VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

VYUŽITÍ DYNAMICKÝCH JAZYKŮ PRO TESTOVÁNÍ GUI NA PLATFORMĚ .NET

TESTING GUI IN .NET USING DYNAMIC LANGUAGES

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **KLÁRA FORMÁNKOVÁ**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. ONDŘEJ LENGÁL, Ph.D.**
SUPERVISOR

BRNO 2020

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2019/2020

Zadání bakalářské práce

Studentka: **Formánková Klára**
Program: Informační technologie
Název: **Využití dynamických jazyků pro testování GUI na platformě .NET**
Testing GUI in .NET Using Dynamic Languages
Kategorie: Softwarové inženýrství

Zadání:

1. Seznamte se s principy testování grafických uživatelských rozhraní (GUI). Porovnejte stávající dynamické jazyky platformy .NET a frameworky pro testování GUI.
2. Definujte požadavky na framework, který bude sloužit k automatickému testování GUI aplikací firmy VF. Dále definujte vhodné jednoduché případy užití, které budou moci být použity jako kostry pro psaní GUI testů.
3. Dle požadavků z bodu 2 proveďte návrh frameworku.
4. Navržený framework implementujte ve vybraném dynamickém jazyce.
5. Naprogramujte zvolený případ užití a napište pro něj automatické testy za použití implementovaného frameworku.
6. Vytvořte za použití frameworku automatické testy GUI aplikace firmy VF.
7. Demonstrujte funkčnost frameworku, zhodnoťte jeho přínos v procesu testování GUI ve firmě VF, a popište další možnosti pro rozšíření.

Literatura:

- Ammann, P., Offutt, J.: Introduction to Software Testing. Cambridge University Press, 2008.
- Foord, Muirhead. IronPython in Action. Manning Publications. 2009.
- Li, Wu. Effective GUI Testing Automation: Developing an Automated GUI Testing Tool. Sybex. 2008.

Pro udělení zápočtu za první semestr je požadováno:

- První tři body zadání.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Lengál Ondřej, Ing., Ph.D.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2019
Datum odevzdání: 14. května 2020
Datum schválení: 31. října 2019

Abstrakt

Tato práce se zabývá automatizací procesu testování grafického uživatelského rozhraní (GUI), konkrétně tvorbou frameworku pro testování GUI aplikací firmy VF, jehož cílem je ušetřit čas testerům a zaručit rychlejší odhalování chyb. Framework je zaměřen na testování aplikací implementovaných v jazyce C#, což v práci vedlo k prozkoumání možností využití dynamických jazyků platformy .NET a existujících nástrojů pro automatické testování GUI. Na základě zjištěných údajů je výsledný framework implementován v jazyce IronPython a s využitím prostředků frameworků White a unittest nabízí metody pro simulaci uživatelských akcí typu kliknutí na tlačítko, vyplnění textboxu apod. Mimo to framework umožňuje testovat zdroje popisků v aplikaci, ověřovat vzájemné překrývání prvků oken aplikace a využívat principů fuzz testování. Hlavním přínosem celé práce je zavedení postupů automatického testování do vývojového procesu ve firmě VF.

Abstract

The thesis deals with the topic of automation testing of graphical user interfaces (GUIs). Specifically, it handles the creation of a framework for testing GUI of applications of the VF company. The framework aims to save testers' time and guarantee faster error detection. It focuses on testing applications implemented in C#, which led to the exploration of the possibilities of using the .NET platform dynamic languages and existing tools for GUI automation testing. Based on the obtained data, the resulting framework is implemented in the IronPython language and, using the White and unittest frameworks, offers methods for simulating user actions such as clicking a button or filling a textbox. Moreover, the framework allows to test sources of the application labels, verify the overlap of application window elements and use the basic methods of fuzz testing. The main benefit of the thesis is the introduction of automation testing procedures into the development process in the VF company.

Klíčová slova

testování, grafické uživatelské rozhraní, GUI, IronPython, C#, WPF, .NET framework, White framework, unittest framework, GUI inspect tools

Keywords

testing, graphical user interface, GUI, IronPython, C#, WPF, .NET framework, White framework, unittest framework, GUI inspect tools

Citace

FORMÁNKOVÁ, Klára. *Využití dynamických jazyků pro testování GUI na platformě .NET*. Brno, 2020. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Ondřej Lengál, Ph.D.

Využití dynamických jazyků pro testování GUI na platformě .NET

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracovala samostatně pod vedením pana Ing. Ondřeje Lengála, Ph.D., uvedla jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpala.

.....
Klára Formánková
24. května 2020

Poděkování

Moje poděkování patří panu Ing. Ondřeji Lengálovi, Ph.D. za odborné vedení, ochotu a pomoc po celou dobu práce. Dále bych ráda poděkovala firmě VF, a.s. za možnost podílet se na vývoji tohoto nástroje a za spoustu užitečných rad při návrhu a implementaci.

Obsah

Kapitola 1

Úvod

Na první pohled se mnohdy může zdát, že grafické uživatelské rozhraní (*Graphical User Interface, GUI*) je jen velmi malou částí softwaru a není tedy třeba se jeho testování příliš věnovat. Na druhou stranu jde ale o vrstvu aplikace, která uživateli prezentuje celý produkt a zpřístupňuje jeho funkčnost. Pokud je funkční část aplikace mistrovským dílem ve svém oboru, ale GUI obsahuje chyby, je velmi pravděpodobné, že aplikaci v budoucnu nikdo neocení a hlavně si nenajde dostatek uživatelů. Grafické uživatelské rozhraní může být silnou zbraní pro nalákání zákazníka ke koupi produktu. GUI lze dnes najít nejen na počítačích nebo mobilních telefonech, ale i na mnoha přístrojích ulehčujících lidem každodenní život, jako je např. mikrovlnná trouba nebo pračka. Ačkoliv slovo „grafické“ v názvu grafické uživatelské rozhraní může znít honosně, jde jen o vyjádření toho, že se nejedná o textové uživatelské rozhraní, při jehož použití by aplikace byla ovládána přes textové příkazy v příkazové řádce. GUI může být často jen černobílé, velmi jednoduché a prezentované na displeji malých rozměrů. Ani v takovém případě ale neztrácí na důležitosti.

Právě zmiňované GUI je hlavním motivem mojí bakalářské práce. Zabývám se především testováním GUI a automatizací tohoto procesu. Motivací k vypracování bakalářské práce na tohle téma mi byla zkušenost s manuálním testováním GUI. Vzhledem k tomu, že testování GUI probíhá během vývoje softwaru opakovaně, je z mojí zkušenosti tento proces nejen časově náročný, ale také náchylný na přehlédnutí chyb. Pokud navíc dochází k testování jednoho produktu delší dobu, může pro člověka odpovědného za testování ztrácet na atraktivnosti.

Cílem práce je obecně zefektivnit proces testování GUI ve firmě VF, a.s. Aktuálně je veškeré testování produktů firmy prováděno manuálně, což sice má svoje výhody (jako např. posouzení úspěšnosti testu člověkem, přesnější popis nalezených chyb apod.), ale zároveň to dělá z testování velmi časově náročnou část vývoje produktu. K ušetření času při testování by měla vést alespoň částečná automatizace procesu testování. Na základě toho vzniká v rámci této bakalářské práce framework, který by měl sloužit k částečné automatizaci testování GUI. V práci se nejprve zabývám obecně principy testování, způsoby provádění testů, testováním GUI a existujícími testovacími frameworky. Následně se věnuji dynamickým jazykům platformy .NET a možnostem jejich využití. V dalších kapitolách práce je popsán návrh řešení, implementace frameworku podle vypracovaného návrhu a práci uzavírá zhodocení funkčnosti vytvořeného nástroje.

Firma VF [15], na jejíž produkty testovací framework cílí, je celosvětovým výrobcem a dodavatelem zařízení a systémů pro radiační ochranu a kontrolu. Mezi nejčastější zákazníky firmy patří jaderné elektrárny, pro které firma vyrábí např. detektory záření, různé sondy, jednotky sběru a zpracování dat, monitorovací zařízení apod. V rámci jaderné ener-

gie se zabývá také osobní dozimetrií a monitorováním kontaminace předmětů a materiálů. Produkty firmy VF dále nachází uplatnění např. v oblasti zdravotnictví nebo v kalibračních laboratořích.

Výsledkem této práce je testovací framework, který poskytuje metody pro ovládání GUI simulující práci uživatele aplikace, jako např. kliknutí na tlačítko nebo vyplnění textboxu. Dále je pomocí metod frameworku možné kontrolovat zdroje popisků v aplikaci, tedy ověřovat, zda vývojář aplikace správně použil pro popisky zdrojové soubory či nikoliv. Jiné metody frameworku umožňují testovat vzájemné přetékání jednotlivých elementů okna, což by mělo vést k odhalení chyb vzniklých např. úpravami GUI již existující aplikace nebo změnou velikosti obrazovky, na které aplikace běží. Posledním způsobem využití frameworku je fuzz testování, tzn. náhodné „klikání“ v aplikaci za účelem dosáhnout pádu aplikace. Celkově by tak automatické testy vytvořené pomocí frameworku měly za testera převzít většinu testování GUI. Úkolem testera je na začátku vývoje vytvořit testy, a ty následně používat během celého vývojového procesu, což by mělo vést k výraznému ušetření času a k rychlejšímu nalezení chyb.

Kapitola 2

Testování grafického uživatelského rozhraní

Tato kapitola se podrobně věnuje testování, přičemž zvláštní pozornost je věnována testování GUI. V první části jsou popsány základní typy testů při vývoji softwaru. Následně se zabývám principy testování GUI, způsoby provádění testů GUI a jejich využití. V závěru jsou popsány některé z existujících frameworků pro testování GUI, jejich výhody, nevýhody a možnosti použití.

Na testování GUI může být pohlíženo ze dvou úhlů. V prvním případě jde o testování samotného GUI, které je odděleno od aplikační logiky. Druhý případ nastává ve chvíli, kdy je produkt testován a kontrolován v takové podobě, v jaké s ním bude na konci vývoje pracovat uživatel, tedy včetně logiky aplikace. V této bakalářské práci se budu dále zabývat druhým z uvedených způsobů. Během testování GUI dochází k ověřování, jestli se GUI chová v souladu s požadavky na výsledný produkt a jestli funguje podle očekávání na podporovaných platformách a zařízeních. Testování GUI by mělo přijít na řadu po úspěšně vykonaných jednotkových a integračních testech, o nichž pojednává sekce 2.1. Cílem testování GUI je odhalení chyb na úrovni uživatelského rozhraní, kdy se předpokládá, že vnitřní funkce systému pracují správně, ale uživatelské rozhraní nedovolí uživateli provádět některé akce. Realita bohužel nikdy není tak jednoduchá a často je nutné otestovat GUI tak, aby byly pokryty všechny funkce systému [23].

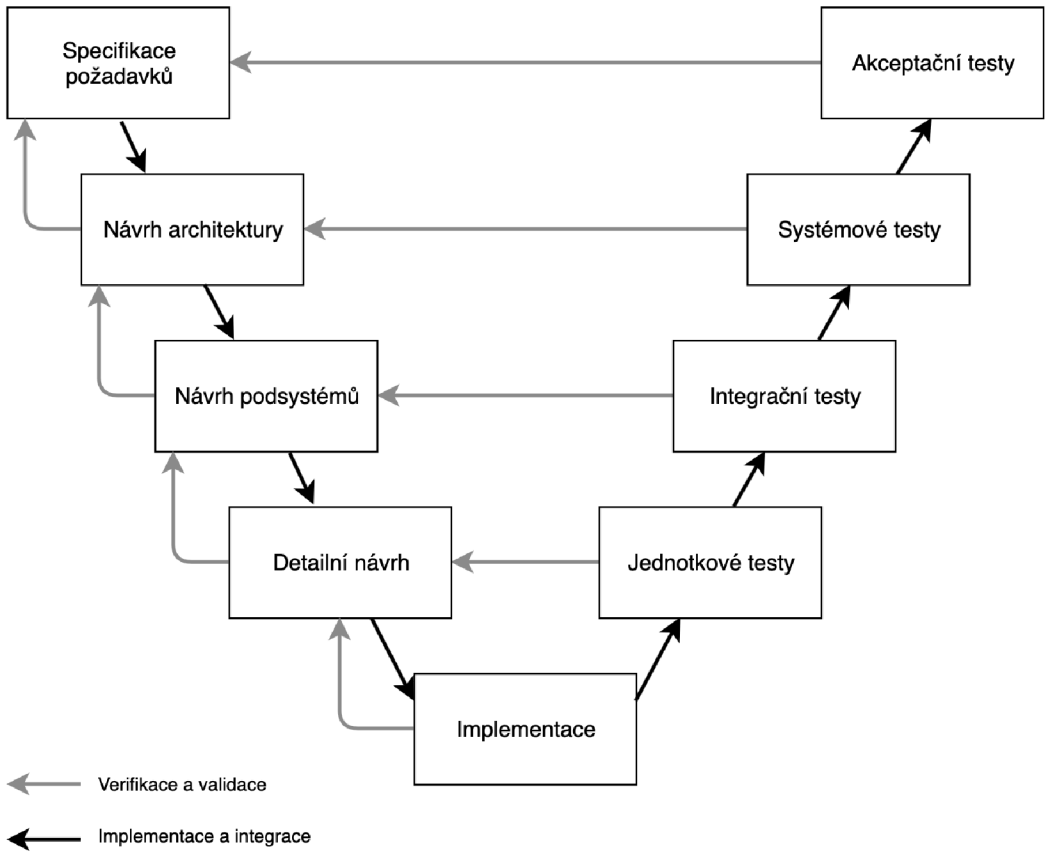
2.1 Základní typy testů

Samotné testování GUI tvoří jen velmi malou část z celého procesu testování. Během vývoje softwaru dochází k tvorbě a spouštění různých testů. Souvislosti mezi fázemi vývoje a odpovídajícími testy ukazuje V-model[1] na Obrázku 2.1 [28, s. 40]. Každá úroveň testů je zaměřena především na tu část softwaru, která vznikla na stejné úrovni. Součástí obrázku jsou černé šipky zobrazující, v jakém pořadí dochází během vývoje softwaru k implementaci a integraci jednotlivých částí. Šedé šipky ukazují, kdy během vývoje dochází k verifikaci[2] a validaci[3] výstupů různých fází vývoje. Hlavní myšlenka V-modelu poukazuje na to, že vývoj a jednotlivé testy jsou stejně důležité aktivity. Organizace modelu do dvou větví navíc

[1] V-model — grafická reprezentace životního cyklu vývoje softwaru

[2] Verifikace — proces ověřování, zda produkty určité fáze procesu vývoje softwaru splňují požadavky stanovené v předchozí fázi [18, s. 11]

[3] Validace — proces kontroly správnosti softwaru vůči zamýšlenému použití [18, s. 11]



Obrázek 2.1: V-model znázorňující životní cyklus vývoje softwaru, upraveno z [28, s. 40]

upozorňuje na to, že tyto aktivity (vývoj a testování) musí zůstat oddělené.

Dále v této sekci popisuji základní typy testů na základě informací z [28, s. 39–65].

Jednotkové testy

Jednotkové testy známé také jako Unit testy se věnují testování jednotlivých komponent neboli jednotek vznikajícího softwaru. Jako jednotky si lze představit nejmenší testovatelné části kódu. Ty se liší v závislosti na použitém programovacím jazyce, jsou to např. třídy, moduly nebo databázové skripty. Zásadní vlastností Unit testů je testování jednotek samostatně a odděleně od ostatních softwarových komponent systému. Z toho jasně vyplývá, že pokud je test jednotky neúspěšný, je chyba právě v této poměrně malé části kódu.

Hlavním úkolem jednotkových testů je určit, zda testovaná komponenta funguje správně, tedy podle požadavků detailního návrhu. To na této úrovni testování znamená kontrolu vstupů a výstupů komponenty. Dalším úkolem je otestovat robustnost jednotky. Jde o stejný přístup jako při testování funkcionality, ale na vstup jednotky jsou přivedena data, která specifikace zakazuje nebo v ní nejsou zmíněna. Správnou reakcí jednotky v těchto situacích je odpovídající reakce na výstupu, např. upozornění uživatele na nevalidní vstup.

Jednotkové testy by se měly dále věnovat také výkonu a udržitelnosti komponenty. Výkonem je myšlena např. efektivita práce s pamětí nebo čas nutný k vykonání kódu kom-

ponenty. V rámci testování udržitelnosti se testy zaměřují na vlastnosti, které definují, jak obtížné je kód komponenty modifikovat.

Integrační testy

Integrační testy jsou druhou úrovní testování a navazují na ty jednotkové. Očekává se tak, že jednotkové testy proběhly a nalezené chyby byly odstraněny. Prvním krokem této fáze testování je spojení jednotek do větších strukturovaných jednotek, tzv. integrace. V návaznosti na to dochází k testování vzájemné spolupráce uvnitř těchto nově vzniklých subsystémů.

K integraci dochází během této fáze testování hned několikrát. Nejprve jsou jednotky spojovány do menších subsystémů, které po otestování dále rostou. Každý nově vzniklý subsystém je třeba opět testovat, a to ideálně po připojení každé jedné komponenty do subsystému.

Cílem integračních testů je odhalit chyby v rozhraních a v interakci mezi subsystémy. Testy jsou tedy zaměřeny na vnitřní rozhraní mezi komponentami. Jako poslední je v této fázi obvykle testován správný přístup do databáze a správná komunikace s dalšími komponentami nebo subsystémy.

Systémové testy

Třetí fází testování jsou systémové testy. Systémové testy kontrolují, zda integrovaný produkt vyhovuje požadavkům na systém. Na první pohled se může zdát, že tyto testy proběhly už během integračního testování, to ale není pravda. Jak vychází z V-modelu, testování v předchozí fázi ověřovalo, že integrované subsystémy vyhovují návrhu podsystémů. Zkoumalo tedy technické požadavky na systém. Systémové testování se věnuje návrhu architektury systému a jeho funkčním i nefunkčním požadavkům.

Systém je testován jako celek v takovém prostředí, pro které byl navržen, tzn. na příslušném hardwarovém vybavení a případně i s nainstalovanými dalšími softwarovými produkty. V systémech využívajících databázi nebo obecně velké množství dat by měl být kladen důraz na kvalitu dat, tedy na jejich konzistenci, úplnost a aktuálnost, což by mělo být testováno právě systémovými testy.

Cílem systémových testů je ověřit, jestli vytvořený systém jako celek splňuje specifikované požadavky, a jak dobře to dělá. Mělo by dojít k odhalení chyb vyplývajících z nesprávné, neúplné nebo nekonzistentní implementace požadavků. Stejně tak by měly být objeveny nedokumentované nebo zapomenuté požadavky.

Akceptační testy

Akceptační testy jsou poslední fází testování, ale během vývoje mohou být prováděny vícekrát. Testování je zaměřeno na zákazníka, čímž se liší od předchozích testovacích fází. Zákazník se na testech aktivně podílí a často za testy přebírá zodpovědnost.

Akceptační testy probíhají podle předem připravených scénářů, které vznikly ve spolupráci zákazníka s dodavatelem. Na základě chování systému v situacích daných scénáři jsou testy vyhodnoceny jako úspěšné nebo nikoli. Pokud akceptační testování skončí jako neúspěšné, je obvykle dodavatel povinen systém upravit tak, aby mohly proběhnout další akceptační testy a jejich výsledek byl s co největší pravděpodobností úspěšný.

Otázkou je, do jaké míry je třeba vzniklý produkt testovat. Odpověď je pro každý systém odlišná. Pro systémy „na míru“ zákazníkovi je akceptační testování nutné a je třeba jej provádět důkladně a věnovat mu dostatek času. Pokud jde ale o standardní software komuni-

kující především s jinými systémy, je třeba testovat komunikaci těchto systémů a akceptační testování se zákazníkem tedy není žádoucí. Většina systémů se ale řadí někam mezi tyto dva extrémy a určit tak, kolik času věnovat akceptačnímu testování, není jednoduché.

Regresní testy

Regresní testování [25, s. 427–429] se odlišuje od předchozích tím, že není přímo spojeno jen s jednou fází vývoje softwaru. To je důvod, proč není součástí V-modelu na Obrázku 2.1. K regresnímu testování může docházet ve všech fázích vývoje, a to především při úpravách a vydávání nových verzí.

Při regresním testování je ověřováno, že vyvíjený a dříve testovaný software funguje stále stejným způsobem i po změně některé z jeho částí. Je totiž možné, že ačkoliv došlo k úpravě a vylepšení jedné části systému, přestala správně fungovat část jiná. Jednotlivé regresní testy by měly vznikat postupně zároveň s přidáváním nových funkcionalit. Seskupením těchto testů vzniká regresní sada, která může sloužit k opakovanému testování.

Další způsoby testování

Mimo výše uvedené typy testů, které přímo vychází z vývoje softwaru a jeho fází, existuje mnoho dalších variant dělení testů [18, s. 21–22]. Jednou z nich je např. rozdělení testů podle toho, zda je pro provádění testů nutné spuštění testovaného programu. Podle toho je testování označováno buď jako statické, nebo jako dynamické. U statického při testování nedochází ke spouštění programu (např. analýza zdrojového kódu), u dynamického je naopak spuštění testovaného programu nutné (např. testování hraničních hodnot vstupů programu). Další z variant dělí testování na black-box a white-box, česky také označované jako černá a bílá skříňka. Jde o rozdělení na základě úrovně znalosti zdrojového kódu testovaného programu, kterou má tester[4] při tvorbě testů. První z jmenovaných, black-box, je způsob, kdy znalost testera zahrnuje pouze externí popis softwaru, což představuje specifikaci vstupů a očekávaných výstupů programu. Na rozdíl od toho při white-box testování zná tester zdrojový kód programu, ze kterého může při tvorbě testů vycházet.

2.2 Principy testování GUI

V moderních softwarových systémech je GUI obvykle věnována více než polovina celkového rozsahu zdrojových kódů. Z toho vyplývá, že testování grafického uživatelského rozhraní je rozsáhlou činností, která může být velmi časově i finančně náročná. Testování GUI lze rozdělit do dvou tříd: testování použitelnosti a funkčnosti. První z jmenovaných testuje, jestli je aplikace použitelná, pochopitelná pro lidi a jestli je práce s uživatelským rozhraním efektivní. Použitelnost GUI je pro aplikaci velmi důležitá, v této bakalářské práci se ale budu zabývat druhým způsobem testování GUI — testováním funkčnosti.

Testování funkčnosti ověřuje, zda uživatelské rozhraní pracuje podle očekávání. V rámci testování funkčnosti GUI jsou kontrolovány ovládací prvky aplikace jako jsou tlačítka, ikony, dialogová okna atd. Podle grafické podoby aplikace lze definovat různé posloupnosti kroků, k jejichž provedení by mělo v rámci testování dojít. Tyto kroky jsou typicky kliknutí na tlačítko, otevření menu nebo zadání textu. Jednotlivé testy jsou potom reprezentovány posloupnostmi těchto kroků a obvykle vedou k vykonání akce s předem daným cílem. Tímto způsobem by mělo dojít k otestování všech proveditelných akcí v rámci GUI. Je zřejmé,

[4] Tester — člověk odpovědný za testování produktu vyvíjeného firmou

že pokud jde o rozsáhlou aplikaci, je i testování GUI rozsáhlý proces. Testování GUI lze provádět dvěma způsoby: manuální a automatické testování. Za nejlepší řešení je obvykle považována kombinace těchto dvou přístupů [18, s. 260–261].

Manuální testování

Obecně lze manuální testování označit za způsob testování, kdy jsou testy prováděny ručně. Z toho vyplývá, že v rámci manuálního testování tester postupně „proklikává“ aplikaci a kontroluje, zda se systém chová očekávaným způsobem. Klikání v aplikaci ale není náhodné. Většinou se při tomto procesu tester řídí předem sestavenými testovacími případy a výsledky testování průběžně zaznamenává do testovacích protokolů. Testovací případy jsou sestaveny na základě obchodních požadavků a jejich tvorbou se zabývá sám tester. To vše provádí tester bez použití jakýchkoli nástrojů pro automatizované testování softwaru.
Výhody manuálního testování jsou následující [27]:

- využití lidského úsudku a intuice v rámci vyhodnocení úspěšnosti testu,
- přesnější popis výsledků testu slovy testera,
- rychlejší testování při malých změnách a
- odpadá nutnost utrácet za automatizační nástroje.

Nevýhody manuálního testování jsou následující [27]:

- větší náchylnost k chybám díky lidskému faktoru,
- časová náročnost testů a
- finanční náročnost testů.

Automatické testování

V rámci automatického testování je snaha ruční „klikání“ testera nahradit testovacími skripty a dosáhnout tak automatizace provádění testů. Testovací skripty vytváří sám tester za pomoci vhodných automatizačních nástrojů. Tato fáze automatického testování je časově náročná. Jejím výsledkem je ale ušetření velkého množství času při samotném testování. Pokud totiž tester napíše testovací skripty pokrývající případy užití softwaru, je jeho práce téměř u konce. Testování od té chvíle nevyžaduje ruční zásah. Automatické testy jsou obvykle spouštěny opakovaně, např. před vydáním nové verze softwaru. Cílem automatizace je dokončit provedení testu za kratší dobu.
Výhody automatického testování jsou následující [18, s. 10]:

- úspora času,
- nízká náchylnost k chybám,
- možnost opakovaného použití testů a
- atraktivnější způsob testování pro testery.

Nevýhody automatického testování jsou následující [18, s. 10]:

- chybí posouzení úspěšnosti testu člověkem,

- nákladná údržba testovacích skriptů a
- nákladné pořízení/vytvoření automatizačních nástrojů.

Automatické testování může být prováděno podle mnoha různých principů, které se odvíjí od použité technologie při vývoji aplikace, operačního systému, typu aplikace apod. Zároveň mohou být automatizovány pouze určité části procesu testování, od čehož se odvíjí podíl manuálního testování aplikace.

Fuzz testování [22] neboli *fuzzing* je jedním z typů automatického testování, který je často využíván nejen při testování GUI. Jde o proces hledání bezpečnostních chyb pomocí automaticky generovaných a spouštěných testů. Úkolem fuzzingu v rámci testování GUI je opakovaně spouštět testovanou aplikaci s různými typy možných vstupů aplikace, jako např. importování souborů, vstup dat skrz formulář nebo třeba kliknutí na tlačítko. Cílem fuzz testování je pomocí náhodně generovaných vstupů najít v aplikaci chyby, které způsobují její pád. Jinak řečeno, fuzz testování vykonává náhodné kroky, čímž se snaží aplikaci „rozbít".

Fuzz testování je považováno za velmi efektivní způsob hledání bezpečnostních chyb, a proto vznikla spousta nástrojů tzv. *fuzzerů*, které tento typ testování aplikací umožňují. Jedním z neznámějších je nástroj *American Fuzzy Lop*. Generování vstupů aplikace ale nemusí být úplně náhodné. Fuzzing může mít různé podoby, které závisí na tom, s jakými znalostmi o aplikaci budou vstupy generovány. Typicky jde o využití informací o vnitřní struktuře aplikace, o struktuře vstupních dat nebo o míře pokrytí již provedenými testy. Obecně ale neplatí, že čím více znalostí o aplikaci fuzzer má, tím dříve najde chybu. Naopak někdy může být náhodné testování úspěšnějším způsobem.

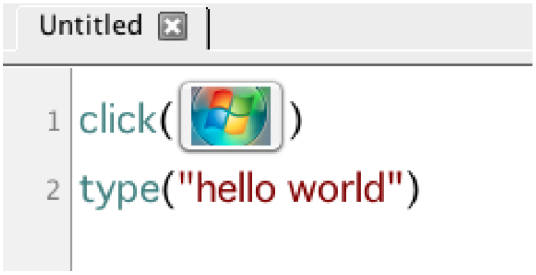
2.3 Frameworky pro testování GUI

Obecně framework označuje pomocný nástroj programátora. Významnou roli hraje ale i v oblasti testování a využití nachází především v rámci automatického testování. Framework pro testování [17] lze chápat jako soubor pravidel pro návrh a tvorbu testovacích případů. Framework je obvykle složen z více souborů a nástrojů (např. knihoven), které se věnují jedné společné problematice. Důvodů pro vznik testovacích frameworků je hned několik, mezi ty nejvýznamnější patří: vyšší rychlost a efektivita testování, lepší přesnost nebo nižší náklady na údržbu testu. K zajištění těchto výhod přispívá především znovupoužitelnost testovacích skriptů, kterou typicky framework zajišťuje.

Příklady testovacích frameworků

Selenium [10] je jeden z nejznámějších a nejpoužívanějších open-source frameworků pro testování GUI a webových aplikací. Použitím frameworku Selenium je uživateli umožněno simulovat běžné činnosti, které ve výsledné podobě produktu provádí koncový uživatel. Tím jsou myšleny akce typu vyplnění textových polí, zaškrtnutí políček, výběr hodnoty z nabídky, kliknutí, pohyb myši atd. Selenium zahrnuje celou řadu nástrojů a knihoven umožňujících automatizaci, z nichž 4 jsou považovány za základní komponenty: Selenium IDE, Selenium RC, Selenium WebDriver a Selenium Grid.

Selenium IDE nabízí první způsob psaní automatických testů. Jak vyplývá z názvu, jde o vývojové prostředí pro psaní testů. Uživatel Selenium IDE tvoří jednotlivé testy jako posloupnosti jednoduchých příkazů, např. `open https://www.overleaf.com`. Tento princip



Obrázek 2.2: Ukázka testu vytvořeného v Sikuli IDE, převzato z [11]

je velmi jednoduchý, zásadní nevýhodou je ovšem možnost spouštění testů pouze v prohlížeči Mozilla Firefox. Selenium IDE je totiž implementován jen jako rozšíření internetového prohlížeče Mozilla Firefox.

Selenium RC nabízí druhou možnost tvorby automatických testů. Tento nástroj oproti Selenium IDE umožňuje psaní testů v programovacím jazyce, který si uživatel může zvolit z řady podporovaných jazyků (C#, Java, Python, Ruby, ...). Selenium RC navíc poskytuje možnost spouštění testů na více různých prohlížečích. Nástupcem právě zmíněného Selenium RC je Selenium WebDriver. Selenium WebDriver umožňuje uživateli jednodušší tvorbu testů. Nevýhodou oproti Selenium RC je ovšem nutnost specifikovat konkrétní prohlížeč, na kterém budou následně testy spouštěny. I přesto je Selenium WebDriver aktuálně nejpoužívanějším nástrojem pro tvorbu testů ve srovnání s ostatními nástroji frameworku Selenium.

Poslední komponentou je Selenium Grid, který slouží pro distribuované testování. Poskytuje možnost testovat na více strojích a prohlížečích zároveň.

Sikuli [11] je populární open-source framework pro automatizaci testování GUI. Jeho vývoj sice od roku 2015 aktivně neprobíhá, framework je ale stále dobře použitelný. Sikuli je využíván pro testování webových a desktopových aplikací a vhodný je obzvlášť v situacích, kdy při testování není známý zdrojový kód testovaného software.

Sikuli pracuje s metodou rozpoznávání obrazu, která slouží k identifikaci jednotlivých prvků obrazovky GUI. Před začátkem testování je třeba, aby každý grafický element aplikace měl uvnitř projektu uloženou svoji grafickou podobu. V rámci testování pomocí Sikuli jsou potom tvořeny snímky obrazovky, které jsou následně srovnávány s původně uloženými obrázky grafických elementů.

Framework je vytvořen jako knihovna jazyka Jython[5]. Testy jsou tak psány se syntaxí jazyka Python, jehož znalost je pro použití frameworku nutná. Pro automatizaci jednoduchých úkolů ale stačí znát pouze základní konstrukce tohoto jazyka. Příklad použití frameworku je ukázán na Obrázku 2.2. Jde o test, jehož úkolem je v operačním systému Windows otevřít menu *Start* a do vyhledávacího pole napsat „hello world“.

White [13] je open-source framework zaměřený na testování klientských aplikací, které jsou založeny na technologiích Win32, WinForms, WPF, Silverlight a SWT. White je implementován v jazyce C#[6], což z něj činí testovací framework pro platformu .NET. Tvořit testy pomocí frameworku White je proto možné v jakémkoliv jazyce platformy .NET. White

[5] Jython — implementace jazyka Python v jazyce Java
[6] C# — programovací jazyk vyvinutý zároveň s platformou .NET

je založen na UI Automation frameworku společnosti Microsoft, který poskytuje rozhraní pro přístup k jednotlivým elementům uživatelského rozhraní a umožňuje jejich následnou manipulaci. White je založen právě na tomto principu, navíc ale poskytuje jednoduché, objektově orientované rozhraní, které z něj dělá snadno použitelný framework pro testování GUI.

Příklad použití frameworku: jednoduchý test v jazyce C#, který v okně reprezentovaném proměnnou `window` nalezne textbox sloužící pro zadání jména:

```
1 //nalezeni elementu podle Automation Id
2 SearchCriteria name =  SearchCriteria.ByAutomationId("name");
3 TextBox textBox = window.Get(name);
4 //vepsani jmena Adam do nalezeneho textboxu
5 textBox.Text = "Adam";
```

V ukázce je element vyhledán na základě *Automation Id*[7], White ale poskytuje další možnosti vyhledávání jako např. podle názvu třídy elementu, indexu nebo textu. Tyto vlastnosti elementů může uživatel frameworku vyhledat za pomoci speciální nástrojů, které jsou nazývány „GUI inspect tools“. Takové nástroje obvykle přehledně zobrazují vlastnosti elementů jednoho okna aplikace. Existuje velké množství takových nástrojů, přičemž každý z nich má svoje výhody i nevýhody a je pak na uživateli, který z nich zvolí. Jako příklad lze uvést nástroje UISpy, Inspect, ViewWizard nebo SPY++.

Ranorex [12] je na rozdíl od předchozích frameworků komerční, což znamená, že za jeho používání musí uživatel zaplatit. Tím je ale uživateli garantována podpora frameworku firmou nebo skupinou vývojářů, která za vznikem frameworku stojí. Testovací framework Ranorex je součástí celé sady nástrojů Ranorex Studio, které uživatel po zaplacení získá. Součástí je například i vývojové prostředí Ranorex IDE.

Ranorex je framework pro automatické testování GUI desktopových, webových i mobilních aplikací. Ranorex Studio nabízí dva odlišné způsoby tvorby testů. První z nich je vhodný pro začátečníky a k jeho použití není třeba psát kód. Tento způsob funguje na principu *click-and-go*, testy jsou tedy tvořeny pouze pomocí klikání. Druhým způsobem testování je psaní testů ve vývojovém prostředí. Jde tak o tvorbu testů ve zdrojových kódech zvoleného jazyka (C#, Java, Python, Ruby, ...). Ranorex Studio využívá Selenium Grid pro paralelní testování a Selenium WebDriver pro tvorbu testů.

Maveryx [7] je automatizační nástroj, který je vhodný nejen pro testování GUI. Framework cílí na testování webových a desktopových aplikací, a to především na aplikace v jazyce Java nebo aplikace pro operační systém Android. Maveryx je v základu open-source nástroj, používání některých jeho rozšíření je ovšem zpoplatněno.

První možností, jak vytvořit testy, je psaní testů tzv. bez kódu. Testy jsou v takovém případě tvořeny v Microsoft Excel pouze pomocí vyplňování vhodných polí, jak je vidět na Obrázku 2.3. Druhá možnost nabízí psát testy v jazyce Java nebo C# v uživatelem zvoleném vývojovém prostředí. Dohromady tyto dvě možnosti zaručují, že framework může používat zkušený tester i naprostý nováček v testování. Maveryx nabízí inteligentní rozpoznávání objektů aplikace, což je výhodné při testování GUI a není proto nutné využívat

[7] Automation Id — identifikátor elementu GUI specifikovaný vývojářem aplikace

	A	B	C	D	E	F
1	Description:	Example of KDT script	Author:	Maveryx	Test Case	
2	Created:		Updated:			
3	OBJECT TYPE	OBJECT NAME	CONTAINER TYPE	CONTAINER NAME	ACTION / KEYWORD	PARAMETER #1
4	BROWSER				MAXIMIZE	
5	HYPERLINK	Buttons			CLICK	
6	BUTTON	Disabled Button			IS_DISABLED	
7	BUTTON	Button			IS_ENABLED	
8	BUTTON	Button			CLICK	
9	HTML_OBJECT	message			HAS_TEXT	You pressed Button!

Obrázek 2.3: Ukázka testu vytvořeného pomocí nástroje Maveryx v Microsoft Excel, převzato z [7]

další nástroje pro vyhledávání jednotlivých elementů GUI.

EggPlant [1] je dalším komerčním frameworkem. Využití nachází především při testování GUI a jeho automatizaci, k čemuž je přizpůsoben. Tím ale možnosti jeho využití nekončí, uživatel je za pomoci EggPlant frameworku schopen testovat celou aplikaci, tedy např. i backend nebo databázi. EggPlant není pouze framework pro testování, je vyvíjen firmou se shodným názvem, která nabízí celou řadu automatizačních nástrojů, za jejichž používání ovšem musí uživatel zaplatit.

Způsob testování spočívá v technologii vyhledávání a srovnávání obrázků, což zaručuje možnost použití frameworku pro testování jakéhokoliv softwaru, kdy nezáleží na technologii použité k vytvoření softwaru. Jde tedy o testování metodou black-box. Samotné testy jsou tvořeny testovacími skripty, které uživatel frameworku píše v jazyce SenseTalk[8].

Robot Framework [9] je open-source nástroj, který slouží pro automatizaci testování a robotickou automatizaci procesů. Jeho tvůrci slibují jednoduchou syntaxi a „humanreadable keywords“, neboli „pro člověka čitelná klíčová slova“. Pomocí klíčových slov jsou při použití Robot Frameworku tvořeny jednotlivé testy. Není přitom nutné implementovat testy výhradně pomocí klíčových slov předdefinovaných frameworkem, ale také si tvořit klíčová slova vlastní. Nástroj je nezávislý na operačním systému a technologii, která byla použita pro vývoj testované aplikace. Je implementován v jazyce Python, ale běží také v jazycích Jython (Java) a IronPython (platforma .NET).

[8] SenseTalk — skriptovací jazyk připomínající angličtinu

Kapitola 3

Dynamické jazyky platformy .NET

První část názvu kapitoly, *dynamické jazyky* [19, s. 16], lze vysvětlit jako programovací jazyky, které jsou dynamicky typované, tedy kde může vývojář měnit typ objektu, ke kterému se proměnná vztahuje. Mnoho běžných akcí, které statické jazyky provádí při kompilaci, lze u dynamických jazyků provádět za běhu programu, např. modifikace objektů nebo úpravy v typovém systému. Nelze ovšem tvrdit, že tyto vlastnosti jsou záležitostí výhradně dynamických jazyků. Dynamické jazyky ale poskytují nástroje pro jednoduché provádění právě zmíněných akcí.

Neméně důležitá je druhá část nadpisu — *platforma .NET* [26, s. 18–23]. Jde o označení softwarových produktů a technologií od firmy Microsoft. Jejím základním prvkem je *.NET Framework*, což je vývojová platforma pro aplikace cílené na technologii .NET. Je složen z modulu *Common Language Runtime (CLR)* a knihovny tříd, anglicky *Framework Class Library (FCL)*. CLR je společné jazykové prostředí pro běh aplikací. Zajišťuje mnoho systémových služeb, např. správu paměti nebo typovou kontrolu. Knihovna tříd potom poskytuje velké množství testovaného a opakovaně použitelného kódu, který může programátor využít při tvorbě vlastních aplikací.

Jazyky platformy .NET lze označit také jako jazyky s virtuálním strojem [29]. Zdrojový kód takových jazyků je v prvním kroku zkompilován do tzv. mezikódu, jinak též *Common Intermediate Language (CIL)*. Ten je následně interpretován virtuálním strojem (v případě .NET jde o CLR) a vzniká strojový kód, který může být vykonán procesorem.

Označení *dynamické jazyky platformy .NET* [19, s. 15–16] potom splňuje výše zmíněné — jde o dynamické jazyky cílené na platformu .NET. Tyto jazyky běží na modulu dynamického jazyka, anglicky *Dynamic Language Runtime (DLR)*[1], který je součástí CLR. Výhodou těchto jazyků je také fakt, že v rámci jednoho projektu mohou být díky překladu do mezikódu zdrojové kódy napsány v různých jazycích běžících na CLR.

3.1 Iron jazyky

Pod pojmem *Iron jazyky* [19, s. 11–14] si lze představit implementace jazyků, které nesou v názvu „Iron“. Jde o skupinu dynamických jazyků cílených na platformu .NET Framework. Používání jazyků z této skupiny je v rámci DLR nejvýraznější. Hlavní výhodou Iron jazyků je bezesporu možnost využívání .NET knihoven, které vývojáři značně usnadňují práci. Vyvíjení aplikací v těchto jazycích je proto rychlé a efektivní. Knihovny poskytují možnost

[1] DLR — běhové prostředí, které zjednodušuje vývoj dynamických jazyků pro platformu .NET

jednoduchého využití vícevláknového programování nebo využití Windows Forms a WPF pro vytváření atraktivních uživatelských rozhraní. Vývojáři navíc nic nebrání naprogramovat třídy v jiném, staticky kompilovaném jazyce platformy .NET, např. C#, a ty následně v kódu Iron jazyka použít.

Ve srovnání s tradičními jazyky platformy .NET je vývoj v Iron jazycích rychlejší, typicky vyžaduje méně kódu a ve výsledné podobě je čitelnější. Tyto vlastnosti z nich dělají jazyky vhodné pro prototypování a pro využití jako skriptovací jazyky. Z této skupiny jazyků se hojně využívají hlavně IronPython a IronRuby, na počátku jejichž vývoje stojí Microsoft. V dnešní době jde již o open-source projekty.

IronPython

IronPython [19, s. 4–14] je ze skupiny Iron jazyků nejpopulárnější a nejvíce využívaný. Jde o implementaci jazyka Python pro platformu .NET. IronPython lze chápat také jako .NET překladač pro programovací jazyk Python. Z toho vyplývá, že syntaxe Pythonu a IronPythonu je totožná. Soubory se zdrojovým kódem v jazyce IronPython mají příponu .py stejně jako soubory s kódem v Pythonu. IronPython je tedy kompletní implementací Pythonu. Hlavní výhodou oproti Pythonu je především už zmíněná možnost využívání .NET knihoven. Nevýhodou je naopak nemožnost jednoduše použít jiné Python knihovny.

IronRuby

IronRuby [19, s. 11–14] je implementace programovacího jazyka Ruby pro rozhraní .NET Framework. Umožňuje tak využívat všechny výhody klasické verze jazyka Ruby. Mimo to může IronRuby, stejně jako IronPython, čerpat z výhod .NET Frameworku a především jeho knihoven. IronRuby však za IronPythonem značně zaostává. Jeho přínos byl pro .NET komunitu tak malý, že je jeho vývoj od roku 2011 neaktivní [4].

IronScheme

IronScheme [5] je implementace funkcionálního programovacího jazyka Scheme cílená na platformu .NET. Za stvořením jazyka stojí jediný autor, který se při vývoji IronScheme pokusil ponaučit z chyb vzniklých při předchozím vývoji jazyka IronLisp[2]. IronScheme je z doposud uvedených jazyků nejméně používaným, což je mimo jiné způsobeno tím, že i samotný Scheme je méně používaným jazykem než Python a Ruby. Navíc k tomu přispívá i fakt, že IronScheme používá vlastní modul DLR, který je speciální úpravou klasického DLR, jenž je pro jazyk IronScheme nevhodný.

3.2 Další dynamické jazyky

Mimo výše zmíněnou skupinu Iron jazyků existují i další dynamické jazyky platformy .NET. Ačkoliv žádný z nich není tak rozšířený jako IronPython, může být užitečné znát jejich vlastnosti a možnosti využití.

[2] IronLisp — implementace funkcionálního jazyka Lisp cílená na platformu .NET

JScript.NET

JScript.NET [6] je programovací jazyk vycházející z jazyka JavaScript[3] vytvořený pro platformu .NET. Stejně jako u dříve zmíněných jazyků je jeho výhodou především možnost využívat .NET knihoven a kombinovat jeho kód s jinými jazyky platformy .NET. Zdrojový kód jazyka JScript.NET je typicky lehce čitelný, spravovatelný a rychle kompilovaný. Tvůrci je označen jako jazyk vhodný pro tvorbu konzolových, webových a grafických aplikací. JScript.NET ale nikdy nezískal velkou pozornost vývojářů a řadí se tak k jazykům, které nejsou příliš rozšířené.

MoonSharp

Dalším příkladem dynamického jazyka platformy .NET je MoonSharp [8]. Jde o interpret jazyka Lua[4] implementovaný v jazyce C#, což z něj dělá jazyk pro platformu .NET. Jeho tvůrci jej považují za jazyk z 99% kompatibilní s aktuální implementací jazyka Lua. Stejně jako Lua je jazyk MoonSharp častou volbou pro tvorbu video her, využití ale nachází také v klasických desktopových nebo mobilních aplikacích. Projekt je aktuálně open-source a jeho vývoj je stále aktivní, i přes poměrně malou skupinu uživatelů.

F#

Při popisu jazyků platformy .NET nelze nezmínit F# [2, 24]. Jazyk F# je funkcionální programovací jazyk pro platformu .NET. Ve funkcionálním programování je program považován za matematickou funkci, z čehož vyplývá zaměření na funkce a konstanty místo stavů a proměnných. Tvůrci je kód jazyka F# označován za jednoduchý, spravovatelný a robustní. Syntaxe jazyka je ovlivněna především programovacími jazyky ML a OCaml. Mezi výhody jazyka F# se řadí bohatá zásoba datových typů, která umožňuje vývojáři jednoduchou implementaci složitých řešení.

F# umožňuje využívat výhody .NET Frameworku, tedy jeho datové typy nebo knihovny. Pro programování poskytuje klasické principy funkcionálních jazyků v kombinaci s objektově orientovaným přístupem. Jeho použití je vhodné pro specifické aplikace stejně jako u jiných funkcionálních jazyků. Mezi oblasti využití jazyka F# lze jmenovat například vědecké aplikace nebo aplikace zaměřené na finančnictví. Využití však nachází i v testování, zejména v rámci jednotkových testů za pomoci existujících testovacích frameworků jako jsou NUnit nebo xUnit.

Jazyk F# je zajímavým a poměrně rozšířeným jazykem, proto je v této kapitole uveden. Řadí se ovšem mezi jazyky statické a silně typované.

[3] JavaScript — programovací jazyk typicky používaný při vývoji webových stránek a aplikací
[4] Lua — výkonný skriptovací jazyk používaný pro tvorbu počítačových her

Kapitola 4

Analýza požadavků na framework

Tato kapitola se zabývá analýzou informací o testovaných aplikacích, ze kterých následně vyplývají konkrétní požadavky na framework. Jsou zde proto popsány aktuální nedostatky procesu testování a detailně rozebrány části aplikace, které mohou být testovány automaticky.

Firma VF, na jejíž produkty vznikající framework cílí, vyrábí zařízení a systémy pro radiační ochranu a kontrolu, např. osobní dozimetry, a je tak významným výrobcem a dodavatelem nejen pro obě české jaderné elektrárny (JE Temelín a JE Dukovany), ale také jaderné elektrárny na Slovensku, Ukrajině nebo v Bulharsku. Mezi produkty firmy se řadí i software, který může sloužit například k analýze hodnot naměřených dozimetry. Software je obvykle tvořen na míru zákazníkovi a jeho potřebám, základní prvky většiny vyvíjených systémů však zůstávají stejné. Aktuálně jsou ve firmě pro testování softwaru využity výhradně postupy manuálního testování, což by měl framework do budoucna změnit a měl by tak být prvním krokem k částečné automatizaci celého procesu testování.

Analýza obecných požadavků

Hlavní nevýhodou manuálního testování GUI je časová náročnost. Tester obvykle začíná s testováním softwaru ve chvíli, kdy ještě není GUI zcela hotové, aby mohly být chyby odhaleny co nejdříve. K testování jednotlivých oken se proto vrací vždy, když dojde k opravě nalezené chyby nebo při uvolnění novější verze softwaru. Tím je způsobeno, že potřebný počet testování některých oken se může pohybovat i v řádu desítek. Časté testování stejných oken navíc zvyšuje riziko přehlédnutí chyby testerem.

Analýza nefunkčních požadavků

Firmou vyvíjený software je zpravidla implementován v jazyce C# a využívá principů WPF (*Windows Presentation Foundation*) [16]. WPF je architektura vytvořená firmou Microsoft sloužící pro tvorbu uživatelského rozhraní pro desktopové aplikace. WPF je součástí .NET Frameworku a je nástupcem předchozí architektury Windows Forms. Pro tvorbu GUI používá WPF obvykle značkovací jazyk XAML (*Extensible Application Markup Language*). Mimo to je pro vyvíjený software typické vysoké procento uživatelských oken s formuláři, a proto framework cílí právě na testování formulářů.

Analýza funkčních požadavků

Funkční požadavky vychází ze zaměření na testování formulářů. První a dle mého názoru nejzásadnější věc, která by měla být otestována, je reakce formuláře na uživatelský vstup. To vyplývá ze samotné podstaty formuláře, který typicky slouží k získávání informací od uživatele. Dále by mělo dojít k otestování správnosti překladu štítků ve formuláři. Jazyk používaný v rámci celého software je potřeba přizpůsobit požadavkům zahraničního zákazníka. To se samozřejmě týká i formulářů, jejichž štítky musí být zobrazeny ve správném jazyce. Další problém, se kterým se lze v rámci formulářů setkat, je přetékání grafických prvků aplikace přes sebe. Tento problém může být způsoben specifickými rozměry obrazovky, na které software u koncového uživatele běží. Přetečení může vzniknout také při překladu popisků a štítků v aplikaci. V rámci návrhu řešení je také potřeba definovat způsob, jakým bude framework spustitelný, jak velká část procesu bude automatizována a jakým způsobem do procesu bude mít možnost zasahovat tester. Tomu se detailně věnuje Kapitola 5.

4.1 Definice požadavků na framework

V této sekci jsou definovány konkrétní požadavky na framework, které plynou z informací uvedených v úvodu této kapitoly.

V první řadě je třeba definovat obecné požadavky na framework:

- jednoduché použití nástroje pro psaní testů a
- možnost využívat vytvořené testy opakovaně.

Dále je zapotřebí určit nefunkční požadavky na framework:

- možnost testovat aplikace implementované v jazyce C# na platformě .NET a
- možnost testovat aplikace architektury WPF.

První funkční požadavek na framework plyne z potřeby testovat reakci formuláře na různé vstupy:

- **Požadavek č.1** — Nástroj musí být schopen otestovat, zda formulář přijímá validní vstupy uživatele a správně reaguje na nevalidní vstupy uživatele.

Konkrétně jde o ověření toho, že pokud uživatel zadá do formuláře pouze validní vstupy a potvrdí jejich odeslání, formulář nevypíše žádná chybová hlášení. Naopak je očekávána zpráva o úspěšném odeslání hodnot a je zobrazen prázdný formulář pro zadání dalších hodnot, případně je okno s formulářem uzavřeno. Pokud jsou zadány nevalidní hodnoty, je očekáváno chybové hlášení.

Další funkční požadavek vznikl v důsledku používání produktů firmy VF v zahraničí:

- **Požadavek č.2** — V rámci formuláře musí být nástroj schopen ověřit, zda jsou štítky a popisky formuláře přeloženy do požadovaného jazyka (obvykle z češtiny do slovenštiny, němčiny nebo ruštiny).

Jak bylo zmíněno v úvodu této kapitoly, vyvíjený software obvykle vychází z jedné základní struktury, která je pro všechny softwarové systémy stejná. Tento základ je přizpůsoben českým zákazníkům, a proto je zapotřebí překladu. Aby byly cizojazyčné výrazy stejné pro všechny části vyvíjeného systému, znovupoužitelné a lehce dohledatelné, jsou překlady českých výrazů do jiných jazyků uloženy v překladových souborech, odkud jsou přímo používány pro zobrazení v systémech. Nástroj by tedy měl být schopen ověřit, zda všechny štítky a popisky formuláře pochází z překladových souborů a případně informovat uživatele frameworku o tomto zdroji dat. Zároveň je třeba upřesnit, že se od nástroje neočekává kontrola správnosti překladu do požadovaného jazyka. Jde čistě o kontrolu zdroje, odkud štítky pochází.

Třetí požadavek se věnuje problému přetékání prvků:

- **Požadavek č.3** — Nástroj musí být schopen otestovat, zda prvky formuláře nepřetékají přes sebe a formulář je tak dobře čitelný i při jeho úpravách.

Poslední z požadavků se týká spouštění testů vytvořených pomocí frameworku:

- **Požadavek č.4** – Nástroj musí umožňovat seskupovat testy do rozsáhlých kolekcí a jejich následné spouštění.

Kapitola 5

Návrh frameworku pro testování GUI

V této kapitole se zabývám návrhem řešení, ke kterému jsem dospěla na základě doposud získaných znalostí. V první části jsou popsány jednoduché případy užití frameworku, které si lze představit jako obecné scénáře pro tvorbu automatických testů za pomoci frameworku. V druhé části se věnuji samotnému návrhu frameworku, kde je vysvětleno, jakým způsobem by mělo dojít ke splnění jednotlivých požadavků. Celá kapitola vychází z informací uvedených v Kapitole 4.

5.1 Jednoduché případy užití

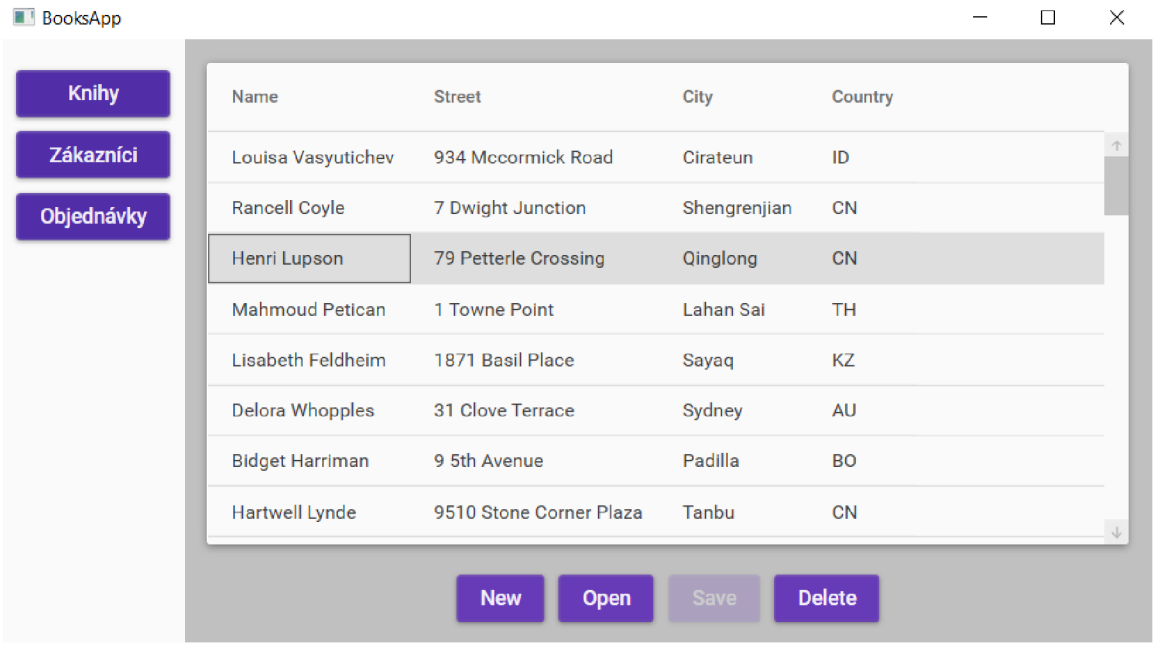
Jak vyplývá z názvu, v této kapitole jsou popsány jednoduché případy užití, které mohou následně sloužit jako kostry pro psaní GUI testů pomocí frameworku. Zároveň mohou být tyto případy užití inspirací pro implementaci konkrétních metod frameworku.

Zobrazení detailu položky v seznamu

Prvním případem užití je testování zobrazení detailu položky ze seznamu. Obecně je v takovém případě třeba zobrazit okno se seznamem, následně jednu z položek seznamu „rozkliknout“ a zkontrolovat, zda se zobrazilo nové okno, které v závislosti na specifikaci aplikace obsahuje např. formulář nebo prostý text s detailními informacemi. Pro lepší představu budu v dalším popise uvažovat okno aplikace, které je vidět na Obrázku 5.1. Pokud tedy chce tester vytvořit odpovídající test startující z hlavního okna aplikace, jsou kroky testu:

1. kliknout na tlačítko „Zákazníci“ v levém menu vedoucí na okno se seznamem zákazníků na Obrázku 5.1,
2. označit řádek zobrazeného seznamu se zákazníkem jménem „Henri Lupson“,
3. kliknout na tlačítko „Open“ pro otevření detailu vybraného řádku a
4. zjistit, zda se v okně aplikace nachází textové pole obsahující jméno „Henri Lupson“.

Po vykonání kroků by mělo dojít k otevření detailu položky s vybraným jménem. Samozřejmostí je, že výběr položky nemusí nutně probíhat na základě jména. Řádek s položkou lze označit také podle jakékoliv jiné hodnoty uvedené v seznamu nebo podle indexu řádku,



Obrázek 5.1: Okno aplikace se seznamem zákazníků

na kterém se položka nachází. V průběhu testu by měla být ideálně kontrolována správnost všech uvedených údajů v okně s detailem.

Vyplnění formuláře pro vytvoření nové položky

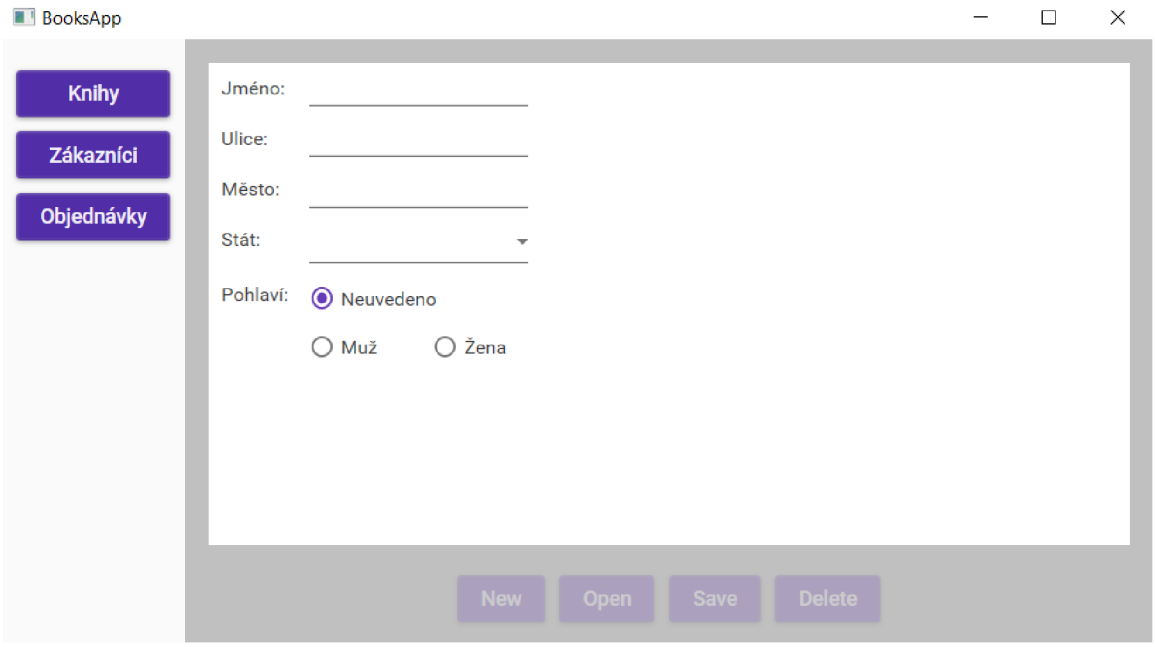
Další případ užití se věnuje formuláři. Ve formuláři dochází typicky k vyplnění textových polí, zaškrtnutí checkboxů, označení radio buttonů apod. Příklad formuláře je ukázán na Obrázku 5.2. Uvedený formulář slouží k vytvoření nového zákazníka, a pokud má být právě tento formulář otestován, je zapotřebí:

1. vyplnit textová pole pro jméno a adresu nového uživatele,
2. zvolit odpovídající radio button pro pohlaví a
3. kliknout na tlačítko „Save“ pro uložení nového uživatele.

To platí za předpokladu, že test bude spuštěn přímo v okně s formulářem. Pokud by měl test začínat z hlavního okna aplikace, musely by předcházet kroky navigující na okno s formulářem, tedy:

1. kliknutí na tlačítko „Zákazníci“ v levém menu a
2. kliknutí na tlačítko „New“ pro tvorbu nového zákazníka.

Některá pole formuláře mohou být navíc povinná a při jejich nevyplnění by mělo dojít k upozornění uživatele aplikace a neuložení nového zákazníka. I tento případ by měl být otestován. Při psaní testu by měl tester definovat přesné hodnoty, které chce do formuláře vyplnit, a na základě jejich správnosti by měl v testu také určit očekávanou reakci formuláře. V případě vyplnění všech potřebných polí korektními hodnotami a zvolení tlačítka pro



Obrázek 5.2: Formulář pro vytvoření nového zákazníka

uložení, by mělo být v rámci testu ověřeno, že opravdu došlo k přidání nového zákazníka do seznamu všech zákazníků. Pokud nebyl formulář vyplněn korektně, měl by být uživatel na tento fakt upozorněn a seznam zákazníků by měl zůstat beze změny.

Zobrazení pop-up okna při mazání položky

Třetím případem může být situace, kdy je třeba otestovat zobrazení pop-up okna při mazání zákazníka ze seznamu, který lze vidět na Obrázku 5.1. Při mazání nějaké položky bývá často očekávané chování aplikace zobrazení pop-up okna pro ujištění se, že uživatel chce položku opravdu smazat. Položka je vymazána až po potvrzení okna uživatelem. Pokud bude test spuštěn z hlavního okna aplikace, jsou jeho kroky následující:

1. kliknout na tlačítko „Zákazníci“ v levém menu,
2. označit v seznamu zákazníka se jménem „Delora Whopples“,
3. kliknout na tlačítko „Delete“,
4. ověřit, zda existuje okno s názvem „DeleteCustomer“ a
5. v pop-up okně kliknout na tlačítko „Ano“.

Poslední krok je nutný pro možnost spustit další test hned po vykonání aktuálního testu. Pokud by cílem testu bylo nejen ověřit výskyt pop-up okna, ale také otestovat, zda byla položka ze seznamu opravdu odstraněna, bylo by třeba přidat krok:

6. ověřit, že se zákazník se jménem „Delora Whopples“ v seznamu nevyskytuje.

5.2 Návrh řešení

V této části budu vycházet ze Sekce 4.1, kde byly definovány požadavky na framework. Zároveň se zde pokusím detailně popsat, jakým způsobem by těchto požadavků mělo být při implementaci dosaženo, a specifikovat způsob, jakým bude framework spustitelný, jak velká část procesu testování bude pomocí frameworku automatizována, a jakým způsobem bude mít možnost do procesu zasahovat tester.

5.2.1 Splnění požadavku č.1

Jak vychází z prvního požadavku, výsledný framework by měl být v první řadě schopný simulovat práci koncového uživatele. Tím je myšleno, že tester najde ve frameworku metody, které budou simulovat konkrétní uživatelské akce jako třeba kliknutí myší, zaškrtnutí checkboxu, vyplnění textu apod. Vzhledem k tomu, že tyto metody jsou obecného charakteru, měl by je uživatel frameworku být schopný použít pro testování jakéhokoliv okna aplikace nejen při testování formulářů, na které framework cílí.

Princip

Aby framework mohl vykonávat konkrétní uživatelské akce, je zapotřebí umět vyhledat jednotlivé elementy okna a s nimi následně pracovat. Příkladem může být akce, kdy je potřeba kliknout na tlačítko pro uložení. Implementace vlastních metod pro vyhledávání elementů by byla pravděpodobně zdlouhavá a složitá, nic by ovšem nemělo bránit použití vyhledávacích metod implementovaných ve frameworku White, jehož princip je popsán v Sekci 2.3. Díky metodám frameworku White by tlačítko pro uložení mohlo být nalezeno např. na základě jeho Automation Id:

```
1 //nalezeni tlacitka podle Automation Id
2 SearchCriteria save =  SearchCriteria.ByAutomationId("save");
3 Button btn_save = window.Get<Button>(save);
4 btn_save.Click();
```

To platí za předpokladu, že v okně reprezentovaném proměnnou `window` nějaké takové tlačítko existuje. Pro efektivní získávání Automation Id a dalších vlastností jednotlivých elementů bude využit nástroj Inspect.

Návrh implementace

Metody simulující akce koncového uživatele by měly být sjednoceny formou knihovny. K těmto metodám by měl mít tester jednoduchý přístup a měl by být schopen pomocí nich tvořit rozsáhlejší kolekce testů pro jednotlivá okna. Příkladem může být kolekce testů pro přihlášení uživatele do systému za předpokladu, že použitá metoda `user_login(login, password)` bude v knihovně implementována:

```
#test validniho prihlaseni
user_login("admin","password")
#testy nevalidniho prihlaseni
user_login("not_user","not_password")
user_login("","password")
user_login("user","")
```

Knihovní metoda `user_login` by pro představu měla obsahovat tyto kroky: vyhledání dialogového okna pro přihlášení uživatele, vyplnění vstupních hodnot metody do vhodných textových polí, stisknutí tlačítka „`OK`“/„`cancel`“ pro potvrzení/zrušení přihlášení, kontrolu provedení/neprovedení přihlášení a výpis informací o úspěšnosti/neúspěšnosti provedení testu.

5.2.2 Splnění požadavků č. 2 a 3

Druhý a třetí požadavek cílí na kontrolu prvků formulářů, konkrétně na správnost překladu štítků a nepřetékání prvků mimo vyhrazený prostor. Je zřejmé, že ověření takových vlastností je vhodné provádět buď globálně pro celou aplikaci, nebo pro určitou část aplikace, nikoliv však pro jednotlivá okna. Pokud by docházelo k testování těchto vlastností štítků v každém okně zvlášť a tester by tak byl nucen psát velmi podobné testy pro desítky oken, nebylo by to příliš efektivní. Proto je definován čtvrtý požadavek, který se zaměřuje na možnost testovat rozsáhlejší části aplikace.

Požadavek č.2

Texty štítků a popisků aplikace mohou být psány přímo v jazyce XAML jako součást zdrojového kódu okna aplikace, nebo mohou být získávány z různých zdrojových souborů s odpovídajícími překlady. V druhém z uvedených případů obsahuje zdrojový kód okna aplikace odkazy na soubory se zdroji textů štítků, přičemž každý takový soubor definuje štítky v jednom jazyce. Na základě těchto odkazů by měl být framework schopen ověřit, z jakých zdrojů pochází štítky dané aplikace, a zda nejsou psány přímo ve zdrojovém kódu okna v jazyce XAML, což je obvykle nevhodné pro správu aplikace.

Požadavek č.3

Pro každý prvek je ve formuláři vyhrazen pouze určitý prostor, který by prvek neměl přesáhnout. Může se tedy stát, že se obsah prvku do plánovaného prostoru nevejde, dojde ke zvětšení vyhrazeného prostoru a prvek začne překrývat jiný prvek. K takovému problému dochází zejména kvůli překladům štítků do jiných jazyků, kdy původní štítek zabírá menší prostor než přeložený štítek. Tento problém může být způsoben také specifickými rozměry obrazovky, na které je výsledná aplikace koncovým zákazníkem spouštěna. Kontrola přetékání prvků by měla být prováděna dynamicky za běhu aplikace porovnáváním velikostí prostorů, které zabírají jednotlivé prvky okna.

5.2.3 Splnění požadavku č. 4

Čtvrtý požadavek se týká spouštění testů. Je třeba se zamyslet, jakým způsobem je vhodné spouštět testy tak, aby byl proces testování efektivní. Je nutné zajistit automatizaci spouštění testů tak, aby tester nemusel spouštět jeden test za druhým. Naopak je třeba, aby nástroj umožňoval seskupovat testy do kolekcí a tyto kolekce mohl tester spouštět jednotlivě, či několik po sobě v rámci jednoho testování.

Tuto vlastnost frameworku by měly zajistit prostředky frameworku `unittest`, které by měly být používány při definování testů i celých kolekcí testů a také při jejich následném spouštění. Prostředky frameworku `unittest` budou využívány pro spouštění testů definovaných pomocí metod knihovny vznikající v rámci prvního požadavku. U testů vytvořených pomocí metod z požadavků č. 2 a 3 bude spouštění probíhat jiným způsobem a rozsáhlejší testování tak bude implementováno přímo uvnitř implementovaných metod.

Kapitola 6

Implementace frameworku

Jak vyplývá z názvu, obsah kapitoly se věnuje samotné implementaci frameworku. Je zřejmé, že se kapitola nemůže zabývat každým detailem zdrojového kódu, pokusím se zde ale shrnout všechny důležité a zajímavé části implementace. Zároveň se pokusím popsat, jak celý vývoj nástroje probíhal, a co mě vedlo k použití konkrétních nástrojů a technologií.

V úvodu kapitoly je vysvětleno, jakým způsobem byly vybrány použité technologie a nástroje pro vývoj frameworku. Další část se zabývá architekturou frameworku. Je zde ukázáno, jak je framework provázán s aplikací, a naznačena vnitřní struktura frameworku. Na to navazuje sekce detailně popisující nejrozsáhlejší část frameworku — knihovnu metod pro ovládání elementů GUI. Následují části věnující se implementaci testování překladů štítků, testování přetékání prvků a fuzz testování. Poslední sekce kapitoly se zabývá možností spouštět více testů zároveň.

6.1 Výběr technologií

Před začátkem samotné implementace bylo třeba rozhodnout, jaké technologie a nástroje budou v rámci vývoje frameworku použity. V této části je vysvětleno, proč jsem zvolila právě tyto technologie, a jak konkrétně budou při implementaci využity. Kromě výběru programovacího jazyka jde také o volbu dalších frameworků, které mi pomohou usnadnit práci a ušetří čas.

IronPython

Prvním rozhodnutím při vývoji frameworku byla volba implementačního jazyka. Výběr jazyků se zúžil při omezení použít dynamický jazyk platformy .NET. Proč právě jazyk platformy .NET, vychází z nefunkčních požadavků uvedených v Sekci 4.1. Vlastnost jazyka „dynamický“ z požadavků nevychází, svůj účel ale nachází při tvorbě testů a také v jejich správném průběhu. Očekává se, že tester bude vytvářet testy ve firemním vývojovém prostředí, které běží zároveň s aplikací. Pokud by testy psal ve statickém jazyce, musel by pokaždé, když implementuje nový test nebo jeho část, znovu spustit vývojové prostředí i s ním i aplikaci. Mimo to jsou testované aplikace typické generováním zdrojového kódu oken dynamicky za běhu aplikace, což také vede na omezení výběru pouze na dynamické jazyky, které umožňují mnoho běžných akcí jako např. modifikace objektů nebo úpravy v typovém systému provádět jednoduše za běhu programu.

Jaké dynamické jazyky platformy .NET jsou na výběr, shrnuje Kapitola 3. Z Iron jazyků je jasným vítězem IronPython a z dalších uvedených jazyků se jako nejvhodnější jeví

MoonSharp. Pokud bych v této práci chtěla vytvořit počítačovou hru, volba by bezpochyby padla na MoonSharp, při tvorbě testovacího frameworku se ale přikloním k implementaci v jazyce IronPython. Vede mě k tomu především fakt, že IronPython je velmi populární a rozšířený jazyk. Jeho vývoj aktivně probíhá a komunita kolem jazyka se stále rozrůstá. V návaznosti na velkou popularitu jazyka Python je navíc jeho kód pro většinu vývojářů i testerů známý a dobře čitelný.

C#

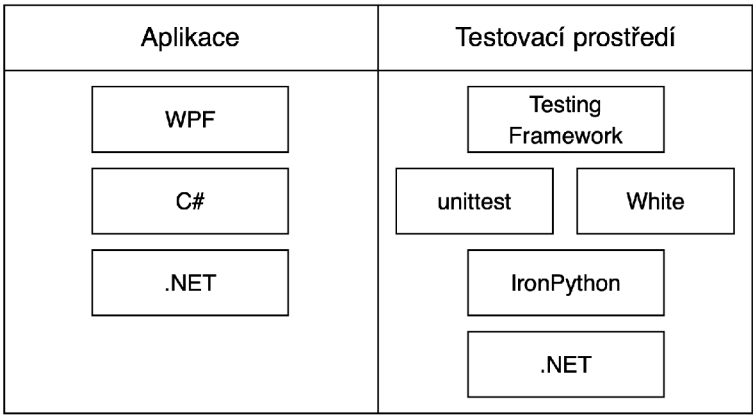
Programovací jazyk C# se řadí mezi jazyky platformy .NET, na rozdíl od jazyka IronPython je však statický. To v první řadě znamená statickou typovou kontrolu, kdy je nutné pro každou proměnnou předem určit datový typ, a ten během používání neměnit. V druhé řadě použití jazyka C# pro psaní testů přímo za běhu aplikace znamená problém, který je popsán v předešlém odstavci týkajícím se jazyka IronPython. Jazyk C# tedy není zvolen jako implementační jazyk testů, část frameworku však bude v tomto jazyce implementována kvůli možnosti využít některé prostředky, které jazyk IronPython nenabízí. Propojení těchto dvou jazyků je však díky platformě .NET velmi jednoduché a přímočaré. Kromě části frameworku bude v jazyce C# implementována také aplikace, na které budou moci probíhat ukázky testů GUI implementované s pomocí vznikajícího frameworku.

Framework White

Třetí použitou technologií je testovací framework White, který je popsán v Sekci 2.3. Princip testování GUI pomocí frameworku White je velmi podobný tomu, který si dává za cíl i mnou vytvářený framework. White je ale implementován v jazyce C#, tedy ve statickém programovacím jazyce, což je nevyhovující pro implementaci všech potřebných funkcí frameworku. Splňuje však alespoň podmínku použití pro platformu .NET. Metody implementované ve frameworku White lze proto využít při vytváření metod pro ovládání elementů GUI, kde není třeba návaznost na dynamicky tvořený kód testované aplikace. White poskytuje jednoduché rozhraní pro manipulaci s elementy GUI. Nabízí metody pro vyhledávání elementů a následně metody pro akce, které lze s konkrétními elementy provádět. Použitím frameworku White se tak snažím vyhnout „objevování kola“ a věnovat ušetřený čas implementaci zajímavějších částí.

Framework `unittest`

Další technologií použitou uvnitř vznikajícího frameworku je framework `unittest`. Jde o implementaci testovacího frameworku JUnit v jazyce IronPython. JUnit je vytvořen v jazyce Java a řadí se mezi nejúspěšnější frameworky pro jednotkové testování (unit testing). `unittest` podporuje automatizaci testování, seskupování testů do kolekcí a odděluje testy od způsobu reportování jejich výsledků [14]. Ačkoliv tato práce necílí na jednotkové testování, je zde stejně dobře možné využít výhody, které framework `unittest` poskytuje. `unittest` ve vznikajícím frameworku usnadňuje spouštění většího množství testů a nabízí metody pro snadné ověření správnosti provedení testu. Dále poskytuje techniky pro jednoduchý a přehledný výpis výsledků jednotlivých testů i pro souhrnný report o úspěšnosti všech vykonaných testů v rámci jednoho běhu.



Obrázek 6.1: Napojení frameworku na testovanou aplikaci a vybrané technologie

6.2 Architektura frameworku

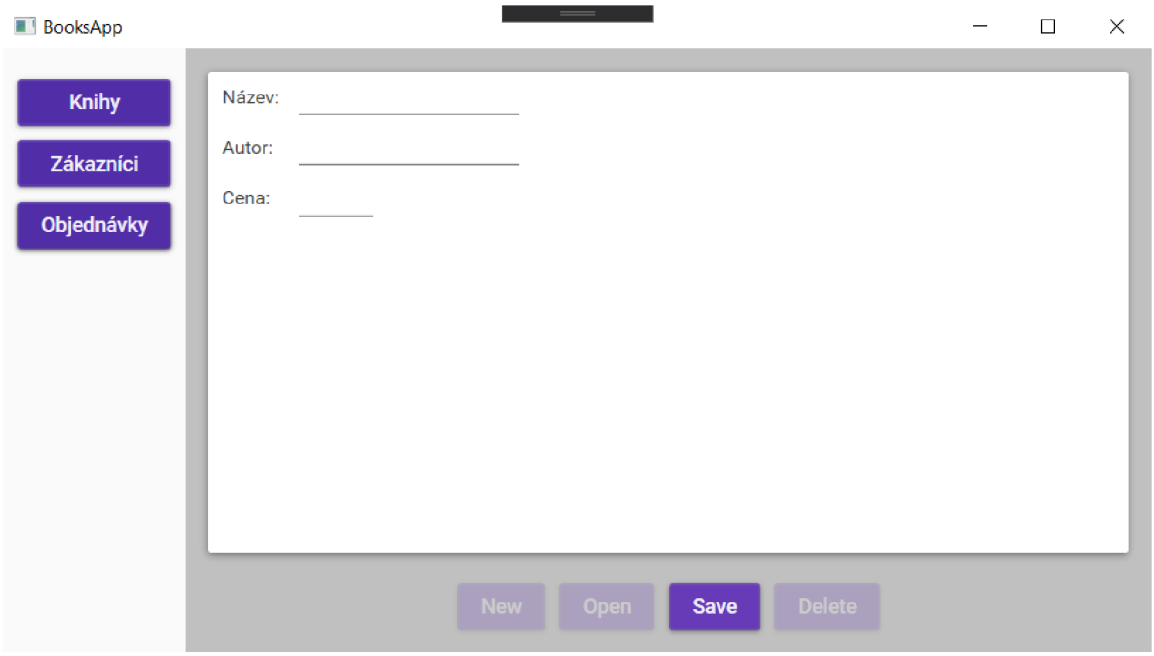
Architekturu frameworku stručně shrnuje Obrázek 6.1. Vzhledem k tomu, že testy musí probíhat za běhu aplikace je nutné, aby byl framework na aplikaci přímo napojen. Za tímto účelem bylo ve firmě vytvořeno testovací prostředí, které umožňuje psaní a spouštění testů GUI aplikace v jazyce IronPython. Testovací prostředí je v implementační části této práce reprezentováno samostatným projektem s názvem *Automation*. Součástí testovacího prostředí je nově vzniklý framework, který se opírá o frameworky White a `unittest`, jazyk IronPython 2.7 a platformu .NET. Posloupnost těchto závislostí lze vidět právě na Obrázku 6.1. V levé části obrázku je znázorněna aplikace a technologie, na kterých je založena, tedy WPF, C# a .NET.

Tvorba testovatelné aplikace

Vznikající framework je určen pro testování aplikací firmy VF, které jsou obvykle postaveny na stejném základu, v rámci firmy označovaném jako *VF.CompositeCore*. Jde o rozsáhlou sadu zdrojových kódů napsaných převážně v programovacím jazyce C#, jejíž instanciací[1] lze vytvořit specifickou aplikaci vyhovující požadavkům zákazníka. Pokud má vzniknout nová aplikace, je použita tato sada a vývoj je tím značně urychlen. Jde o propracovaný projekt, který si firma kvůli konkurenci pečlivě chrání. Z tohoto důvodu bylo třeba v mojí bakalářské práci vytvořit vlastní jednoduchou aplikaci bez použití VF.CompositeCore, která bude dodržovat stejná pravidla jako firemní aplikace, což umožní prezentovat funkčnost testovacího frameworku a přitom nezveřejnit zdrojový kód patřící firmě VF.

Zaměření nové aplikace mohlo být libovolné, a tak jsem vybrala „prodej knih“. Vytvořená aplikace proto nese název *BooksApp* a její zdrojové soubory lze nalézt uvnitř projektu `PrismTest`. Samozřejmě nejde o plnohodnotnou aplikaci. Vznikla pouze jednoduchá aplikace, na které mohou být předvedeny testy vytvořené pomocí frameworku. Aby mohla být nově vznikající aplikace testována stejným způsobem jako firemní aplikace, musí být psána v jazyce C# a grafické rozhraní vytvořeno podle principů architektury WPF, tedy ve značkovacím jazyce XAML. Smyslem WPF je stavět grafické rozhraní aplikace z jednotlivých grafických prvků nazývaných „controls“, jako jsou popisky, textová pole či tlačítka. Vzhled

[1] Instanciace — proces vytváření reálné instance podle nějaké předlohy



Obrázek 6.2: Ukázka formuláře pro vložení nové knihy

takových prvků je předdefinován ve WPF a při psaní zdrojového kódu tak stačí pouze využít správných klíčových slov, např. `Label` pro popisek nebo `Button` pro tlačítko. Výsledný kód v jazyce XAML definující vzhled formuláře na Obrázku 6.2 potom může vypadat takto:

```
<Grid>
    <Label Margin="5">Nazev:</Label>
    <TextBox Width="150" Grid.Column="1" Margin="5" Name="Name"
             Text="{Binding NewBook.Name}"/>

    <Label Margin="5" Grid.Row="1">Autor:</Label>
    <TextBox Grid.Row="1" Grid.Column="1"  Margin="5" Name="Author"
             Text="{Binding NewBook.Author}"/>

    <Label Margin="5" Grid.Row="2">Cena:</Label>
    <TextBox Width="50" HorizontalAlignment="Left" Grid.Row="2"
             Grid.Column="1" Margin="5" Name="Price"
             Text="{BindingNewBook.Price}"/>
</Grid>
```

Pro jednodušší a rychlejší implementaci aplikace jsem zvolila použití technologie Prism. Framework Prism slouží pro vytváření udržovatelných a testovatelných aplikací v jazyce XAML ve WPF. Prism nabízí implementaci řady návrhových vzorů, které jsou užitečné v dobře strukturovaných a udržovatelných aplikacích [3]. Z možných návrhových vzorů jsem zvolila MVVM (*Model-View-ViewModel*), který je typický pro WPF aplikace. MVVM umožňuje oddělit logiku aplikace od uživatelského rozhraní, psát přehledný a spravovatelný

Obrázek 6.3: Okno testovacího prostředí pro psaní testů

kód a zmenšit celkové množství zdrojového kódu aplikace. Jak vyplývá z názvu, zdrojové kódy aplikace jsou rozděleny do tří vrstev: Model, View a ViewModel. Vrstva Model zahrnuje data, se kterými aplikace pracuje. V aplikaci BooksApp nejsou data brána z databáze, jak by tomu pravděpodobně bylo ve většině jiných aplikací. Pro účely aplikace byla data náhodně vygenerována ve formátu XML, ze kterého jsou vždy při spuštění aplikace znovu načítána. Vrstvu Model lze ve struktuře aplikace nalézt v adresáři *Data*. Další vrstva, View, může být označena také jako prezentační. Jde o vrstvu reprezentující uživatelské rozhraní aplikace v jazyce XAML. Poslední vrstvou je ViewModel. Tato vrstva slouží jako prostředník mezi vrstvami View a Model a uchovává stav aplikace. S vrstvou View je propojena pomocí bindingu[2].

Významným prvkem aplikace je boční menu v levé části obrazovky, které slouží pro přechod mezi třemi základními obrazovkami aplikace, jak lze vidět na Obrázku 6.2. Sekce „Knihy“ a „Zákazníci“ jsou potom velmi podobné. Při otevření obě tyto sekce zobrazují seznam všech položek a nabídku základních operací s nimi. Sekce „Objednávky“ slouží čistě pro účely testování, a jsou v ní proto umístěny jednotlivé grafické elementy, které s tématem aplikace nijak nesouvisí.

[2]Binding — technika umožňující prezentovat a upravovat data .NET objektů v uživatelském rozhraní aplikace (synchronizace dat mezi View a ViewModel)

Dynamicky linkovaný modul s frameworkem

Implementační část této bakalářské práce je rozdělena do dvou projektů, které odpovídají architektuře na Obrázku 6.1. Prvním projektem je výše popsaná aplikace *BooksApp*. Druhý projekt nese název *Automation* a jde o dynamicky linkovaný modul, jehož součástí je vznikající framework. Tento modul tvoří testovací prostředí a obsahuje všechny důležité prostředky pro psaní testů, jako jsou framework White, framework `unittest` a IronPython. Ke spuštění testovacího prostředí dochází zároveň se spuštěním testované aplikace. Uživateli se tak objeví nejen okno aplikace, ale i okno s jednoduchým vývojovým prostředím, které je ukázáno na Obrázku 6.3. Testovací prostředí vyvinuté firmou vypadá ve skutečnosti trochu jinak, nabízí více funkcí a je snadno použitelné. Testovací prostředí, jež je součástí implementační části této práce je sice inspirováno firemním, muselo ale projít řadou úprav, aby mohlo být použitelné pro vytvořenou aplikaci a především návrhový vzor MVVM, pro který není původně určeno. Kvůli tomu ztratilo vývojové prostředí několik funkcí a pro běžné užití není příliš efektivní.

Testovací prostředí plní především zmíněnou funkci vývojového prostředí, což z něj dělá nástroj sloužící pro psaní, spouštění a vyhodnocování testů. Psaní testů probíhá v jazyce IronPython v prostřední části okna, kde je otevřen soubor `Test.py`. Kód uvnitř tohoto souboru je spouštěn zeleným tlačítkem v horní části okna. Červené tlačítko slouží pro uložení souboru `Test.py` a černé pro zastavení probíhajícího testu. Modré a žluté tlačítko jsou nefunkční, ale pro účely ukázání práce s frameworkem nejsou důležité a v testovacím prostředí jsou jen jako pozůstatky z firemní verze. V rámci firemní verze testovacího prostředí je možno otevřít v testovacím prostředí jakékoliv soubory s příponou .py. Dolní část okna slouží pro výpis o úspěšnosti testů, jehož podoba se mění v závislosti na typu spouštěných testů. Samozřejmostí je, že uživatel testovacího prostředí může psát jakýkoliv kód v jazyce IronPython, který nesouvisí se vznikajícím frameworkem, ani testováním, a tento kód bez problémů spustit. Očekává se ale především použití pro testování aplikací za pomoci metod vznikajícího frameworku, případně za pomoci frameworků White a `unittest`.

Struktura frameworku

Vnitřní struktura frameworku je rozdělena podle funkcí, které framework při testování plní. První tři odrážky rozdělení zároveň odpovídají prvním třem požadavkům na framework definovaným v Sekci 4.1. Čtvrtá odrážka se týká fuzz testování implementovaného mimo požadavky. Zdrojové soubory frameworku se nachází v rámci projektu `Automation` uvnitř adresáře `GUITestingFramework/`, jehož struktura je následující:

- `ElementsLib/` — knihovna metod pro manipulaci s elementy GUI,
- `TestlOverlap/` — metody pro testování vzájemného překrývání elementů,
- `TestResources.py` — metody pro testování zdrojů popisků v aplikaci a
- `FuzzTesting.py` — metody umožňující fuzz testování aplikace.

Mimo strukturu frameworku, uvnitř adresáře `Scripts/` stojí testovací skript `Test.py`, který je vždy otevřen v testovacím prostředí a uživatel v něm může tvořit testy, které mohou být spouštěny zeleným tlačítkem v okně `Automation`. Dále jsou v adresáři `Scripts/` uloženy další soubory s testy implementovanými pro ukázku.

6.3 Knihovna metod pro manipulaci s elementy GUI

Nejrozsáhlejší částí celého frameworku je knihovna metod, které slouží pro manipulaci s jednotlivými elementy GUI. Knihovna sjednocuje všechny soubory týkající se elementů a všech akcí s nimi souvisejícími. Pro příklad lze uvést tlačítko a možnosti jej ovládat. V takovém případě je třeba vytvořit metody jako nalezení tlačítka, kliknutí na tlačítko nebo zjištění, zda konkrétní tlačítko na obrazovce aplikace existuje. V rámci knihovny jsou metody pro ovládání prvků se stejnými charakteristickými vlastnostmi vždy uloženy do jednoho souboru, struktura knihovny je proto následující:

- Boxes.py — textbox, checkbox, combobox,
- Buttons.py — button, radio button, toggle button, expander,
- Datagrid.py — datagrid cell, datagrid header, column filter,
- DateTime.py — calendar, clock,
- Login.py — metody pro přihlášení uživatele a
- OtherControls.py — ovládání kontrolních prvků mimo uvedené skupiny.

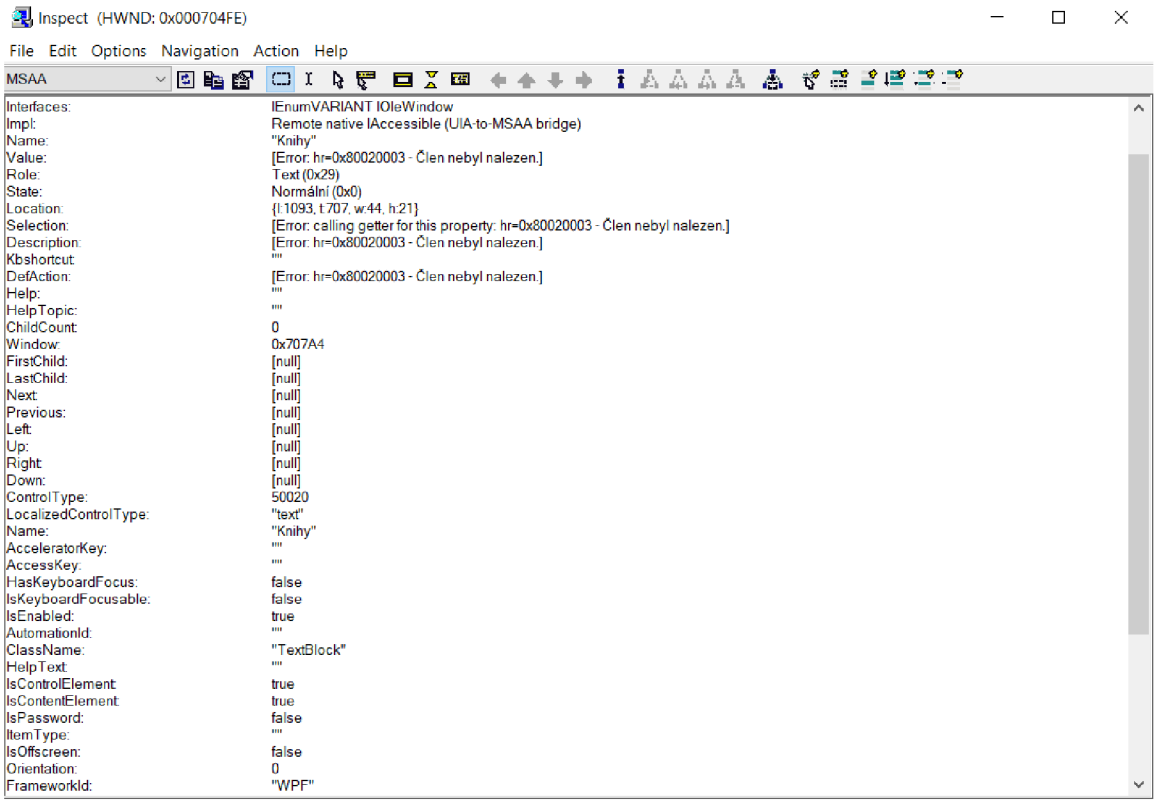
Soubor `Login.py` je v aplikaci BooksApp nepoužitelný vzhledem k tomu, že přihlašování do aplikace není implementováno. Ve zdrojových kódech frameworku ovšem zůstal, jeho metody jsou po importu odpovídajících firemních knihoven použitelné pro testování přihlášení do firemních aplikací. V souboru `Login.py` jsou navíc implementovány ukázky testů přihlášení uživatele. Podobně jsou v souboru `Datagrid.py` ponechány metody pro manipulaci s filtry sloupců nebo metoda pro otevření položky z rozbalovacího menu v souboru `OtherControls.py`.

Implementace metod

První nutností před samotnou implementací metod bylo seznámit se s metodami frameworku White, které slouží pro vyhledávání elementů GUI a pro jejich manipulaci. To znamená, prozkoumat vše, co spadá pod adresář `White/src/TestStack.White/UIItems/`. Metody pro vyhledávání jsou zde sjednoceny v adresáři `/Finders`. Soubory s příponou .cs obsahující metody pro různé akce s elementy jsou většinou přímo v adresáři `/UIItems`. Pokud jde ale o více souborů týkajících se ovládání jednoho elementu, jsou tyto soubory sjednoceny do adresářů, jako např. `/ListBoxItems`.

Dalším krokem bylo prozkoumané metody použít ve vlastním kódu. Zdrojový kód knihovny má být vytvořen v jazyce IronPython, zatímco framework White je napsán v jazyce C#. V obou případech jde tak o jazyky platformy .NET, což umožňuje volat metody frameworku White přímo z IronPython kódu. Jako příklad lze uvést situaci, kdy je potřeba najít na obrazovce aplikace tlačítko a kliknout na něj. Pro lepší představivost bude ukázka vycházet z faktu, že je třeba najít a kliknout na tlačítko „Knihy“ na obrazovce na Obrázku 6.2. Z dokumentace frameworku White [13] vychází, že nejprve je zapotřebí „získat“ okno aplikace a uvnitř něj potom najít správné tlačítko. Pro nalezení okna na základě jeho názvu slouží metoda `GetWindow()`. Okno aplikace BooksApp lze proto získat následovně:

```
1 window = application.GetWindow("BooksApp", InitializeOption.NoCache);
```



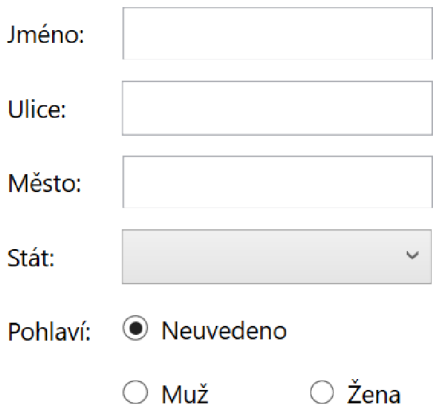
Obrázek 6.4: Okno aplikace Inspect při najetí myší na tlačítko „Knihy“

Parametr `InitializeOption.NoCache` označuje, že kontrolní prvky budou nalezeny pouze na požádání.

Dalším krokem je v rámci nalezeného okna získat tlačítko „Knihy“. Pro nalezení konkrétních prvků okna slouží metoda `SearchCriteria()` z frameworku White. Tato metoda dokáže vyhledávat elementy na základě různých kritérií, jako je název třídy, text, Automation Id, typ kontrolního prvku apod. Takové vlastnosti jsou každému prvku přiřazeny buď defaultně, nebo až při implementaci v XAML kódu. Pro zjišťování vlastností prvků existují nástroje označované jako GUI inspect tools již zmíněné v Sekci 2.3. Okno nástroje Inspect, který jsem při implementaci zvolila, při najetí myší na tlačítko „Knihy“ zobrazuje Obrázek 6.4. Konkrétní vlastnosti, podle kterých může být tlačítko vyhledáno a nástroj Inspect je dokázal najít, jsou Name: „Knihy“, ControlType: 50020, LocalizedControlType: „text“, ClassName: „TextBlock“ a FrameworkId: „WPF“. Nejčastěji je používáno vyhledávání na základě ClassName, Name, Automation Id a indexu. Framework White následně nabízí metodu pro kliknutí na nalezené tlačítko a výsledný kód může vypadat takto:

```
1 btn = w.Get(SearchCriteria.ByClassName("TextBlock").AndByText("Knihy"))
2 btn.Click()
```

Podobným způsobem jsou implementovány všechny metody knihovny. Pokud by výše uvedený kód pro vyhledání tlačítka a kliknutí na něj byl definován jako funkce s názvem `click_button(text)`, jejímž parametrem by byl text, který je nápisem na tlačítku, mohla by tato funkce být použita jako jeden z kroků testu. Uživatel frameworku potom tvoří testy



Obrázek 6.5: Formulář pro vložení nového zákazníka

tak, že postupně volá metody knihovny se zvolenými parametry, a tím ovládá chod celé aplikace.

Vzniklé metody je možné z uživatelského pohledu rozdělit do dvou tříd. Do první z nich spadají metody pro ovládání prvků. Tedy metody jako kliknutí na tlačítko, vyplnění textboxu, zaškrtnutí checkboxu apod. Takových metod je v knihovně definováno celkem 52. Do druhé třídy se řadí metody skrývající v názvu „is enabled“. Tyto metody slouží pro ověření, zda se hledaný prvek vyskytuje, respektive nevyskytuje, v aktuálním okně aplikace. Takové metody mohou najít využití především v situacích, jako je nesprávné vyplnění formuláře, kdy má zůstat tlačítko pro uložení skryto. Metod z této třídy je v knihovně celkově implementováno 56. Mimo to se v knihovně vyskytuje množství pomocných metod, z nichž nejpočetnější je skupina metod pro vyhledávání prvků GUI, které mohou být případně také využity při psaní testů.

Všechny metody knihovny, které jsou přímo používány při psaní testů skrývají ve svém názvu parametr, který je třeba metodě předat. Nejčastěji jde o text, index nebo Automation Id, podle kterých je následně prvek vyhledáván. Požadavek na takový parametr je součástí názvu metody jako „bytext“, „byindex“ nebo „byid“. Metodám ze skupiny „is enabled“ je navíc třeba předat parametr s aktuální instancí `TestCase`, což vyžaduje framework `unittest` pro možnost použití metod typu assert. Pro uživatele vyvíjeného frameworku to znamená jako první parametr takových metod předávat parametr `self`. Pro lepší orientaci začínají názvy metod, kterým je potřeba předávat parametr `self`, slovem „check“.

Použití knihovny

Metody knihovny by měly být používány pro simulaci jednotlivých kroků koncového uživatele aplikace. Test vytvořený pomocí knihovny by v ideálním případě měl sestávat pouze z postupného volání metod knihovny. Jako příklad může sloužit test, který má za úkol otestovat, zda je možné vyplnit všechny položky formuláře pro vložení nového zákazníka na Obrázku 6.5 a zjistit, zda je poté možné vyplněná data uložit. V takovém případě vypadá zdrojový kód testu následovně:

```
1  class CustomerTests(unittest.TestCase):
2      def test_new_customer_form(self):
3          #vyplneni textovych poli se jmenem a adresou
4          fillin_textbox_byid("Adam Marek", "Name")
5          fillin_textbox_byid("Veselá 2", "Street")
6          fillin_textbox_byid("Brno", "City")
7          #vyplneni comboboxu se statem
8          click_combobox_byid("Country")
9          click_textblock_bytext("ID")
10         #vyplneni radio buttonu s pohlavim
11         click_radio_bytext("Muž")
12         #kontrola, zda je na tlacitko "Save" mozne kliknout
13         check_button_bytext_isenabled(self, "Save")
14
15 print("STARTED")
16 #nacteni testu z tridy CustomerTests
17 suite = unittest.TestLoader().loadTestsFromTestCase(CustomerTests)
18 #spusteni nactenych testu
19 unittest.TextTestRunner(verbosity=2).run(suite)
20 print("FINISHED")
```

Zdrojový kód testu vyplnění formuláře pro vložení nového zákazníka a ověření možnosti kliknout na tlačítko „Save“ je v metodě `test_new_customer_form(self)`, kde jsou zároveň v komentářích popsány jednotlivé kroky a jejich význam. Ve zdrojovém kódu je vidět, jak funguje předávání parametrů metodám, tzn. že metody přijímají takový parametr, jaký je součástí jejich názvů a metoda `check_button_bytext_isenabled(self, "Save")` má na vstupu parametr `self`. Metoda s testem je součástí třídy `CustomerTests`, kde může být vytvořeno více podobných metod s testy. Na řádku číslo 15 začíná spouštění testu výpisem slova „STARTED“. Následuje řádek kódu, který dokáže načíst všechny testy třídy `CustomerTests`, a další řádek tyto testy spouští. Poslední řádek pouze informuje uživatele, že testy byly dokončeny. Pro lepší pochopení, jak dochází ke spouštění testů, je třeba vysvětlit některé použité prostředky z `unittest` frameworku [14]:

- `TestCase` je třída reprezentující jeden test,
- `suite` je kolekce jednotlivých testů (`TestCase`),
- `TestLoader()` je třída sloužící pro vytvoření kolekcí testů (`suite`) z tříd nebo modulů,
- `loadTestsFromTestCase(BookTests)` je metoda třídy `TestLoader()`, která umí získat kolekci všech testů (`suite`) z třídy na vstupu metody (třída na vstupu musí být odvozena z třídy `TestCase`),
- `TextTestRunner()` je třída sloužící pro spouštění testů, která je konfigurovatelná několika parametry, přičemž `verbosity=2` označuje detailnější výpis informací o průběhu testu a

- `run(suite)` je metoda třídy `TextTestRunner()`, která na vstupu přijímá parametr označující jeden test nebo celou kolekci testů, a tyto testy postupně spouští.

Důležité je upozornit, že metoda `loadTestsFromTestCase(BookTests)` načte jako testy pouze metody, které začínají slovem „test“. V rámci třídy odvozené z `TestCase` lze proto vytvořit i pomocné metody, pokud budou pojmenovány vhodným způsobem.

Z názvu třídy `CustomerTests` lze odvodit, že testy implementované uvnitř této třídy se budou odehrávat v okně aplikace se seznamem zákazníků. Tester by tak mohl využít metod `setUp()` a `tearDown()` frameworku `unittest` pro pohodlnější testování. Metoda `setUp()` je volána před spuštěním každého testu dané třídy a v uvedeném případě by její kód mohl zajišťovat navigaci do okna se seznamem zákazníků:

```
1 class CustomerTests(unittest.TestCase):
2     def setUp(self):
3         click_button_bytext("Zákazníci")
```

Pokud v metodě `setUp()` dojde k chybě, test nebude vykonán a na výstupu se objeví `Error`. Ke spuštění metody `tearDown()` dochází naopak po provedení každého testu, úspěšného i neúspěšného, a dokonce i v případě, že test vyvolá výjimku. Úkolem metody `tearDown()` je po testu tzv. „uklidit“, což může znamenat např. odstranění položky, která byla při testu vytvořena. Tyto dvě metody jsou souhrnně označovány jako *Test Fixture*. Jde o udržení určitého stavu aplikace, který je využíván jako výchozí stav pro provádění testů. Cílem *Test Fixture* je zajistit jednotné prostředí pro spouštění testů tak, aby mohly být testy prováděny opakovaně. Tím je zaručeno, že výsledky opakovaných testů jsou porovnatelné.

Z použití těchto prostředků zároveň vychází, že se framework `unittest` stará i o vyhodnocení testů. Informace o úspěšnosti testů jsou vypsány do testovacího prostředí, kde došlo i ke spuštění testů. Každý test je vyhodnocen jako úspěšný (OK) nebo neúspěšný (FAIL, popř. ERROR) a výpis uzavírá informace o souhrnném počtu neúspěšných testů ze všech provedených. Vyhodnocení může vypadat např. takto:

```
test1 (__main__.BookTests) ... ok
test2 (__main__.BookTests) ... ok
test3 (__main__.BookTests) ... FAIL

======================================================================
FAIL: test3 (__main__.BookTests)
----------------------------------------------------------------------
Traceback (most recent call last):
    File "Test.py", line 56, in test3
    File "C:\VF_testingFramework\BooksApp_Test\K_PrismTest\Automation
            \Scripts\Buttons.py", line 49, in
            check_button_byindex_isenabled
        case.assertTrue(is_enabled_button_byindex("Button", index))
    AssertionError: 0 is not true

----------------------------------------------------------------------
Ran 3 tests in 10.560s
FAILED (failures=1)
```

Proto je zřejmé, že uživatel vznikajícího frameworku musí mít alespoň základní znalost vybraných prostředků frameworku `unittest`, bez kterých by spouštění testů sestavených z metod knihovny bylo neefektivní a hlavně by nedocházelo k vyhodnocení spuštěných testů, neboť pro metody knihovny žádný mechanismus vyhodnocování není ve frameworku implementován.

6.4 Metody pro testování překladů štítků

Při tvorbě aplikace dochází k definování různých štítků a popisků, které jsou pro koncového uživatele aplikace viditelné. Příkladem může být třeba samotný název aplikace (titulek hlavního okna aplikace). Ten bývá často viditelný po celou dobu práce s aplikací a ve zdrojovém kódu okna aplikace v jazyce XAML je definován jako:

```
Title="BooksApp"
```

Název je pro každou vytvořenou aplikaci obvykle jiný, a proto je vhodné, pokud je definován právě takto jako řetězcový literál přímo ve zdrojovém kódu. Popiskům ve formě řetězcových literálů je třeba se v jiných situacích vyhnout. Lepším řešením, jak vložit popisky do zdrojového kódu aplikace, je získávat je ze souborů k tomu určeným, což ve zdrojovém kódu může vypadat takto:

```
Title="{Source={StaticResource ResourceKey}}"
```

Aby tento kód fungoval, je zapotřebí nejdříve v XAML kódu definovat `ResourceKey`, kde jsou formulovány požadované popisky buď přímo, nebo pomocí souborů k tomu určeným. Klíčové slovo `StaticResource` zde říká, že zdroj je získán pouze jednou, a to při načtení XAML souboru. Pokud poté dojde ke změně zdroje, je tento fakt ignorován. Druhou možností je použití `DynamicResource`, kdy je zdroj popisků načítán dynamicky za běhu aplikace vždy, když je změněn. Další možností, jak získat konkrétní popisek z určitého zdroje, je pomocí bindingu. Binding umožňuje zobrazit v popisku data, která jsou uložena v proměnných uvnitř aplikace. Ve zdrojovém kódu může být binding použit následovně:

```
Text="{Binding Book.Name}"
```

Cílem je tedy ověřit, zda jsou štítky a popisky v aplikaci napsány formou řetězcových literálů přímo ve zdrojových kódech oken, nebo zda jsou načítány z odpovídajících zdrojů. Vzhledem k tomu, že ve většině případů je vhodné využít pro popisky specializované soubory se zdroji, měl by framework upozorňovat uživatele na popisky, které jsou psány ve formě řetězcových literálů.

Možnosti řešení

Pro testování překladů štítků a popisků aplikace existuje více možných způsobů řešení. Jako první z nich se nabízí možnost zjišťovat, jestli jsou všechny popisky v aplikaci ve stejném jazyce. V takovém případě by muselo dojít k vytvoření snímku každého okna aplikace, získání všech popisků a následně porovnání, zda se jedná o text v daném jazyce. Takové řešení je ale náročné na implementaci a naráží na problém ve chvíli, kdy jsou některé pojmy v aplikaci záměrně ponechány bez překladu, např. v angličtině. Druhou možností je za běhu aplikace testovat, jestli má atribut popisku definující zobrazený text nastavenou lokální hodnotu (řetězcový literál) nebo je napojen na nějaký resource provider, který poskytuje

štítkům zdroje dat. Pokud by ale zdroj popisku byl definován pomocí `StaticResource`, jevil by se za běhu jako lokálně nastavená hodnota, což by framework vyhodnotil jako chybu. Třetí možnost vede na prohledávání zdrojových kódů. V takovém případě by framework nabízel metody pro prohledávání XAML souborů, ve kterých by hledal výskyt popisků definovaných jako řetězcové literály. Ani tato možnost není stoprocentní, ale je poměrně nenáročná na implementaci a spolehlivější než předchozí dvě uvedené. U posledních dvou možností je ovšem nutné zmínit, že nedochází k testování překladů štítků, ale pouze ke kontrole, že štítky jsou brány z nějakých zdrojových souborů.

Implementace řešení

Pro implementaci jsem zvolila třetí z možných řešení, jehož princip je v prohledávání zdrojových souborů. Je tedy třeba získat cestu k XAML souboru, otevřít jej a hledat v něm výskyt lokálně definovaných štítků. Pro tento účel vznikla ve frameworku metoda `test_resources(path)`, která na vstupu očekává absolutní cestu k souboru nebo adresáři, jenž má být otestován. Dochází samozřejmě k testování pouze souborů s příponou .xaml, ostatní soubory jsou ignorovány. Z příkladů zdrojových kódů štítků v úvodu této sekce vychází, že pokud je text popisku brán z nějakého zdroje, je syntaxe atributu následující:

```
atribut="{...}"
```

V nesprávném řešení (lokální definice popisku) jsou složené závorky vynechány. Díky tomu mohou být lokálně definované popisky snadno nalezeny pomocí následujícího regulárního výrazu:

```
="[^{].[^}]*"
```

Tomuto výrazu ale odpovídá velké množství atributů, které nejsou cílem hledání, jako např. `Width` nebo `Margin`. V důsledku toho bylo nutné specifikovat pouze atributy, které se používají pro definování textů zobrazených v aplikaci. Mezi takové atributy patří `Text`, `Content` a `Title`.

Použití při testování

Aktuálně celé testování překladů štítků probíhá při zavolání metody `test_resources()`. Přesnější by bylo napsat, že pomocí této metody dochází k ověřování, že jsou popisky aplikace brány z různých zdrojových souborů, tedy nejsou napsány přímo ve zdrojových souborech okna aplikace. Původně měla metoda `test_resources()` kromě této funkcionality i informovat uživatele o konkrétních zdrojových souborech, odkud jsou popisky aplikace získávány. Zjišťování zdroje popisku je ale problematické. Zdroj bývá často definován jako `StaticResource ResourceKey`, kde `ResourceKey` označuje klíč, pod kterým má WPF hledat zdroj dat pro popisek. Klíč může být definován dříve ve stejném souboru, kde se nachází popisek, nebo se může nalézat v jakémkoliv hierarchicky nadřazeném souboru. Obvykle se takový zdroj definuje až v kořenovém XAML souboru aplikace *App.xaml*, aby byl dostupný ve zdrojových kódech všech oken. To stejné platí při použití `DynamicResource`. Z toho důvodu je po dohodě ve firmě funkčnost omezena pouze na hledání popisků ve formě řetězcových literálů.

Pokud chce uživatel frameworku otestovat, kde se ve zdrojových kódech aplikace nachází popisky definované jako řetězcové literály, může použít metodu `test_resources(path)`. Metoda na vstupu přijímá jeden parametr, kterým může být buď absolutní cesta k XAML

souboru, jenž má být otestován, nebo absolutní cesta k adresáři, v němž budou otestovány všechny soubory s příponou .xaml. Nutností je ale parametr metodě předat v tomto tvaru:

```
test_resources(r"C:\Users\Views\InvoiceListView.xaml")
```

Písmeno „r“ před cestou k souboru umožňuje předat metodě parametr správně včetně speciálních znaků. Výstupem této metody je pro každý soubor informace „OK“ (soubor bez řetězcových literálů pro popisky) nebo „WARNING“ (soubor s výskytem řetězcového literálu v alespoň jednom z popisků) s následným označením čísla řádku v souboru, kde je popisek definován, a výpisem obsahu tohoto řádku:

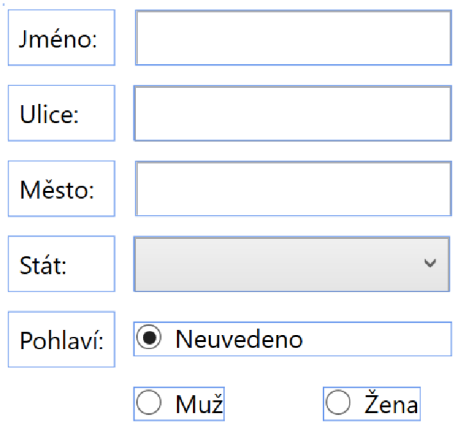
```
STARTED
Resources Test:

 <BookDetailView.xaml> OK
 <BookListView.xaml> OK
 <InvoiceListView.xaml> WARNING
 Hardcoded string in attribute 'Text' on line 34,
     <DatePicker Grid.Row="1" Grid.Column="1" Text="2.3.2019">

 <MainWindow.xaml> WARNING
 Hardcoded string in attribute 'Title' on line 19,
     Title="BooksApp" Height="450" Width="800">

 <MenuView.xaml> OK
FINISHED
```

6.5 Metody pro testování přetékání prvků formuláře

Vzhled aplikací vytvářených ve firmě je definován v jazyce XAML, z čehož vyplývá, že GUI je tvořeno jednotlivými grafickými prvky jazyka XAML. Tyto prvky jsou nazývány jako „kontrolní“ (*Controls*) a vývojář pomocí jejich skládání utváří obsah všech oken aplikace. Každý kontrolní prvek zabírá v rámci okna aplikace určitý prostor, který může vyplňovat. Pokud by obsahem prvku byl text, který by se nevešel do vyhrazeného prostoru, bude v aplikaci zobrazena jen část textu vyplňující prostor vyhrazený pro prvek, zbytek nebude viditelný. V technologii WPF má prostor pro každý prvek tvar obdélníku a v okně s formulářem si lze takové obdélníky představit jako na Obrázku 6.6. Z toho vyplývá, že testování přetékání prvků formuláře by mělo probíhat na základě porovnávání vzájemné polohy těchto obdélníků.

Analýza testovaných objektů

Důležitou vlastností každého okna aplikace je vlastnost `Content`. Jde o element reprezentující rozložení jednotlivých prvků okna. Jako `Content` daného okna může být brán právě jeden grafický prvek a všechny další prvky okna jsou jeho potomky. Prvky často využívané jako `Content` jsou např. `Grid`, `StackPanel` nebo `Canvas`. Ve formuláři na Obrázku 6.6 je jako `Content` definován `Grid` a zjednodušený zdrojový kód okna v jazyce XAML potom vypadá takto:



Obrázek 6.6: Formulář se zvýrazněnými obdélníky, které zabírají jednotlivé prvky

```
1  <Grid>
2      <Label>Jméno:</Label><TextBox/>
3      <Label>Ulice:</Label><TextBox/>
4      <Label>Město:</Label><TextBox/>
5      <Label>Stát</Label><ComboBox/>
6      <Label>Pohlaví</Label>
7      <RadioButton>Neuvedeno</RadioButton>
8      <RadioButton>Muž</RadioButton>
9      <RadioButton>Žena</RadioButton>
10 </Grid>
```

`Content` typu `Grid` je pro formuláře typický, lze pomocí něj jednoduše definovat rozložení okna na řádky a sloupce. Pokud tedy testování cílí na přetékání prvků formuláře, lze tento požadavek zobecnit na testování přetékání prvků v rámci rozložení typu `Grid`.

Další podstatnou informací pro testování přetékání je způsob, jakým je obsah oken tvořen. V každé aplikaci architektury WPF je definován vzhled hlavního okna (obvykle v souboru `MainWindow.xaml`). V aplikaci BooksApp je už samotné hlavní okno definováno jako `Grid`, což, jak lze vidět na Obrázku 6.2, rozděluje okno na dva sloupce: levý s tlačítky a pravý s různým obsahem. Uvnitř pravého sloupce je opět definován `Grid`, který rozděluje sloupec na dva řádky: horní s obsahem okna a dolní s tlačítky. Uvnitř horního řádku se znovu nachází `Grid`, ten lze rozlišit podle bílého pozadí. Jeho náplní je prvek `ContentControl` sloužící pro přepínání obsahu. Jako obsah je na Obrázku 6.2 zobrazen prvek `NewBookView`, jinak řečeno obsah souboru `NewBookView.xaml`, ve kterém je opět definován `Grid`, tentokrát už s konkrétními prvky formuláře, tedy s několika prvky typu `Label` a `TextBox`. A přesně takto postupně je třeba se k formuláři dostat i při testování přetékání. Zároveň při tomto způsobu procházení různých prvků směrem k formuláři dochází k otestování přetékání všech prvků nalezených cestou.

Implementace řešení

Prvním krokem před implementací metod pro testování přetékání prvků formuláře bylo prozkoumat existující prostředky vhodné k manipulaci s obdélníky prvků. Takové prostředky jsem nalezla v jazyce C# jako součást jmenného prostoru `System.Windows`, který mimo jiné poskytuje třídy pro základní prvky WPF a pro jejich vyhledávání v okně aplikace. Mezi základní prvky WPF se řadí např. `Button`, `Grid`, `Calendar` apod. `System.Windows` nabízí také metody pro ovládání těchto prvků a zprostředkovává vývojáři jejich vlastnosti.

Řešení je proto implementováno v jazyce C#. V rámci projektu `Automation` se metody pro testování přetékání prvků formuláře nachází v souboru `TestRectangles.cs`. Celé testování přetékání je řízeno metodou `TestRectanglesOverlap()`, z níž jsou volány další pomocné metody. Průběh testu lze zjednodušit a rozdělit do následujících fází:

1. získání instance hlavního okna aplikace,
2. získání instance prvku, který určuje rozložení hlavního okna,
3. otestování přetékání potomků prvku získaného v předchozím kroku,
4. procházení přes všechny prvky typu `Grid` nebo `ContentControl` až k formuláři aktuálně zobrazenému v okně aplikace a
5. průběžné testování přetékání potomků nalezených prvků.

Poslední zmíněný bod je nejdůležitějším krokem testu. Aby mohlo dojít k otestování, zda se některé obdélníky prvků nepřekrývají, je nejprve zapotřebí tyto obdélníky získat. K získání pozice obdélníku jednoho prvku dochází v metodě `GetAbsolutePlacement()`, jejíž pseudokód lze vyjádřit jako:

```
1  Rectangle GetAbsolutePlacement(element)
2  {
3      absolutePosition = element.Point(0, 0);
4      mainWindowPosition = MainWindow.Point(0, 0);
5
6      absolutePosition = new Point(
7          absolutePosition.X - mainWindowPosition.X,
8          absolutePosition.Y - mainWindowPosition.Y);
9      return new Rect(
10         absolutePosition.X, absolutePosition.Y,
11         element.ActualWidth, element.ActualHeight);
12 }
```

Na řádcích číslo 3 a 4 dochází k inicializaci bodů levých horních rohů prvku a hlavního okna aplikace. Na řádku číslo 6 začíná výpočet souřadnic bodu levého horního rohu prvku v rámci okna. Před tímto výpočtem byly známé 2 body, ale nebyla známá jejich vzájemná poloha. Jinak řečeno, nebyla známá poloha prvku uvnitř okna aplikace. Na řádku číslo 9 je vytvořen obdélník, který je definován pomocí souřadnice osy x definující levou stranu obdélníku (`absolutePosition.X`), souřadnice horní strany (`absolutePosition.Y`), aktuální šířky prvku (`element.ActualWidth`) a aktuální výšky prvku (`element.ActualHeight`).

Je zřejmé, že metoda `GetAbsolutePlacement()` musí být volána pro všechny potomky prvku definujícího rozložení. Nalezené obdélníky jsou v metodě `FindChildren()` řetězeny do seznamu, aby mohlo následně dojít k testování jejich vzájemné polohy v metodě `TestRactanglesOverlap()`. Testování vzájemné polohy obdélníků probíhá pomocí metody `IntersectsWith()` ze jmenného prostoru `System.Windows`. Použití této metody je následující:

```
rectangles[i].IntersectsWith(rectangles[j])
```

Přesně takto je metoda použita ve zdrojovém kódu testu. Metoda zde pracuje pouze se seznamem obdélníků `rectangles`. Proměnná `rectangles[i]` označuje jeden obdélník ze seznamu a `rectangles[j]` označuje jiný obdélník seznamu. Metoda `IntersectsWith()` poté určuje, zda je průnik těchto dvou obdélníků prázdný, či nikoliv.

V souboru `TestRectangles.cs` jsou dále definovány metody, které slouží pro správný průchod přes všechny prvky nadřazené testovanému formuláři:

- `FindGrid()` - nalezení potomka prvku aktuálního rozložení, který je typu `Grid`,
- `FindContentControl()` - nalezení potomka prvku aktuálního rozložení, který je typu `ContentControl` a
- `GetFollowingGrid()` - nalezení dalšího potomka typu `Grid`, který nemusí být přímým potomkem aktuálního prvku.

Výsledkem testování přetékání prvků formuláře je slovník s objekty typu `TestedElement`. Třída `TestedElement` je implementována v souboru `TestedElement.cs` uvnitř adresáře `TestOverlap/`. Třída uchovává 3 atributy: typ prvku, název prvku a výsledek testování daného prvku. V průběhu testu je pro každý testovaný prvek rozložení vytvořena instance třídy `TestedElement`, která slouží pro ukládání právě zmíněných atributů. Tato data jsou následně použita pro výpis informací pro uživatele frameworku. Uživatel tak může jednoduše zjistit, v rámci kterého rozložení k přetékání prvků dochází. V případě, že test proběhne bez chyby, je z vypsaných informací možné zjistit, přes které vrstvy aplikace test proběhl.

Aby mohlo být testování přetékání jednoduše spouštěno z testovacího prostředí, bylo třeba vytvořit soubor v jazyce IronPython (`TestOverlap.py`). V něm je implementována metoda `test_overlap()`, která pouze volá C# metodu `TestRectanglesOverlap()` a řídí výpis informací o průběhu testu.

Použití při testování

Pokud chce uživatel frameworku otestovat, zda některé prvky vybraného formuláře nepřekrývají jiné prvky formuláře, zavolá z testovacího prostředí metodu `test_overlap()`, na jejímž vstupu nejsou žádné parametry. Metoda totiž testuje pouze aktuálně zobrazené okno v aplikaci. Uživatel tedy zobrazí v aplikaci okno, které chce otestovat, a až poté spouští test přetékání. Vyhodnocení testu okna s formulářem na Obrázku 6.7 může vypadat takto:

```
Grid "LayoutRoot" - OK
Grid "InnerGrid" - OK
Grid "Card" - OK
NewBookView - OK
Grid "NewBookLayout" - FAIL
```

Obrázek 6.7: Ukázka formulářového okna, kde se nápis „Autor:“ a textbox překrývají

Každý vypsaný řádek reprezentuje jednu úroveň (prvek definující rozložení), přes kterou test prošel. Průchod testu skrz úrovně okna končí ve chvíli, kdy se objeví chyba, nebo když test nenalezne další prvek rozložení, který by mohl otestovat. Z uvedeného výpisu lze zjistit, že test prošel skrz 5 vrstev aplikace, které detailně popisuje část *Analýza testovaných objektů* dříve v této sekci. První z vrstev, `Grid` s názvem „LayoutRoot“, je rozdělení hlavního okna na dva sloupce, „InnerGrid“ rozděluje pravý sloupec na dva řádky a „Card“ je `Grid` v horním řádku s bílým pozadím a obsahem konkrétního okna. Obsah okna zde tvoří formulář pro vložení nové knihy, který je implementován v souboru NewBookView.xaml, odtud plyne typ prvku `NewBookView`. Uvnitř něj je další `Grid`, tentokrát s názvem „NewBookLayout“, který určuje rozložení jednotlivých prvků formuláře, tzn. rozložení prvků typu `Label` a `TextBox`. Syntaxe každého vypsaného řádku je: `typ_testovaného_prvku „název_testovaného_prvku" - výsledek_testu`. Přičemž název prvku není povinný, a proto u `NewBookView` chybí.

Testování přetékání prvků lze samozřejmě spustit i pro jiné než formulářové okno. V takovém případě budou otestovány všechny dosažitelné vrstvy. Je tak možné, že dojde k otestování všech potřebných prvků okna, ale také může dojít k otestování pouze rozložení hlavního okna aplikace a ukončení testu. Proto je vhodné, aby byly prvky rozložení v aplikaci pojmenovány. V první řadě to umožní uživateli frameworku lépe dohledat chybu, ale také bude zřejmé, jaké vrstvy test prošel a jaké nikoliv.

6.6 Fuzz testování

Ačkoliv v požadavcích na framework fuzz testování zmíněno není, vznikl po implementaci předchozích částí nápad, že by jeho zařazení do funkcí frameworku mohlo být přínosné. Tento nápad navíc podpořil fakt, že dílčí kroky, které by mohly být při fuzz testování využity, už ve frameworku implementovány jsou. A i kdyby přímo tyto metody frameworku nebyly použity, není třeba se dlouze seznamovat s novými technologiemi, neboť by pro implementaci měly stačit prostředky frameworku White. Proto by měla být implementace fuzzingu poměrně nenáročná a pro použití při testování výhodná.

Princip řešení

Základní princip fuzz testování je popsán v Sekci 2.2. Použití fuzzingu pro potřeby testování aplikací firmy VF si lze představit jako generování a vykonávání náhodných uživatelských vstupů, jako např. kliknutí na různé typy tlačítek, vyplňování různých typů boxů nebo

přecházení mezi okny aplikace. Cílem je tedy vykonat určitý počet náhodných kroků za poměrně krátký čas a pokusit se tím aplikaci „rozbít“. Jednotlivé kroky vedoucí k pádu aplikace by přitom měly být ukládány, aby mohl být konkrétní test vedoucí k chybě opakovatelný a tester mohl nalezenou chybu reportovat vývojáři.

Implementace řešení

Kód vykonávající funkci fuzz testování je implementován v souboru `FuzzTesting.py` konkrétně uvnitř metody `fuzz_testing(steps)`. Implementována je jednoduchá funkčnost fuzzeru, který ke generování náhodných vstupů nepoužívá žádné znalosti o vnitřní struktuře aplikace. Kód metody `fuzz_testing(steps)` pracuje vždy s aktuálním oknem aplikace a zjednodušeně lze jeho kroky popsat takto:

1. získání všech prvků aktuálního okna aplikace,
2. ze všech prvků okna získání pouze těch, které jsou viditelné a přístupné pro kliknutí,
3. vygenerování náhodného indexu prvku ze všech získaných v předchozím kroku,
4. kliknutí na náhodně vybraný prvek a
5. v případě textboxu vepsání textu „text123“.

Uvedené kroky jsou postupně vykonány tolikrát, kolikrát specifikuje uživatel metody v parametru `steps`. Uživatel tedy definuje počet náhodných kliknutí, popř. vepsání textu s cílem aplikaci „rozbít“. Pro implementaci vyhledávání a ovládání jednotlivých prvků okna byly opět použity prostředky frameworku White. Dále byla k implementaci využita knihovna `random`, jejíž metoda `randint()` zajišťuje výběr náhodného indexu prvku.

Při implementaci bylo třeba vyřešit, jaké informace o průběhu fuzz testování uživateli poskytovat a také jakým způsobem je zpřístupnit. Pro možnost provedený test zopakovat a případně detailně reportovat nalezenou chybu, metoda ukládá každý vykonaný krok. Uložené kroky ovšem nemohou být vypsány do testovacího prostředí, jak tomu je u jiných metod frameworku. Ve chvíli, kdy se aplikaci podaří „rozbít“, totiž spadne nejen aplikace, ale i testovací prostředí, které je k aplikaci dynamicky přilinkováno. V takovém případě by tedy výstup testu byl nedohledatelný. Navíc se očekává, že při fuzz testování budou prováděny stovky kroků, což by při výpisu do testovacího prostředí nebylo přehledné. Pro tyto účely je v rámci každého spuštění metody `fuzz_testing(steps)` vytvořen logovací soubor `fuzing_DDMMYY-HHMMSS.txt`, kam se vykonané kroky testu ukládají. Každý krok má v logovacím souboru vyhrazen jeden řádek, kde jsou informace o prvku okna, na nějž bylo v rámci testu kliknuto, popř. vepsán text. Všechny logovací soubory jsou ukládány do adresáře `logfiles/`, kde jsou dohledatelné podle data a času uloženého jako součást názvu.

Uživatel frameworku poté spouští fuzz testování se 100 kroky uvnitř testovacího prostředí takto: `fuzz_testing(100)`. Souhrnný výsledek testu se následně objeví v dolní části testovacího prostředí a informuje uživatele o úspěšném vykonání a ukončení testu. Neúspěch je indikován pádem jak aplikace, tak testovacího prostředí, přičemž kroky vedoucí k pádu jsou zaznamenány v odpovídajícím logovacím souboru.

6.7 Automatizace spouštění testů

V poslední fázi implementace bylo třeba vyřešit, jakým způsobem testy vytvořené pomocí frameworku spouštět tak, aby testování bylo efektivní. Především je třeba zaručit možnost

spouštět více testů v rámci jednoho testování. Pod tím si lze představit, že tester klikne na tlačítko spustit, čímž odstartuje první test z předem vytvořené kolekce testů. Následně tester nechá testy běžet a věnuje se jiné práci. Po dokončení všech testů, jejichž počet je očekáván v řádu desítek, tester projde výsledné hodnocení testů a na základě zjištěných problémů dále testuje, případně reportuje chyby apod. Cílem je tedy vytvořit mechanismus, který umožní skládat testy do kolekcí a následně kolekce spouštět, a to ideálně i několik kolekcí za sebou.

Implementace řešení

Implementace spouštění testů se liší pro různé typy metod frameworku. Prvním typem jsou metody knihovny pro manipulaci s elementy GUI, pro jejichž spouštění jsou využívány prostředky frameworku `unittest`. V Sekci 6.3 je popsáno použití metod knihovny při testování, kde je zároveň ukázáno, jakým způsobem lze testy seskupovat do kolekcí. V okně `Automation` potom mohou být takto vytvořené kolekce testů spouštěny, pokud jsou uloženy v souboru `Test.py`. Pro reálné použití frameworku je samozřejmě velmi nevýhodné, aby všechny testy musely být implementovány v jediném souboru. Spouštění výhradně souboru `Test.py` je řešení pouze pro účely zjednodušeného testovacího prostředí, které je používáno v praktické části této práce. Při použití frameworku uvnitř plně funkčního testovacího prostředí bude testování probíhat trochu odlišně. Uvnitř plně funkčního testovacího prostředí lze vytvářet nové adresáře i nové testovací skripty v jazyce IronPython. V testovacím prostředí je také možné otevřít několik souborů zároveň a každý z nich libovolně upravovat. Při reálném použití by tak ideálně uvnitř adresáře `Scripts/` byly uloženy všechny vytvořené testovací skripty, které by byly tématicky sjednoceny do adresářů odpovídajících názvů. Spouštění testů poté probíhá vždy ze souboru `main.py`. Aby mohly být spouštěny konkrétní testy nebo sady testů, je třeba pro každý adresář s testovacími skripty vytvořit právě jeden soubor s názvem `__init__.py`. Struktura adresáře `Scripts/` by potom měla vypadat takto:

- `Scripts/`
  - `BookTests/`
    - `__init__.py`
    - `BookInsertTests.py`
    - `BookDetailTests.py`
  - `CustomerTests/`
    - `__init__.py`
    - `CustomerRemoveTests.py`
  - `main.py`

Soubory `main.py` a `__init__.py` slouží nejen pro přehlednější strukturu vytvořených testů, ale především pro automatizaci spouštění testů. Jsou v nich použity prostředky frameworku `unittest`, které umožňují jednoduše tvořit kolekce testů a následně tyto kolekce spouštět. Obsah souboru `main.py` si lze představit následovně:

```
1  #import adresaru s testovacimi skripty
2  import BookTests
3  import CustomerTests
4
5  if __name__ == '__main__':
6      runner = unittest.TextTestRunner(failfast=False, verbosity=2)
7      #spusteni kolekci testu vytvorenych v souborech __init__.py
8      runner.run(BookTests.suite())
9      runner.run(CustomerTests.suite())
```

Změny obsahu souboru `main.py` budou pro jiné případy použití spočívat pouze v importování nových adresářů a spouštění dalších kolekcí testů. Kolekce testů budou definovány v souborech `__init__.py` takto:

```
1  def suite():
2      suite = unittest.TestSuite()
3      suite.addTest(unittest.makeSuite(TestsClassName))
4      suite.addTest(TestsClassName('test_name1'))
```

Proměnná `suite` zde reprezentuje kolekci testů a je odvozena z třídy `TestSuite` frameworku `unittest`. Do kolekce mohou být přidány buď všechny testy definované v třídě `TestsClassName` (řádek číslo 3), nebo mohou být do kolekce testy přidávány jednotlivě (řádek číslo 4).

Druhý typ metod frameworku představuje metoda `test_resources()`, která ověřuje zdroje štítků a popisků aplikace. V tomto případě je možnost rozsáhlejšího testování aplikace řešena jiným způsobem. Vstupem metody může být jak jediný soubor, tak celý adresář, uvnitř kterého jsou otestovány všechny soubory s příponou .xaml. Je velmi pravděpodobné, že všechny soubory definující vzhled aplikace (soubory s příponou .xaml) budou uloženy v jednom adresáři. Pokud je cesta k tomuto adresáři předána metodě `test_resources()`, dojde k otestování všech zdrojů štítků a popisků v aplikaci.

Třetím typem metody frameworku je metoda `test_overlap()` sloužící pro testování přetékání prvků. Metoda vždy testuje aktuálně otevřené okno, a proto se u této metody nepodařilo realizovat spuštění globálně pro celou aplikaci. Pokud by ale pro testování aplikace byla taková funkčnost výhodná, může ji tester implementovat pomocí metod knihovny pro manipulaci s elementy GUI. V takovém případě by test sestával z kroků vedoucích na různá okna aplikace, přičemž pro každé nově otevřené okno by byla volána metoda `test_overlap()`. Tento postup by zaručoval globální otestování všech oken aplikace. Zároveň lze stejným způsobem vytvořit testy pouze pro určité části aplikace. Je zřejmé, že tvorba takových testů by mohla být časově náročná, a proto by měly testy vzniknout hned na začátku vývoje testované aplikace. Poté by testy mohly být využívány opakovaně při regresním testování a investovaný čas do jejich tvorby na začátku vývoje by měl smysl.

Posledním typem je metoda `fuzz_testing()` plnící funkci fuzz testování. U této metody není třeba implementovat automatické spouštění. Uživatel vždy specifikuje vyžadovaný počet kroků, a pokud chce testování opakovat, jednoduše volá metodu `fuzz_testing()` několikrát za sebou uvnitř jednoho testovacího skriptu.

Kapitola 7

Hodnocení funkčnosti nástroje

V rámci této bakalářské práce vznikl testovací framework, jehož vývoj ale odevzdáním práce nekončí. Cílem této kapitoly je zhodnotit, do jaké míry vytvořený nástroj splnil očekávání. V první řadě je třeba shrnout, k čemu framework v aktuálním stavu slouží a stručně demonstrovat jeho funkčnost. Jelikož framework vznikal cíleně pro potřeby firmy VF, je vhodné zmínit jeho přínos v této firmě i přínos, který pro firmu může mít do budoucna, pokud v jeho vývoji bude pokračovat. Framework je teprve ve svých počátcích, důležité je tak i zamyslet se nad možnostmi pro jeho rozšíření, které by mohly testerům ve firmě ještě více usnadnit práci, což je hlavním účelem vzniku tohoto nástroje.

7.1 Demonstrace funkčnosti frameworku

Použití jednotlivých částí frameworku je ukázáno přímo v sekcích, které se věnují implementaci dané části. Obecně může plnit nástroj při testování 4 různé funkce:

- manipulace s prvky GUI,
- kontrola výskytu řetězcových literálů v popiscích aplikace,
- ověřování vzájemného překrývání grafických prvků a
- fuzz testování aplikace.

Je zřejmé, že fuzz testování, testování výskytu řetězcových literálů a testování vzájemného přetékání grafických prvků nebudou používány tak často jako první zmíněný bod. Práce testera s frameworkem bude především znamenat práci s knihovnou metod pro manipulaci s elementy GUI. Očekává se, že pro každé okno nebo sadu oken aplikace vznikne kolekce testů GUI, které budou psány pomocí metod knihovny. Tyto testy poté najdou využití při regresním testování, protože mohou být spouštěny opakovaně. Testování výskytu řetězcových literálů v popiscích aplikace by naopak mělo být používáno globálně pro celou aplikaci, jak je popsáno v Sekci 6.7. Ke globálnímu testování výskytu řetězcových literálů v popiscích dojde pravděpodobně jen párkrát během celého vývoje aplikace. Případně k tomuto testování může dojít až při dokončování aplikace. Opravy zjištěných chyb by totiž neměly být časově náročné a hlavně by neměly mít dopad na funkčnost aplikace. Testování přetékání prvků by mělo být součástí regresních testů, a proto je vhodné, aby testy vznikly hned na začátku vývoje aplikace. Fuzz testování se od ostatních odlišuje a jeho používání se očekává až ve chvíli, kdy je aplikace plně funkční. Obecná pravidla pro tvorbu testovacích skriptů vyjadřuje následující kód:

```
class TestsClassName(unittest.TestCase):
    def test_name(self):
        ... volani metod knihovny frameworku ...

    def test_name1(self):
        ... volani metod knihovny frameworku ...

print("STARTED")
#nacteni vsech testu z tridy TestsClassName
suite = unittest.TestLoader().loadTestsFromTestCase(TestsClassName)
#spusteni nactenych testu
unittest.TextTestRunner(verbosity=2).run(suite)

... volani metod frameworku mimo knihovnu ...
print("FINISHED")
```

Jde pouze o příklad syntaxe testovacího skriptu, který znázorňuje, na jakém místě používat různé typy metod frameworku. Neočekává se ovšem, že by všechny tyto typy metod byly používány zároveň. Naopak, budou velmi pravděpodobně různé typy metod používány zvlášť v různých testovacích skriptech.

Příklady použití nástroje

Jako první ukázku použití frameworku při testování jsem zvolila druhý z případů užití uvedených v Sekci 5.1, který popisuje vyplnění formuláře pro vytvoření nové položky. Konkrétně jde o situaci, kdy je třeba vytvořit nového zákazníka a ověřit, že se právě přidaný zákazník vyskytuje v seznamu zákazníků:

```
from Buttons import *
from Boxes import *
from OtherControls import *

class CustomerInsertTests(unittest.TestCase):
    def setUp(self):
        #kliknuti na tlacitko "Zakaznici" vedouci na okno se seznamem knih
        click_button_bytext("Zákazníci")

    def test_insert_customer(self):
        #kliknuti na tlacitko "New" pro ziskani okna
        #s formularem pro noveho zakaznika
        click_button_bytext("New")

        #vyplneni vsech poli formulare
        fillin_textbox_byid("Adam Marek", "Name")
```

```
17          fillin_textbox_byid("Veselá 2", "Street")
18          fillin_textbox_byid("Brno", "City")
19          click_combobox_byid("Country")
20          click_textblock_bytext("ID")
21          click_radio_bytext("Muž")
22
23          #ulozeni noveho zakaznika pomoci tlacitka "Save"
24          click_button_bytext("Save")
25          #kontrola, zda novy zakaznik existuje v seznamu
26          check_textblock_bytext_isenabled(self, "Adam Marek")
27
28  print("STARTED")
29  suite = unittest.TestLoader().loadTestsFromTestCase(CustomerInsertTests)
30  unittest.TextTestRunner(verbosity=2).run(suite)
31  print("FINISHED")
```

Jednotlivé kroky testu a jejich význam jsou popsány přímo v komentářích zdrojového kódu. Pokud by některý prvek formuláře nebyl přístupný pro psaní, test by skončil s výsledkem `ERROR` a informací, jaká akce se nezdařila. V rámci třídy `CustomerInsertTests` by mělo být ideálně vytvořeno několik testů podobných `test_insert_customer()`. Každý z nich by měl za úkol např. vyplnit pouze část formuláře a následně by docházelo ke kontrole pomocí metody `check_button_bytext_isenabled("Save")`, zda je pro daný případ možné kliknout na tlačítko „Save“, či nikoliv.

Dalším příkladem použití nástroje může být situace, kdy chce uživatel frameworku otestovat, jestli se ve zdrojových kódech aplikace nenachází řetězcové literály, které by při pozdějších úpravách bylo složité vyhledat a změnit. Uživatel proto vytvoří velmi jednoduchý testovací skript, v němž chce otestovat všechny soubory s příponou .xaml uložené v jednom adresáři:

```
1  from TestResources import *
2
3  test_resources(r"absolute\path\to\xaml\files\directory")
```

Třetí ukázka použití frameworku vychází z možnosti testovat vzájemné překrývání elementů okna. Jak bylo zmíněno v Sekci 6.7, je možné tuto funkčnost kombinovat s metodami knihovny, a tím dosáhnout schopnosti otestovat více oken v rámci jednoho testovacího skriptu. Pokud by měla být tímto způsobem otestována všechna okna spadající do části „Knihy“ aplikace BooksApp, vypadal by testovací skript, který bude spuštěn pro úvodní okno aplikace, takto:

```
1  from TestOverlap import *
2  from Buttons import *
3  from Datagrid import *
4
```

```
5  test_overlap() #testovani uvodniho okna
6  click_button_bytext("Knihy")
7  test_overlap() #testovani okna se seznamem knih
8  click_button_bytext("New")
9  test_overlap() #testovani okna s formularem pro vytvoreni knihy
10 click_button_bytext("Knihy")
11 click_datagridcell_byindex(1)
12 click_button_bytext("Open")
13 test_overlap() #testovani okna detailu vybrane knihy
```

Poslední ukázkou je příklad fuzz testování. Pokud chce uživatel frameworku spustit v aplikaci dvakrát po sobě fuzz testování, které pokaždé vykoná 500 kroků, je třeba vytvořit následující testovací skript:

```
1  from FuzzTesting import *
2
3  fuzz_testing(500)
4  fuzz_testing(500)
```

Všechny výše uvedené příklady použití jsou implementovány v souboru `Test.py`, který je defaultně otevřen v testovacím prostředí a může být bez velkých úprav spuštěn. Dále lze v adresáři `Scripts/` najít adresář `VFAppTests/`, který obsahuje testy vytvořené pomocí frameworku pro testování jedné z firemních aplikací. Mimo to je v adresáři `Scripts/` uložen soubor `MoreTests.py`, v němž jsou implementovány další testy aplikace BooksApp pro ukázku.

7.2 Přínos frameworku

Po dokončení implementace každé z částí frameworku byla funkčnost vytvořených metod vyzkoušena na vybrané firemní aplikaci, což vedlo k četným úpravám metod i způsobu jejich spouštění. Tímto experimentováním byly metody frameworku otestovány a zároveň byl ověřován jejich přínos pro proces testování ve firmě, který shrnuje tato sekce.

Hlavním přínosem frameworku pro firmu je jednoznačně zavedení automatizace do procesu testování. Před vznikem frameworku byly ve firmě používány výhradně postupy manuálního testování, které sice přinášely dobré výsledky, ale zároveň měly i spoustu nevýhod, a to především časovou a s ní související finanční náročnost. Cílem frameworku samozřejmě nebylo automatizovat celý proces testování. Hlavním účelem vzniku bylo zavést principy automatického testování do vývojového procesu ve firmě, které by testerům ušetřily čas. Motivací k vytvoření frameworku, který by automatizoval právě GUI, byl fakt, že testeři ve firmě stráví přes 80% pracovního času testováním aplikace jako celku, což jinak řečeno znamená, že přes 80% času věnují práci s GUI. To obvykle znamená vyvíjenou aplikaci různě „proklikávat“ a na základě toho hledat a reportovat chyby. Právě proto je nejdůležitější částí frameworku knihovna metod pro manipulaci s elementy, která poskytuje metody vykonávající klikání v aplikaci. Využití automatického testování přináší také atraktivnější práci pro testera. Na místo zdlouhavého klikání v aplikaci může za pomoci frameworku

tvořit automatické testy, které budou aplikaci „proklikávat“ za něj. Takto vytvořené testy jsou navíc použitelné opakovaně, což vede k výraznému ušetření času při často opakovaném testování některých oken. Při častém testování některých oken zároveň vzrůstá šance přehlédnutí chyby testerem, čemuž by měly automatické testy zabránit.

Velkým přínosem je zároveň zavedení fuzz testování do vývojového procesu firmy, které dosud žádným způsobem využíváno nebylo. Používání fuzzingu je tak novinkou pro testery i vývojáře a jeho dopad na lepší kvalitu firemních aplikací se ukáže až po delší době od jeho zavedení. Při experimentování s fuzz testováním, které nabízí framework, se však některé již dříve nalezené chyby aplikace podařilo odhalit, a to za poměrně krátký čas, což ukazuje, že využití fuzzingu má při testování smysl.

7.3 Možnosti pro rozšíření

Aktuální verze frameworku je pro testování dobře použitelná, ale zároveň nabízí několik možností pro rozšíření. Prvním návrhem na rozšíření, které se zavedením automatického testování nepochybně souvisí, je využívání testů vytvořených pomocí frameworku při *CI — Continuous Integration*[1]. Automatické testování se při CI využívá v rámci každé nové integrace. Pro vývojový proces typicky existuje CI server, na kterém dochází k integraci jednotlivých částí. Na CI serveru může být zároveň uložena i sada testů, jež je automaticky spouštěna při každé nové integraci. Ať už spuštěné testy narazí, či nenarazí na chybu, o jejich výsledku a úspěšnosti nové integrace jsou informováni všichni členové, což vede k lepší spolupráci v týmu. Framework by tak měl být do budoucna rozšířen takovým způsobem, aby zaručoval možnost implementovat sadu testů, která by při integraci mohla být automaticky spuštěna. Aktuální verze frameworku umožňuje tvořit a následně spouštět kolekce testů, proto je možné, že lze framework pro CI využívat bez dalších úprav. Tato možnost však nebyla otestována, a proto je zmíněna pouze jako možné rozšíření.

Další možností pro vylepšení frameworku je rozšířit funkčnost fuzz testování. Nápad zavést fuzzing do procesu testování ve firmě byl členy vývojového týmu vnímán pozitivně a do budoucna se očekává jeho časté využívání. Tento fakt vede na myšlenku implementovat kromě jednoduchého generování náhodných vstupů aplikace i další typy fuzz testování. Tzn. implementovat fuzzery, které budou generovat vstupy na základě znalostí o testované aplikaci, jako např. na základě znalosti vnitřní struktury aplikace, struktury vstupních dat nebo na základě informací o míře pokrytí již provedenými testy.

Dále by bylo možné framework rozšířit o využití *DI — Dependency Injection*[2]. Takové rozšíření by umožňovalo uvnitř testu zpřístupnit instance všech ViewModelů aplikace a tím i data uvnitř ViewModelů. Při testování by to znamenalo možnost kontrolovat, zda se data zobrazená v GUI aplikace správně objevují i v odpovídajícím ViewModelu. Stejně tak by mohlo dojít ke kontrole předávání dat i opačným směrem, tedy z ViewModelů do jednotlivých oken (View). To lze využít např. při vyplnění formuláře a následné kontrole, že se vyplněná data správně objevují i ve ViewModelu daného formulářového okna.

[1] Continuous Integration — postup vývoje software, při kterém jednotliví členové týmu často integrují svoji práci [20]

[2] Dependency Injection — způsob vytváření závislostí mezi komponentami, kdy zodpovědnost za získávání závislých komponent přebírá DI kontejner [21]

Kapitola 8

Závěr

Cílem práce bylo v první řadě seznámit se s principy testování grafického uživatelského rozhraní (GUI) a následně prozkoumat existující testovací frameworky zaměřené na GUI. Dále bylo třeba porovnat dynamické jazyky platformy .NET a ověřit možnosti jejich využití pro testování GUI. Dalším úkolem bylo provést analýzu požadavků na výsledný framework, následně požadavky definovat a na základě nich vytvořit návrh nástroje. Na to navazovala fáze implementace nástroje, jejíž součástí bylo i vytvoření jednoduché aplikace, na které by mohly být testy napsané pomocí frameworku předvedeny. Poslední částí práce bylo ověřit funkčnost vytvořeného nástroje, zhodnotit jeho přínos a navrhnout možnosti na jeho rozšíření.

Poznatky získané během teoretické části práce vedly k výběru frameworku White a jazyka IronPython pro implementaci výsledného nástroje. V souladu s definovanými požadavky a vytvořeným návrhem vznikl v implementační části práce framework poskytující 4 typy metod využitelné pro testování GUI. Prvním typem jsou metody spadající do knihovny metod pro manipulaci s elementy GUI, které slouží pro simulaci uživatelských vstupů aplikace, tedy metody umožňující např. kliknout na tlačítko, vyplnit textbox nebo zaškrtnout checkbox. Druhý typ metod poskytuje možnost kontrolovat zdroje dat popisků zobrazených v aplikaci, což by mělo zabránit výskytu řetězcových literálů v definici popisků a vést k lépe spravovatelnému kódu. Třetí typ metod zajišťuje testování vzájemného překrývání prvků viditelných v okně aplikace a pomáhá tak odhalit chyby vzniklé při úpravách aplikace. Poslední možností využití frameworku je fuzzing, neboli náhodné „klikání“ v aplikaci, které cílí na odhalení bezpečnostních chyb. Všechny typy metod byly během vývoje postupně vyzkoušeny, případně upraveny a v poslední části práce byl zhodnocen jejich přínos.

Cílem vzniku frameworku bylo ulehčit proces testování GUI testerům ve firmě VF, což se především díky vytvoření knihovny metod pro manipulaci s elementy GUI a fuzz testování povedlo. Hlavním přínosem práce je zavedení automatizace do procesu testování ve firmě, kde byly před vznikem frameworku využívány principy pouze manuálního testování. Vytvořený framework je již nyní využíván pro testování GUI aplikací firmy VF a jeho vývoj bude i nadále pokračovat. Do budoucna by prvním krokem měla být úprava frameworku tak, aby mohly být testy s jeho pomocí vytvořené automaticky spouštěny při *Continuous Integration*, čili při integrování jednotlivých částí vyvíjené aplikace.

Literatura

[1] *EggPlant* [online]. [cit. 2020-2-22]. Dostupné z: https://www.eggplantsoftware.com.

[2] *F# dokumentace* [online]. [cit. 2019-12-18]. Dostupné z: https://docs.microsoft.com/cs-cz/dotnet/fsharp/.

[3] *Introduction to Prism* [online]. [cit. 2020-3-31]. Dostupné z: https://prismlibrary.com/docs/.

[4] *IronRuby* [online]. [cit. 2019-12-18]. Dostupné z: http://ironruby.net.

[5] *IronScheme* [online]. [cit. 2019-12-18]. Dostupné z: https://github.com/IronScheme/IronScheme.

[6] *JScript .NET Fundamentals* [online]. [cit. 2020-2-22]. Dostupné z: http://www.functionx.com/jscript/Lesson01.htm.

[7] *Maveryx - Test It Simple* [online]. [cit. 2020-2-22]. Dostupné z: https://www.maveryx.com.

[8] *MoonSharp* [online]. [cit. 2020-2-20]. Dostupné z: https://www.moonsharp.org/.

[9] *Robot Framework* [online]. [cit. 2020-3-10]. Dostupné z: https://robotframework.org.

[10] *The Selenium Browser Automation Project* [online]. [cit. 2020-1-20]. Dostupné z: https://selenium.dev/documentation/en/.

[11] *Sikuli Documentation* [online]. [cit. 2020-2-22]. Dostupné z: http://doc.sikuli.org.

[12] *Test Automation for All* [online]. [cit. 2020-1-20]. Dostupné z: https://www.ranorex.com.

[13] *TestStack.White* [online]. [cit. 2020-1-20]. Dostupné z: https://teststackwhite.readthedocs.io/en/latest/.

[14] *Unittest — Unit testing framework* [online]. [cit. 2020-3-25]. Dostupné z: https://ironpython-test.readthedocs.io/en/latest/library/unittest.html.

[15] *VF NUCLEAR* [online]. [cit. 2020-2-05]. Dostupné z: https://www.vfnuclear.com/cz/.

[16] *Začínáme s WPF* [online]. [cit. 2020-1-20]. Dostupné z: https://docs.microsoft.com/cs-cz/visualstudio/designers/getting-started-with-wpf?view=vs-2019.

[17] Aebersold, K. *Test Automation Frameworks* [online]. [cit. 2019-12-10]. Dostupné z: https://smartbear.com/learn/automated-testing/test-automation-frameworks/.

[18] Ammann, P. *Introduction to software testing.* New York: Cambridge University Press, 2008. ISBN 978-0-521-88038-1.

[19] Foord, M. J. a Muirhead, C. *IronPython in Action.* Greenwich, CT: Manning Publications Co., 2009. ISBN 978-1-933988-33-7.

[20] Fowler, M. *Continuous Integration* [online], 10. září 2000. Revidováno 1. 5. 2006 [cit. 2020-5-6]. Dostupné z: https://martinfowler.com/articles/continuousIntegration.html.

[21] Fowler, M. *Inversion of Control Containers and the Dependency Injection pattern* [online], 23. ledna 2004 [cit. 2020-5-6]. Dostupné z: https://martinfowler.com/articles/injection.html?

[22] Godefroid, P. Fuzzing: Hack, Art, and Science. *Commun. ACM.* New York, NY, USA: Association for Computing Machinery. leden 2020, sv. 63, č. 2, s. 70–76. DOI: 10.1145/3363824. ISSN 0001-0782.

[23] Kitner, R. *Typy testování software (třídění testů)* [online]. [cit. 2019-12-10]. Dostupné z: https://kitner.cz/testovani_softwaru/typy-testovani-software-trideni-testu/.

[24] Petříček, T. *F# Language Overview* [online]. [cit. 2019-12-18]. Dostupné z: http://tomasp.net/articles/fsharp-i-introduction/article.pdf.

[25] Pezzè, M. a Young, M. *Software testing and analysis : process, principles, and techniques.* Hoboken: John Wiley & Sons, 2008. ISBN 978-0-471-45593-6.

[26] Richter, J. *.NET Framework : programování aplikací.* 1. vyd. Praha: Grada, 2002. ISBN 80-247-0450-1.

[27] Rungta, K. *Automation Testing Vs. Manual Testing: What's the Difference?* [online]. [cit. 2019-11-21]. Dostupné z: https://www.guru99.com/difference-automated-vs-manual-testing.html.

[28] Spillner, A. *Software testing foundations : a study guide for the certified tester exam : foundation level, ISTQB compliant.* 4th edition. Santa Barbara, CA: Rocky Nook, 2014. ISBN 9781937538422.

[29] Čápka, D. *Lekce 1 - Úvod do C# a .NET frameworku* [online]. [cit. 2019-12-18]. Dostupné z: https://www.itnetwork.cz/csharp/zaklady/c-sharp-tutorial-uvod-do-jazyka-a-dot-net-framework.