro aktivaci klávesy na klávesnici ………………………….………*60*

Obr. 12. Výsledná třída HandlerProvider ………………………………..………..*61*

Obr. 13. Rozhraní pro TextBox element……………………..……………..………..*62*

Obr. 14. Implementace třídy TextBoxHandler……………….………..……………..*63*

Obr. 15. Enumerátor s přednastavenými hodnotami pro čekání……………………….*65*

Obr. 16. Metoda Transform pro převedení XML na HTML………………….………...*68*

Obr. 17. Ukázka XSLT souboru……………………...…………………….………...*70*

Obr. 18. Finální HTML report………………...………………………….……...……...*71*

Obr. 19. První TC……………...……………………….……...………………………..*72*

Obr. 20. Druhý TC.…………………………...…………………………………….…*73*

Obr. 21. Třetí TC……………………….…………….………...…………………………*73*

Obr. 22. Čtvrtý TC……………………………...……………………………….. .*74*

Obr. 23. Pátý TC…………………………….....…………………………………*75*

Obr. 24. Šestý TC…………………………...…………………………………..*75*

Obr. 25. Sedmý TC……………………………...…………………………………*76*

Obr. 26. Osmý TC……………………………...………………………………..*76*

Obr. 27. Kostra pro jednotlivé testy……………..……………………….………...……..*78*

Obr. 28. TC1 přepsaný do kódu……………………………………….………...……..*80*

Obr. 29. TC2 přepsaný do kódu……………………..…………….………...…….……*80*

Obr. 30. TC3 přepsaný do kódu……………………………...……………………*81*

Obr. 31. TC4 přepsaný do kódu…………………………...…………………………*81*

Obr. 32. TC5 přepsaný do kódu……………………………………………...…………*82*

Obr. 33. TC6 přepsaný do kódu…………………………………...……………………*82*

Obr. 34. TC7 přepsaný do kódu…………………………...…………………………*83*

Obr. 35. TC8 přepsaný do kódu……………………………………………...…………*83*

Obr. 36. Webový server pro ovládání testů …………...……………………..…………*85*

Obr. 37. Spouštěcí příkaz……………………………………………...…...……...*86*

Obr. 38. Výsledný mail…………………………………………...…...………..*87*

SEZNAM PŘÍLOH

Příloha P 1: Obsah vloženého zipu na CD

PŘÍLOHA P 1: OBSAH VLOŽENÉHO ZIPU NA CD

Obsah:

- Elektronická verze bakalářské práce
- Zdrojové soubory pro implementované řešení ve složce UITestHandler
- Spustitelné programy implementovaného řešení ve složce UITestEntry
- Návod ke spuštění všech řešení