VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

FRAMEWORK PRO TESTOVÁNÍ GRAFICKÉHO UŽIVATELSKÉHO ROZHRANÍ

FRAMEWORK FOR GRAPHICAL USER INTERFACE TESTING

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **ERIK BÁČA**
AUTHOR

VEDOUCÍ PRÁCE **Ing. JITKA KOCNOVÁ**
SUPERVISOR

BRNO 2022

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačových systémů (UPSY) Akademický rok 2021/2022

Zadání bakalářské práce

Student: **Báča Erik**
Program: Informační technologie
Název: **Framework pro testování grafického uživatelského rozhraní**
Framework for Graphical User Interface Testing
Kategorie: Umělá inteligence

Zadání:

1. Seznamte se s existujícími frameworky pro testování grafického uživatelského rozhraní (GUI).
2. Navrhněte framework pro testování GUI, který bude pro generování a vyhodnocování testů využívat vhodně zvolené algoritmy z oblasti Soft Computingu.
3. Navržený systém implementujte.
4. Vytvořte sadu testů, na kterých demonstrujete funkčnost navrženého frameworku.
5. Systém otestujte, zhodnoťte dosažené výsledky a navrhněte možná rozšíření.

Literatura:

- Dle pokynů vedoucího.

Pro udělení zápočtu za první semestr je požadováno:

- Splnění bodů 1 a 2 zadání.

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

Vedoucí práce: **Kocnová Jitka, Ing.**
Vedoucí ústavu: Sekanina Lukáš, prof. Ing., Ph.D.
Datum zadání: 1. listopadu 2021
Datum odevzdání: 11. května 2022
Datum schválení: 29. října 2021

Abstrakt

Tato práce popisuje vývoj frameworku určeného k testování grafického uživatelského rozhraní pomocí algoritmů z oblasti Soft Computingu. Vývoj je rozdělen do čtyř fází. První fáze se týká seznámení s již existujícími frameworky pro testování GUI a jejich rozboru. V druhé fázi proběhne výběr vhodných nástrojů pro vývoj, vhodných algoritmů a návrh implementace. Třetí fáze se následně zabývá samotnou implementací frameworku a v poslední fázi proběhne otestování systému, zhodnocení dosažených výsledků a návrh možných rozšíření.

Abstract

This thesis is describing the development of a framework for graphic user interface testing with usage of Soft Computing algorithms. Development is divided into four phases. The first phase is acquaintance with existing GUI testing frameworks and their analysis. The second phase is about choosing appropriate technologies for development, appropriate algorithms and implementation design. Then there is the framework implementation phase itself and last phase with the testing, result evaluation and improvements proposal.

Klíčová slova

Testování, GUI, uživatelské rozhraní, automatické testy, Soft Computing, C#, Windows Presentation Foundation

Keywords

Testing, GUI, user interface, automatic tests, Soft Computing, C#, Windows Presentation Foundation

Citace

BÁČA, Erik. *Framework pro testování grafického uživatelského rozhraní.* Brno, 2022. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Jitka Kocnová

Framework pro testování grafického uživatelského rozhraní

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením paní Ing. Jitky Kocnové. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Erik Báča
9. května 2022

Poděkování

Rád bych poděkoval paní Ing. Jitce Kocnové za velmi užitečné rady, odborné konzultace a vstřícnost. Další poděkování patří mé rodině a přátelům za pomoc a podporu při psaní práce. Poslední poděkování patří firmě Bagira Systems, která mi umožnila otestovat framework na reálných systémech.

Obsah

Kapitola 1

Úvod

Počítače od jejich vzniku ušly obrovský kus cesty. Počítačový hardware je dnes na nesrovnatelně vyšší úrovni, než byl před lety a díky tomu je dnes možné vytvářet velmi složitý a kvalitní software. Jak ale zajistit kvalitu složitého softwaru? Správnou odpovědí na tuto otázku je testování.

Dnešní vývoj softwaru se bez kvalitního testování neobejde. Pro jednoduché aplikace může veškeré testování zastat sám programátor ručně. Velké korporátní firmy mají celé týmy testerů a do popředí se víc a víc dostává testování automatické. Ať už je testování řešeno jakkoli, lze téměř jistě říci, že se bez něj kvalitní software vytvořit nedá.

V dnešní době nabízí vývoj softwaru mnoho firem, většina z nich se však potýká s nedostatkem zaměstnanců. To způsobuje, že čas každého zaměstnance je velmi důležitý a vhodnou metodou pro zajištění co nejvyšší efektivity zaměstnanců je automatizace. Testování softwaru patří k oblastem, ve kterých je automatizace uplatnitelná ve velké míře. Vývojem a výběrem vhodných automatizovaných testovacích nástrojů je možné ušetřit čas zaměstanců, zlepšit celý vývojový proces a tím zkvalitnit i samotný vyvíjený software.

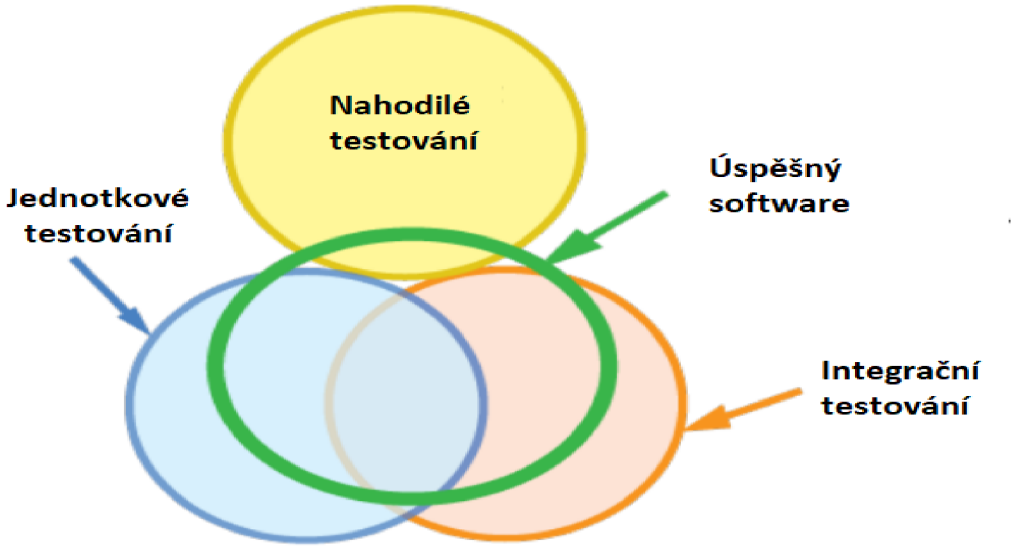
Cílem této práce je vytvořit framework, který testuje uživatelské grafické rozhraní. Framework bude použitelný na mnoho různých aplikací a programů a pro jeho použití se nebude nutné učit žádnou složitou syntaxi. Mimo framework samotný bude práce obsahovat i menší aplikaci, na které bude možné ukázat, jak framework funguje. Testy budou generovány a vyhodnocovány pomocí algoritmů z oblasti Soft Computingu a podle nastavení se bude testovat grafický výstup nebo log dané aplikace.

První část této práce se zabývá seznámením s již existujícím softwarem pro testování GUI a jeho analýzou. Druhá část se zabývá čistě výběrem vhodných technologií a návrhem implementace frameworku a poslední části řeší samotnou implementaci, návrh rozšíření a zhodnocení fungování frameworku.

Kapitola 2

Softwarové testování

Vývojáři softwaru jsou často nuceni využívat procesy systematického testování. To zahrnuje vše od jednotkových testů jednotlivých funkcí až po testování funkčnosti celého programu. Pro většinu vývojářů je úspěšnost testování (jak interního, tak testování zákazníkem) faktorem při získávání práce, povýšení nebo i propuštění. Níže uvedený diagram popisuje vztah mezi úspěšným vývojem softwaru a jeho testováním.



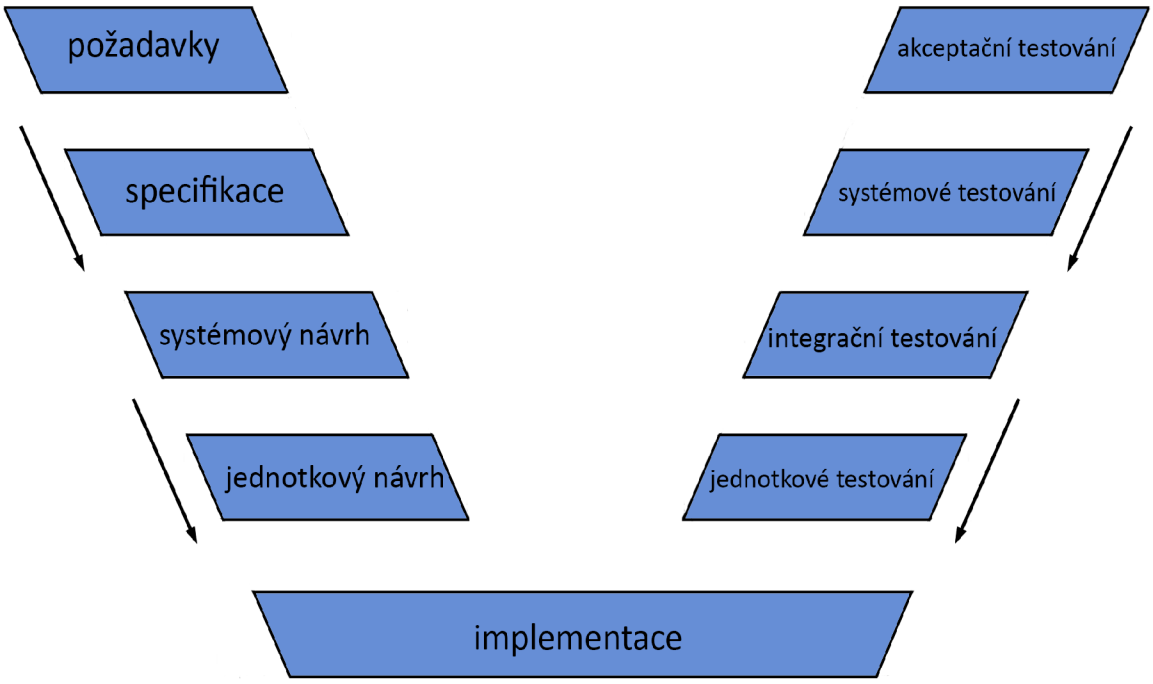
Obrázek 2.1: Vennův diagram softwarového testování

Diagram ukazuje, že šance na vytvoření úspěšného produktu bez systematické, formální testovací strategie existuje. Jsou to případy malých projektů vyvíjených velmi malými týmy s jednoduchými vývojovými standardy. I u takto malých projektů se v produkci objevují chyby. Počet a komplexnost těchto chyb roste spolu s velikostí týmu a komplexností softwaru, a proto je systematické testování nutností. [26]

2.1 Druhy testování

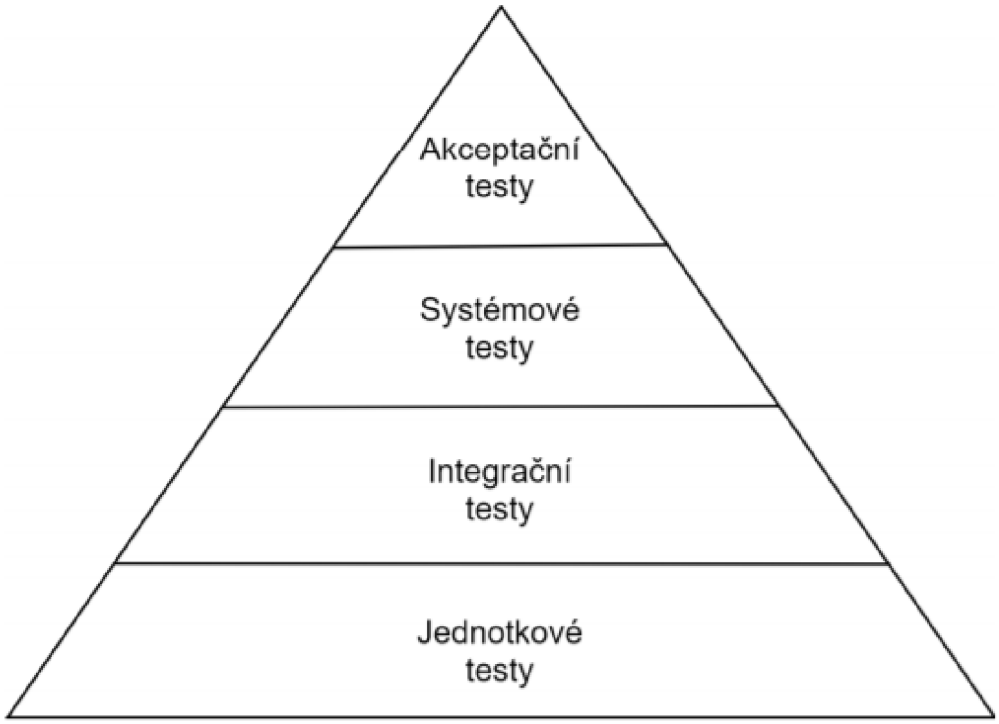
V dnešní době existuje nesčetné množství různých druhů testování. Testy je možné dělit podle fáze testování (jednotkové testy, integrační testy, systémové testy či akceptační testy), podle znalosti kódu (bílá skříňka, černá skříňka), podle způsobu realizace (manuální, automatizované), nebo podle dimenzí kvality (funkčnost, výkon, použitelnost). Nyní bude následovat rozbor těchto druhů vycházející z článku *Typy testování software* [16].

První čtyři druhy vycházejí z tzv. Vodopádového modelu (V-modelu), který reprezentuje celý životní cyklus softwaru. Testovaní je podle něj, stejně jako návrh softwaru, rozděleno do čtyř úrovní, čímž se snaží zdůraznit důležitost testování. Každá úroveň má svoje specifické vlastnosti: cíle testování, testované objekty, testovací prostředí, či strategii testování.



Obrázek 2.2: V-model testovacího cyklu

S těmito druhy se pojí i koncept tzv. testovací pyramidy. To je koncept zabývající se ideálním počtem testů jednotlivých druhů. Tvar pyramidy odkazuje na skutečnost, že na nižších úrovních by mělo být více testů než na úrovních vyšších. Tato skutečnost vyplývá z faktu, že na nejnižších úrovních běží testy nejrychleji a jsou nejméně finančně náročné.



Obrázek 2.3: Testovací pyramida

Rozdělení do úrovní se může lišit v závislosti na testovaném systému, či vývojové strategii, ale obvykle obsahuje tyto části:

Jednotkové testování

Jde o testování nejmenších částí (jednotek) softwaru například metod, funkcí, či tříd. Probíhá ještě před integrací testované jednotky do systému. Testy jsou často automatizované a obvykle je vytváří sám vývojář dané jednotky. Testovaná jednotka by měla být nejlépe úplně izolovaná od zbytku systému. Na tomto druhu testování je založeno programování řízené testy (test-driven development), což je přístup k vývoji, kdy jsou nejdříve vytvořeny jednotkové testy a až následně se implementují dané jednotky systému.

Integrační testování

Integrační testování ověřuje, zda moduly uvnitř systému fungují přesně tak, jak by měly. Cílem není ověřit, zda moduly pracují správně jednotlivě, ale v součinnosti uvnitř systému. Testování probíhá po integraci modulů do systému a jsou stále spíše doménou programátorů. Využívají se především na velkých projektech, které takových modulů obsahují mnoho.

Systémové testování

Systémové testování se zaměřuje na chování a schopnosti celého systému nebo produktu a ověřuje funkcionální i nefunkcionální vlastnosti systému. Funkcionální systémové testování často zahrnuje end-to-end testy – testy, které testují celou aplikaci s využitím reálných

scénářů za účelem ověřit komunikaci aplikace s hardwarem, sítí, databází a dalšími aplikacemi. Tato úroveň testování je obvykle prováděna nezávislými testery z důvodu zajištění objektivity testování. [27]

Akceptační testování

Akceptační testy ověřují splnění požadavků zákazníka. Testování probíhá po dokončení celého systému a dělí se na dva hlavní druhy - uživatelské a business. Uživatelské akceptační testování probíhá na straně zákazníka. Ten po předání hotového systému kontroluje, zda dodaný systém splňuje jeho požadavky. Druhý druh akceptačního testování ověřuje, zda systém projde uživatelským akceptačním testováním.

Následující dva druhy testování se zaměřují na dimenze kvality. Dimenze kvality jsou určité vlastnosti softwaru, které se vztahují k požadavkům zákazníka. Takových dimenzí existuje mnoho (výkon, spolehlivost, funkčnost, podporovatelnost atd.), ale nejdůležitější z nich jsou tyto:

Výkonové testování

Výkonové testování ověřuje rychlost systému, jeho reakční dobu a využití prostředků. Požadavky na tento systém obvykle specifikuje zákazník. V menších systémech to může být specifikováno jen neformálně („systém se nesmí sekat“), v rozsáhlejších systémech, kde je výkon nejdůležitější, může být specifikován přesný časový limit náročnějších operací.

Spolehlivostní testování

Spolehlivost je pravděpodobnost bezporuchového provozu softwaru ve specifickém čase a prostředí. Každý software vyžaduje určitou dávku této spolehlivosti. Spolehlivost internetového blogu studenta střední školy nemusí být na moc vysoké úrovni, naproti tomu software letecké společnosti, který koordinuje cesty letadel, tak aby se nesrazily, musí být tak spolehlivý jak to jen jde. Z těchto důvodů bývá testování spolehlivosti jednou z nejdůležitějších fází životního cyklu softwaru. [24]

Další dva druhy testování se zabývají znalostí kódu. Volba techniky zabývající se znalostí kódu závisí na několika faktorech. Mezi tyto faktory například patří typ komponenty/systému, její složitost, úrovně a typy rizik, předchozí zkušenosti s použitím technik nebo očekávané typy defektů pro daný typ komponenty/systému.

Testování bílé skříňky

Testování bílé skříňky k vytváření testů využívá znalost implementace jednotlivých testovaných komponent. Mezi takové testování patří například testy příkazů, či testy rozhodování a je pro ně nutná znalost testovaného kódu. Největší výhodou je možnost postavit testy na přísných definicích, matematických analýzách nebo přesném měření. To je umožněno analýzou kódu, která probíhá ještě před vytvářením testů. Nevýhodou tohoto testování je možnost vytváření testů a testování až v pozdních fázích životního cyklu softwaru. [23]

Testování černé skříňky

Testování černé skříňky je založeno na vytváření testů čistě podle specifikace nebo dokumentace testovaných komponent. Specifikace popisuje, jaké výstupy by měla komponenta vracet po zadání specifických vstupů. Žádné informace týkající se vnitřní implementace komponenty nejsou brány v potaz. Testování se používá na základě rozdělení dat do tzv. tříd ekvivalence, analýzy hraničních hodnot, či testování dle rozhodovací tabulky. Výhodou je například možnost testovat bez analýzy implementace, nicméně nevýhody pramení z neznalosti vstupů, které by mohly způsobovat problémy.

Poslední dva druhy testování se zabývají způsobem realizace testů (automatické, manuální). Jelikož se manuálním testům bohužel nemůžeme plně vyhnout, je realizace testů hlavně o výběru těch, které se vyplatí automatizovat.

Manuální testování

Manuální testování je jeden z nejzákladnějších a nejdůležitějších typů testování. Testování realizují lidé k tomu určení (testeři). Ověřují se zde různé scénáře, vstupní data apod., vše na základě lidských zdrojů bez využití speciálního softwaru. Největší nevýhody spočívají ve velké časové náročnosti a nemožnosti automatizace. [17]

Automatické testování

Automatické testování je založeno na využití specifického softwaru k automatickému testování. Testovací scénáře jsou vytvářeny tak, aby je nemusel provádět člověk (jsou prováděny programem). Hlavní výhody spočívají ve velmi vysoké rychlosti a jednoduchém opakování testovacích scénářů. Automatizovat jde dokonce i samotné spouštění testů. Téměř všechny dnešní vývojové firmy využívají nějaký verzovací systém, pro uchování informací o tom, kdo, kdy a jakým způsobem změnil určitou část implementace. Většina takových systémů dovoluje při nahrání nové verze implementovaného systému automaticky spustit testy a tím zjistit, zda nová verze nerozbila nějakou starší funkcionalitu.

Tato práce se bude dále zabývat automatizovanými testy černé skříňky grafického uživatelského rozhraní, protože framework, který bude v této práci navržen, by měl být použitelný bez znalosti kódu testované aplikace.

2.2 Testování GUI

Uživatelské rozhraní (UI) umožňuje lidem ovládat elektronická zařízení. Grafické uživatelské rozhraní (GUI) je jeho grafickou (vzhledovou) nadstavbou. Složité komplexní systémy obvykle vyžadují složité komplexní grafické rozhraní a jediná možnost, jak zajistit správnou funkčnost tohoto rozhraní je jeho testování. Testování grafického rozhraní je aktivita, při které se tester snaží napodobit uživatele a zkouší, jestli jeho úkony vyvolávají správné reakce. Toto testování, je možné stejně jako většinu ostatních druhů testování, z určité části automatizovat. Pro tuto automatizaci existují různé nástroje, ze kterých bude vycházeno při návrhu i implementaci frameworku. V následující podčásti budou popsány ty nejběžnější. [21]

2.2.1 Existující nástroje pro testování GUI

Appium

Appium je open-source nástroj pro automatizaci testů pro nativní, hybridní, mobilní webové i desktopové aplikace. Mezi jeho hlavní výhody patří kompatibilita s velkou škálou programovacích jazyků, jako například Objective-C, JavaScript (Node), PHP, Python, Ruby a C# nebo multiplatformnost. Appium umožňuje psát testy na platformách jako: iOS, Android nebo Windows za použití stejné Application Programming Interface (dále API), což umožňuje velkou znovupoužitelnost kódu. Hlavním důvodem těchto výhod je tzv. **Klient-Server architektura**. Appium je webový server s REST API, ten poslouchá příkazy, následně je provádí a vrací HTTP odpovědi s výsledky. Tato architektura umožňuje psát testy ve všech programovacích jazycích, které dokáží komunikovat s API.[1]

Selenium

Selenium je projekt zastřešující velké množství nástrojů a knihoven pro podporu automatizace webových prohlížečů. Používá se mimo jiné k automatizaci testů. Selenium nabízí více druhů testování, jako například: akceptační, výkonnostní, či modulové (viz. kapitola 2). Umožňuje spuštění testů paralelně na více zařízeních pomocí tzv. Gridu. To nám dovolí testovat stránky v různých prohlížečích s různými konfiguracemi zároveň. Přes značný počet výhod má Selenium ale i velkou nevýhodu: nemožnost testování desktopových či mobilních aplikací. [5]

FitNesse

FitNesse je nástroj pro specifikaci a verifikaci akceptačních kritérií aplikací. Specifikace (akceptační testy) jsou psány stylem wiki, díky čemuž jsou propojeny všechny zainteresované strany v tvorbě a dodání aplikace. Hlavní cíl nástroje je zapojit do tvorby specifikace všechny strany (včetně neprogramátorů), vytvořit a otestovat požadavky, co nejvíce odpovídající požadavkům zákazníka. FitNesse pracuje v úrovni těsně pod GUI a testuje, zda různé vstupy do aplikace vyvolají správné výsledky. [22]

Squish

Squish je komerční nástroj pro automatické regresní a systémové testování GUI. Podporuje všechny druhy aplikací napříč platformami. Jako největší výhodu tohoto nástroje lze považovat rozpoznávání objektů v aplikaci. To umožňuje správné testování nehledě na to, jak aktuální aplikace v danou chvíli vypadá a tím řeší jeden z největších problémů testování GUI. Podporuje testování desktop, embedded i mobilních aplikací a spoustu technologií pro vytváření GUI, jako Qt, QML, Qt Quick, Qt Webkit, Qt WebEngine, Java Swing, AWT, SWT, RCP, JavaFx apod. [6]

Sahi

Sahi je sada nástrojů pro testování webových, mobilních, Windows, Sap GUI a Java aplikací. Je určená pro rychlou a spolehlivou automatizaci. Hlavním cílem je zrychlit procesy vývoje software, a proto je velmi vhodná pro agilní týmy vývojářů. Sada podporuje skriptovací jazyky Javascript a Java. [4]

Telerik Test Studio

Telerik Test Studio je dalším nástrojem zaměřeným na testování při agilním vývoji. Velkou výhodou nástroje je jeho vhodnost pro quality assurance (dále QA) testery bez velkých zkušeností s programováním. Nástroj umožňuje funkcionální, Api či zátěžové testování a je vytvořen tak, aby jednotlivé strany ve vývoji měly rychlý přístup k tomu co potřebují. Díky tomu může QA tester testovat aniž by musel umět programovat, programátor může psát testovací skripty a manažer má jednoduchý a rychlý přístup k výsledkům testů. [7]

TestStack.White

White je .NET framework. Zaměřuje se na aplikace založené na Win32, WinForms, WPF, Silverlight a SWT platforms. Nespecifikuje žádný skriptovací jazyk či prostředí, takže k testování lze zvolit cokoli z .NET. Zároveň neobsahuje žádnou možnost testovat bez psaní kódu, tudíž bez znalosti programování prakticky není možné nástroj používat. Framework není nadále udržován. [9]

TestComplete

TestComplete slouží k automatickému testování jakékoli desktopové, webové či mobilní aplikace. Umožňuje vytváření testů pomocí nahrávání, či pomocí skriptů v libovolném moderním skriptovacím jazyku (JavaScript, Python apod.). Nabízí pokročilé rozpoznávání objektů pomocí strojového učení a umělé inteligence. [8]

Rapise

Rapise umožňuje automatické testování desktopových, mobilních i webových aplikací a dokonce i testování API. Rapise má vlastní IDE, které velmi ulehčuje vytváření testů, hledání chyb a nasazení na jakoukoli testovanou platformu a podporuje využití dalších nástrojů, jako Selenium, či Appium. Pro rozpoznávání objektů používá umělou inteligenci a přináší spoustu možností jak urychlit a zefektivnit testování. [3]

Kapitola 3

Návrh frameworku

V následující kapitole bude popsán návrh celého frameworku, který dostal pracovní název BTest. V úvodní části kapitoly bude vysvětleno co by framework měl umět. Následně budou zmíněny oblasti Soft-computingu ze kterých bude čerpáno a technologie, které budou použity pro vývoj. V poslední části kapitoly budou jednotlivě popsány nejdůležitější části frameworku.

3.1 Cíle frameworku

Navrhovaný framework umožní rychlé vytváření automatizovaných testů. Vytváření testů bude probíhat na základě tzv. Record and play. Record and play je způsob automatizace testů, kde uživatel vytváří testy nahráním přesných kroků (u kterých má test následně ověřovat funkčnost) a zaznamenáním správného výsledku. Tímto způsobem uživatel nahraje libovolný počet testů, které se budou následně spouštět. Výhodou bude například možnost přidávat testy postupně při vývoji aplikace. Po nahrání testů bude možné spustit testování. Testování bude replikovat testy uživatele a vyhodnocovat, zda vyvolávají správné výsledky. Testy budou replikovány v náhodném pořadí, dokud uživatel testování neukončí. Po ukončení testování framework zobrazí uživateli, které kombinace testů způsobují chybné chování.

3.2 Výběr vhodných technologií

.NET framework

Pro implementaci frameworku byl zvolen programovací jazyk C#. Tento jazyk, stejně jako mnoho dalších, spadá do takzvaného .NET frameworku. Jedná se o soubor technologií pro vývoj na všechny platformy (Windows, Linux i Mac). Největší výhodou jsou knihovny, které tento soubor nabízí. Pro framework jsou z nich nejrelevantnější knihovny Windows Presentation Foundation (WPF) pro tvorbu GUI a Entity Framework pro práci s databází.

Windows Presentation Foundation

Windows Presentation Foundation (WPF) je architektura uživatelského rozhraní, která vytváří klientské aplikace pro stolní počítače. Vývojová platforma WPF podporuje širokou škálu funkcí pro vývoj aplikací, včetně aplikačního modelu, prostředků, ovládacích prvků, grafiky, rozložení, datových vazeb, dokumentů a zabezpečení. To umožní vytvořit framework

s přívětivým uživatelským rozhraním, vyžadující funkčností, vysokou rozšiřitelností a udržitelností. [18]

Entity Framework

Entity Framework je technologií pro tzv. Objektově relační mapování (ORM). ORM zajišťuje nadstavbu nad databází, díky které můžeme ovládat databázi pomocí objektů. To výrazně urychluje práci a v objektově orientovaných jazycích (kterým C# je) je díky tomu kód čitelnější.

MS-SQL

Perzistence dat (např. uchování testů po ukončení aplikace) bude řešena relační databází od společnosti Microsoft. Poskytovatel databáze bude Microsoft SQL Server. Tyto technologie byly zvoleny z důvodu vysoké kompatibility s Entity Frameworkem a .NET Frameworkem obecně.

Model–view–viewmodel

Z důvodu rozšiřitelnosti a udržovatelnosti frameworku byl zvolen návrhový vzor Model–view–viewmodel (MVVM). Tento návrhový vzor umožňuje oddělit aplikační logiku od uživatelského rozhraní. Jednotlivá okna uživatelského rozhraní (Views) jsou ovládány z tzv. Viewmodelů (třídy pro obsluhu Views). Viewmodely dostávají data od repozitářů (abstrakce nad databází) a k práci s nimi používají tzv. Servisy. Servisy jsou třídy, které pomáhají předávat data mezi Viewmodely a případně je vhodně upravovat.

Windows API

C# umožňuje přímé volání Windows API, které bude použito pro automatické provádění testů. Veškeré grafické aplikace a programy pro operační systém (OS) Windows komunikují a ovládají operační systém díky volání Windows API. Tato technologie umožňuje přímý přístup aplikací k OS Windows a hardwaru. To frameworku poskytne možnost automaticky hýbat myší, či klikat na obrazovku. [2]

GeneticSharp

Pro tvorbu genetických algoritmů navrhovaného frameworku bude využita knihovna GeneticSharp. Je to multiplatformní a vícevláknová C# knihovna pro zjednodušení a zrychlení vývoje aplikací, které využívají genetických algoritmů. [14]

EmguCV

EmguCV je multiplatformní balíček postavený na knihovně OpenCV. Tento balíček umožňuje využívat funkcí knihovny OpenCV v .NET programovacích jazycích. OpenCV je knihovna vytvořená v jazyce C++ a obsahuje stovky algoritmů počítačového vidění. Funkce této knihovny budou využity pro porovnání výsledku testu se správným výsledkem. Díky tomu bude framework schopen vyhodnotit, zda byl daný test úspěšný. [12]

3.3 Výběr vhodných Soft-computing algoritmů

Soft-computing je velký soubor algoritmů obsahující např. neuronové sítě, fuzzy logiku a genetické algoritmy. Pro správné pochopení a vhodný výběr algoritmů je dobré seznámit se základními oblastmi tohoto souboru. [20]

3.3.1 Hlavní oblasti Soft-computingu

Fuzzy logika

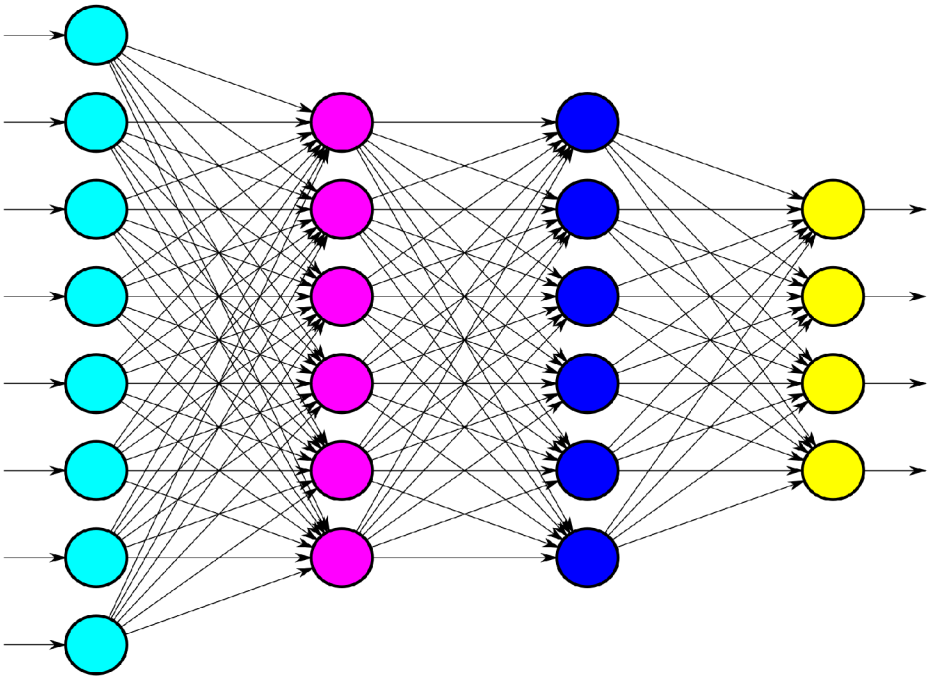
Fuzzy logika je rozšířením Boolovské logiky. S tímto rozšířením přišel Lotfi Zadeh v roce 1965. Pravidla této logiky jsou formulována v přirozeném jazyce a místo striktně binárního vyjádření pravdy, logika obsahuje 0 a 1 pouze jako určité extrémy. To umožňuje označit výrok za více či méně pravdivý. Díky těmto vlastnostem se logika velmi přibližuje reálnému uvažování člověka. [13]

Pokud svítí červená...	Pokud je moje rychlost vysoká...	A pokud je světlo blízko...	Tak tvrdě brzdím.
Pokud svítí červená...	Pokud je moje rychlost nízká...	A pokud je světlo daleko...	Tak udržuji rychlost.
Pokud svítí oranžová...	Pokud je moje rychlost průměrná...	A pokud je světlo daleko...	Tak jemně brzdím.
Pokud svítí zelená...	Pokud je moje rychlost nízká...	A pokud je světlo blízko...	Tak zrychluji.

Obrázek 3.1: Fuzzy logika

Neuronová síť

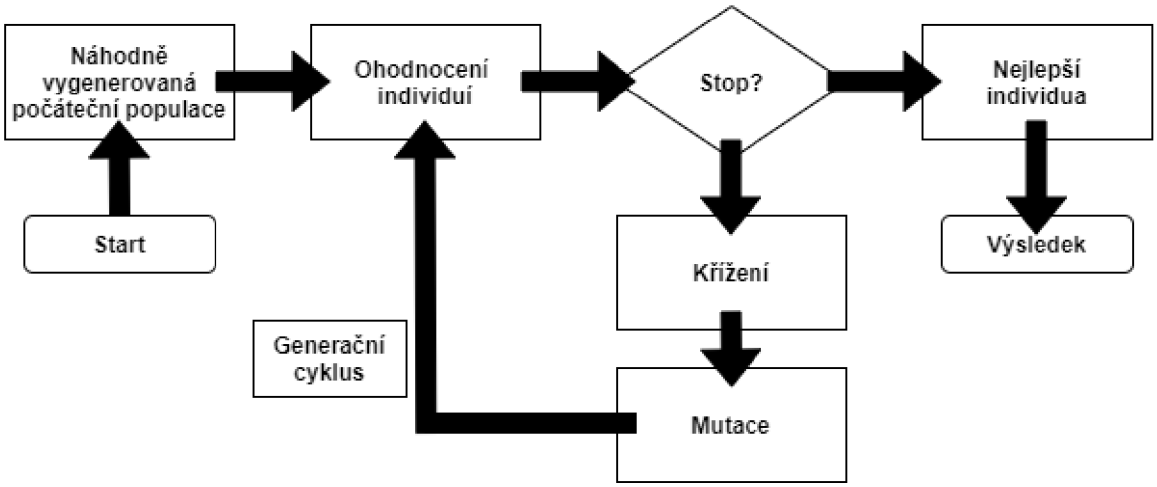
Neuronová síť je vhodná především k řešení problémů, u kterých nelze dosáhnout přesného výsledku, ale je možné se k němu přiblížit. Neuronové sítě jsou definovány jako množina propojených jednoduchých výpočetních jednotek (neuronů), které jsou založeny na podobnosti s biologickým neuronem. Výpočetní schopnosti sítě jsou určeny váhami jednotlivých spojení. Proces nastavování vah sítě je označován jako učení. Množina neuronů a způsob jejich vzájemného propojení je označován jako topologie sítě. Podle vlastností topologií se rozlišují různé architektury sítí. Jednotícím prvkem jsou vstupní a výstupní neurony (vstupní a výstupní vrstva). Zatímco výstupní vrstva neuronů je zřejmou nutností, vstupní vrstva je volitelná. Vstupní hodnoty mohou být vloženy přímo na požadovaná spojení, nicméně používanějším přístupem je pro každý vstup vytvořit vstupní neuron, který má konstantní hodnotu danou vstupem sítě. Na tento vstupní neuron pak může být napojen zbytek sítě. Všechny ostatní neurony jsou označovány jako skryté – angl. „hidden“. [15] [28] [11]



Obrázek 3.2: Neuronová síť

Genetické algoritmy

Genetické algoritmy jsou heuristické postupy, inspirované biologickými procesy (evoluční biologií), které hledají řešení pro úlohy, špatně řešitelné konvenčními metodami. Postupně vytvářejí jednotlivé generace populací, kde každý jedinec populace představuje tzv. kandidátní řešení - možné vhodné řešení úlohy. Následně se různými způsoby v každé populaci vybírá jedinec, který bude sloužit jako rodič dané populace. Populace se generují až do ukončení algoritmu. To může nastat různými způsoby, nejideálnějším ukončením je nalezení vhodného řešení, nicméně může nastat například i ukončení z důvodu dosažení maximálního počtu generací. Maximální počet generací je určen před spuštěním algoritmu. [25]



Obrázek 3.3: Genetický algoritmus[10]

3.3.2 Algoritmy vhodné pro framework

Pro navrhovaný framework bude nejvhodnější použití genetického algoritmu, protože je nejvhodnější volbou pro řešení dané problematiky. Jeho účelem bude najít nejnižší počet testů vedoucích k chybě (viz 3.4). Úvodní populací algoritmu bude nalezená cesta k chybě. Ohodnocující (fitness) funkcí bude délka cesty vedoucí k chybě.

3.4 Návrh nejdůležitějších částí

V dalších odstavcích budou jednotlivě rozebrány nejdůležitější části celého frameworku. Nejdříve jak bude gui frameworku interagovat s uživatelem, jaké možnosti bude uživatel mít, co a kde najde a jak to bude vypadat. Následně se bude práce věnovat tomu, co se bude dít v pozadí. V první části se text bude věnovat technikám použitým ke generování a vyhodnocování testů. Následovat bude soupis a hlubší popis algoritmů, které budou použity.

Nahrávání a replikace testů

Po stisknutí tlačítka „Record test“ začne framework zaznamenávat veškerá další kliknutí. Ke kliknutí si uloží místo a čas (určený uživatelem), který má čekat, než kliknutí při testu provede. Kliknutí budou frameworkem zaznamenávány až dokud uživatel nepotvrdí test klávesou Enter. Nasledně se při replikaci těchto testů bude myš přesouvat automaticky a klikat na stejná místa, která zaznamenal uživatel. K automatickému pohybu myši a klikání bude framework využívat přímé volání Windows API, které C# umožňuje.

Pořadí testů

Hlavní účel navrhovaného frameworku bude ověřit, že testovaný systém funguje správně jako celek. Z toho důvodu neproběhnou všechny testy po spuštění testování pouze jednou. Testy proběhnou vícekrát (přesný počet bude záviset na počtu testů) a v různém pořadí. Díky tomu bude možné zjistit chybu v případě, že díky úkonům z jednoho testu nebudou fungovat úkony druhého. Pořadí testů první iterace určí uživatel a ty následující budou určeny náhodně.

Určení nejnižšího počtu testů vedoucích k chybě

V případě, že testovací framework nalezne chybu, pokusí se nalézt nejnižší možný počet testů a jejich pořadí pro replikaci této chyby. K nalezení nejnižšího počtu testů využije framework genetický algoritmus. Pro pochopení účelu tohoto hledaní si lze představit scénář, ve kterém provedení druhého testu znemožní úspěch testu číslo pět. Když framework tuto chybu nalezne v první iteraci, tak zjistí, že některé testy předcházející testu pět, znemožní úspěch tohoto testu. Pomocí již zmíněného genetického algoritmu následně vyřadí testy jedna, tři, čtyři a do výsledků vypíše, že provedení testu číslo dvě znemožní úspěch testu pět. To usnadní práci vývojáře, protože bude mít přesné kroky, které vedou k chybě a díky tomu ji bude možné rychle opravit.

Vyhodnocování testů

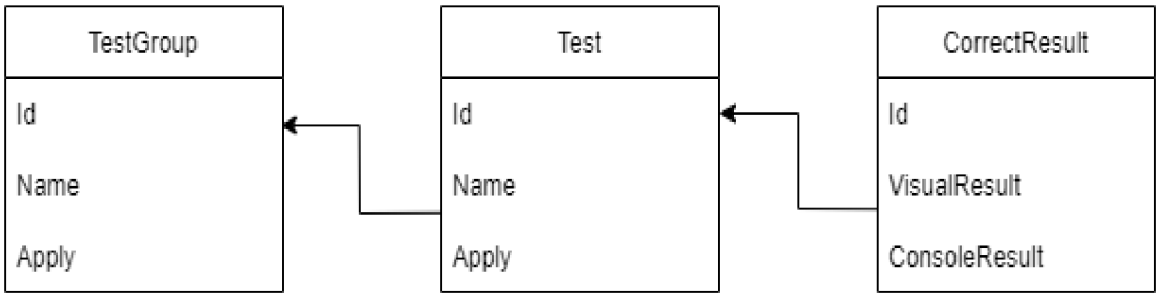
Vyhodnocování testů bude závislé na nastavení frameworku. První možností bude vyhodnocování na základě logu testované aplikace. V tomto případě uživatel po nahrání testů vybere konzoli, do které daná aplikace loguje. Framework si uloží správný výpis konzole a následně po provedení zkontroluje, zda se log neliší od referenčního. Pokud se log nějakým způsobem liší, nastala chyba a framework se pokusí najít minimální počet a přesné pořadí testů, které chybu způsobuje. V opačném případě je výsledek testu úspěšný. Druhou možností bude vyhodnocování na základě vizuálního zobrazení. Po nahrání testu uživatel vybere obrazovku nebo její část, které by se měla obrazovka podobat po provedení testu. Částí obrazovky může být například nově otevřené okno. V případě určité přesnosti (zadané uživatelem) skončí test úspěšně. V opačném případě bude postup stejný, jako u vyhodnocování logu aplikace.

3.4.1 Návrh celkové implementace

Poslední částí těsně před implementací je její celkový návrh. Návrh bude rozebrán od nejnižších vrstev (databáze) až po jednotlivé pohledy (views) uživatelského rozhraní. Bude představeno celkové rozložení frameworku do vrstev a bude popsán jejich smysl.

Návrh databáze

Účel databáze v navrhovaném frameworku bude uchování nahraných testů rozdělených do skupin s jejich výsledky po ukončení aplikace. Pro uchování těchto informací bude databáze rozdělena na tři tabulky - TestGroup, Test a CorrectResult. Tabulka TestGroup bude obsahovat název dané skupiny (name) a zahrnutí skupiny testů do aktuálního testování (apply). Tabulka Test bude uchovávat název testu (name) a zahrnutí testu do aktuálního testování. Poslední tabulkou databáze je CorrectResult. Ta bude uchovávat korektní výsledky testů. Mimo tyto informace bude mít každá tabulka primární klíč id, pro jednoznačnou identifikaci.



Návrh repozitářů

Pro komunikaci s databází a její odstínění od zbytku aplikace, z důvodu zajištění co nejvyšší udržitelnosti a rozšiřitelnosti kódu, budou implementovány repozitáře. Každá tabulka bude mít svůj specifický repozitář, který bude nadstavbou nad repozitářem generickým. Generický repozitář bude obsluhovat základní práci s databází (GetAll, GetById, Update a Remove).

3.4.2 Návrh aplikace pro zkoušku frameworku

Mimo testu v reálném prostředí bude vytvořena aplikace pro zkoušku funkčnosti frameworku. Do této aplikace bude úmyslně zavedena chyba, kterou by měl navrhovaný framework odhalit.

Kalkulačka

Aplikací pro zkoušku frameworku bude jednoduchá kalkulačka. Ta bude umět základní matematické operace (+ , - , * , /) mezi dvěma čísly. Kalkulačka bude z počátku počítat správně všechny operace. V případě provedení operace *, však přestane pracovat správně operace +, až dokud nebude provedena operace - či /. Taková chyba umožní dobré zhodnocení, zda framework dokáže najít chybu a správně ji genetickým algoritmem optimalizovat.

Obrázek 3.4: Návrh kalkulačky

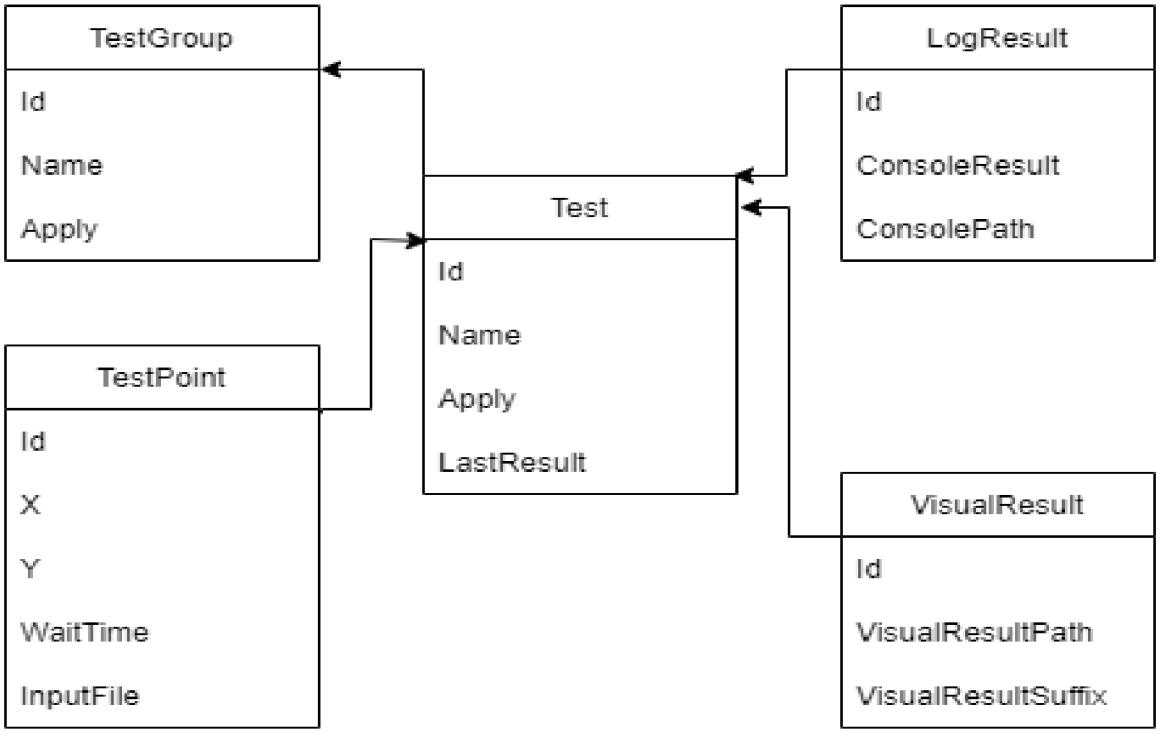
Kapitola 4

Implementace frameworku

Tato část práce bude zaměřena na samotnou implementaci frameworku. Nejdříve dojde k popisu implementovaných vrstev od nejnižších k nejvyšším a následně na nich bude ukázáno, jak implementovaný framework vypadá.

4.1 Databáze

Nejnižší vrstvou frameworku je databáze. Ta byla podle návrhu implementována v Entity Frameworku. Oproti návrhu se databáze mírně liší. Kvůli lepší přehlednosti a čitelnosti kódu bylo nutné tabulku CorrectResult rozdělit na tabulky Visual Result a Console Result. Pro jednotlivé body testu (ClickPoints - body na obrazovce, na které se při provádění daného testu kliká) byla taktéž vytvořena samostatná tabulka. Její součástí jsou například souřadnice daného bodu, čas který má framework čekat před daným kliknutím, či cesta k souboru, který má framework po kliknutí vložit. Vysledná databáze tedy obsahuje pět tabulek (viz diagram).



4.2 Repozitáře

Druhou nejnižší vrstvou jsou repozitáře. Ty jsou důležité hlavně pro odstínění databázové vrstvy, které je nutné kvůli vysoké kvalitě kódu. Každý repozitář obsluhuje jednu tabulku v databázi. S danými tabulkami tudíž nepracuje nikdo jiný, než k tomu určený repozitář. V implementaci navrhovaného frameworku byl z důvodu vysoké rozšiřitelnosti vytvořen jeden generický repozitář. Ten obsahuje logiku základních operací nad databází, které jsou nutné pro každou tabulku (vytvoření záznamu, smazání záznamu, aktualizaci záznamu a získání záznamu podle jeho id). Generický repozitář následně dědí repozitáře jednotlivých tabulek, přes které se následně pracuje s databází. Z důvodu ještě vyšší kvality kódu nejsou repozitáře používány přímo, ale všechny mají vlastní rozhraní (interface).

4.3 Modely

Modely reprezentují data frameworku, u těch jednodušších (v případě navrženého frameworku např. data skupin) je možné pracovat přímo s entitami ukládanými do databáze, u těch složitějších, které jsou obvykle uložené v několika tabulkách, to vhodné není a pro tyto data jsou tu modely. Navržený framework obsahuje čtyři hlavní modely. Prvním je TestModel. Ten obsahuje modely výsledků testu (ResultModel), Clickpointy testu (body na obrazovce na které se při testu kliká), výsledek testu (úspěch, neúspěch) a skupinu, do které test patří. Dalším modelem je ResultModel, obsahující pouze dvě vlastnosti, z nichž je použita vždy pouze jedna. V první vlastnosti je uložen vizuální model výsledku (VisualResultModel) a v druhé model výsledku logu (LogResultModel). Poslední dva modely jsou již dříve zmíněné výsledky (vizuální, či výsledek logu). Vizuální model obsahuje cestu k souboru s výsledkem a správný výsledek v datovém typu Bitmap. Výsledek logu obsahuje správný výsledek logu a cestu k logovacímu souboru.

4.4 Servisy

Jednou z nejdůležitějších částí celého frameworku jsou servisy. Servisy obsahují nejdůležitější logiku celého frameworku jako např. zaznamenávání klávesnice a myši, simulaci pohybu a klikání myši nebo vyhodnocování výsledků daných testů. Nyní bude následovat popis těch nejdůležitějších.

4.4.1 Bootstrapper

Bootstrapper je prvním a zdánlivě nedůležitým servisem. Bootstrapper není nutný pro funkčnost frameworku, nicméně je nezbytný pro vysokou kvalitu kódu a jeho rozšiřitelnost. Bootstrapper umožňuje tzv. Vkládání závislostí (Dependency injection), což je v objektově orientovaném programování způsob propojování jednotlivých komponent programu bez nutnosti vložení referencí před spuštěním programu. Pro Vkládání závislostí byl v navrhovaném frameworku zvolen Nuget balíček Autofac, z důvodu jeho vysoké kvality. Bootstrapper obsahuje jednu veřejnou statickou metodu, která pomocí privátních metod zaregistruje všechny potřebné reference podle předem určeného nastavení.

4.4.2 DatabaseService

DatabaseServis byl vytvořen proto, že bez něj by bylo nutné do každého view-modelu importovat repozitáře pro všechny tabulky, se kterými daný view-model pracuje. To by v některých případech mohlo znamenat zbytečné zhoršení čitelnosti kódu. Implementace tohoto servisu umožňuje importovat do každého view-modelu pouze jeden servis, který umožňuje pracovat se všemi potřebnými repozitáři (tabulkami). Funkčnost tohoto servisu je stěžejní pro práci s databází, proto budou nyní popsány jednotlivé metody.

LoadTests()

Metoda LoadTests() je volána hned při spuštění frameworku (po importu referencí Bootstrapperem). Metoda nejdříve vytáhne z databáze všechny testy, následně všechny ClickPointy a výsledky které k nim patří a vytvoří modely (TestModel). Z těchto modelů následně vytvoří kolekci List a ten vrátí.

GetTest(int id)

Metoda GetTest(int id) je volána vždy, když framework potřebuje vytáhnout z databáze určitý test, podle jeho id. Metoda funguje obdobně, jako metoda LoadTests(), s tím rozdílem, že místo kolekce testů z databáze tahá pouze jeden určitý test a jeho Clickpointy a výsledky. Následně opět vytvoří model, který vrátí volajícímu.

SaveTest(TestModel test)

Metoda SaveTest(TestModel test) je volána po vytvoření nového testu. Metoda ukládá nový test do databáze. Nejdříve vytvoří novou entitu (TestEntity), kterou uloží, následně vytvoří entity všech Clickpointů a uloží je do databáze k novému testu. Posledním krokem je vytvoření entit pro výsledky, ty jsou následně taktéž uloženy do databáze a přiřazeny k novému testu.

EditTest(TestModel test)

Metoda EditTest(TestModel test) je volána vždy, když framework potřebuje upravit výsledek určitého testu. Metoda nejdříve vytáhne z databáze entitu daného testu podle modelu a pokud je poslední výsledek v databázi rozdílný od aktuálního, uloží do databáze výsledek aktuální.

DeleteTest(int id), LoadTestGroups(), SaveGroup(TestGroupEntity testGroup), EditGroup(TestGroupEntity testGroup), DeleteGroup(TestGroupEntity testGroup)

Tyto metody dělají přesně to, co je zřejmé z jejich názvu. Metoda DeleteTest(int id) maže z databáze test s daným id. Metoda LoadTestGroups() vytáhne z databáze všechny skupiny a vrací kolekci List s těmito skupinami. Metoda SaveGroup(TestGroupEntity testGroup) ukládá nově vytvořenou skupinu do databáze. Metoda EditGroup(TestGroupEntity testGroup) upravuje skupinu v databázi a metoda DeleteGroup(TestGroupEntity testGroup) danou skupinu z databáze maže.

4.4.3 FileManager

Servis FileManager umožňuje frameworku pracovat se soubory. Obsahuje veřejné metody pro vytvoření snímku obrazovky, uložení snímku do souboru na disk, získání přesné velikosti oblasti v pixelech a čtení logu ze souboru. Servis využívají komponenty, které pracují s výsledky testů. Těmi jsou hlavně ResultService, který vyhodnocuje výsledky testů a viewmodely sloužící k ukládání a konfiguraci správných výsledků.

4.4.4 HookManager

Pro nahrávání testů je nezbytné odchytávat kliknutí, pohyby myši a kliknutí na klávesnici. To se může zdát jako jednoduchá a přímočará věc a ve své podstatě je, nicméně problém nastává, když uživatel začne pracovat s aplikací, se kterou framework nemá nic společného (každá testovaná aplikace). Framework potom nedokáže odchytávat kliknutí, protože je odchytává jiná aplikace. Způsob jak tuto problematiku vyřešit je vytvořit takzvaný GlobalHook. Ten odchytává události Windows API, takže zvládne zachytit kliknutí do jakékoli aplikace či kamkoli do Windows. Jako základ pro tuto funkcionalitu byl použit kód ze článku Processing Global Mouse and Keyboard Hooks in C# [19] od George Mamaladze, který byl následně upraven. Výsledek umožňuje frameworku zachytávat kliknutí, pohyb myši či kliknutí na klávesnici kdykoli je potřeba, přestože framework běží pouze v pozadí.

4.4.5 RecordService

Díky HookManageru framework dokáže odchytávat pohyby a kliknutí myši, i kliknutí na klávesnici. Dále je nutné tuto aktivitu uživatele zaznamenat a uložit. K tomu slouží RecordService. Ten z uživatelského pohledu obsahuje hlavně metody pro spuštění, zastavení či pozastavení nahrávání. Spuštění nahrávání funguje v několika bodech. Prvním je připojení vlastních metod na HookManager. Tyto metody jsou následně vyvolány při pohybu myši, či kliknutí na myš nebo klávesnici. Pro co nejnižší zátěž se nezaznamenává pohyb myši, ale pouze kliknutí, při testování se následně vypočítá nejkratší cesta myši k danému bodu. Při každém kliknutí myši se vytvoří nový ClickPoint a přidá se danému testu. Při kliknutí na klávesnici se zjišťuje, zda uživatel kliknul na nějaký pro framework klíčový znak (například Enter pro ukončení nahrávání, či F pro otevření FileManageru), pokud ano - správně reaguje, pokud ne - nestane se nic. Po stisknutí klávesy Enter se zastaví nahrávání (odpojení metod od HookManageru), uloží se obraz aktuálního stavu obrazovky a framework se přepne do výsledkového okna.

4.4.6 NativeMethods

NativeMethods je servis umožňující používání Windows API. To frameworku umožňuje programově nastavovat pozici myši, získat aktuální pozici myši, či na myš kliknout bez ohledu na její pozici (libovolný monitor, libovolná část obrazovky, libovolná aplikace). Další funkce, které servis NativeMethods poskytuje, dovolují programově používat klávesnici (libovolně na ní psát, či používat klávesové zkratky).

4.4.7 MouseSimulationService

MouseSimulationService využívá možností NativeMethods a pohybuje myší po obrazovce v průběhu testování. Obsahuje vlastnost Stop, která slouží k nouzovému zastavení testování uživatelem a metodu MoveCursorWithPath. Tato metoda v parametrech přijímá počáteční

bod, koncový bod a počet kroků. Metoda obsahuje cyklus, který postupně pohybuje myší (voláním NativeMethods) po nejkratší možné cestě až do koncového bodu.

4.4.8 TestService

Jedním z nejdůležitější servisů frameworku je TestService. Jeho součástí jsou metody pro přidání, či odebrání testů, které uživatel v danou chvíli potřebuje spustit. Nejpodstatnější metodou servisu je metoda ExecuteTests. Tato metoda postupně prochází všechny spuštěné testy. V každém testu hýbe myší pomocí MouseSimulationService, kliká na všechny potřebné body pomocí NativeMethods, pokud daný bod obsahuje soubor, vloží ho na místo kliknutí, případně před kliknutím počká předem stanovený čas. Nakonec pomocí Servisu ResultService porovná všechny výsledky a do databáze uloží výsledek testu.

4.4.9 ResultService

Posledním servisem frameworku je již dříve zmíněný ResultService. Ten slouží k porovnání výsledků testu se správnými výsledky z databáze. Servis umožňuje, jak vyhledávání správného výsledku v logu aplikace po testu, tak i porovnání obrazů pro kontrolu vizuálních výsledků. Vyhledávání správného výsledku v logu funguje na principu regulárních výrazů. Prvním krokem je načtení logu testu. Druhým krokem pak vyhledání regulárního výrazu z databáze v logu (pomocí třídy Regex a její metody IsMatch). Posledním krokem je navrácení výsledku (true, pokud byl regulární výraz v logu nalezen, false pokud ne). Pro porovnání vizuálních obrazů slouží metody, které pracují s EmguCV. Pro framework bylo testováno několik metod porovnávání vizuálních obrazů. Nejvyšší přesnosti dosahovala metoda „Template Matching“, proto byla pro framework zvolena právě tato metoda. Níže je vyobrazeno její použití.

```
public double GetVisualPercentageResult( Bitmap source, Bitmap template )
{
        try
        {
                var src = source.ToImage<Bgr, byte>();
                var tmp = template.ToImage<Bgr, byte>();

                var vp = ProcessImage( tmp.Convert<Gray, byte>(),
                                       src.Convert<Gray, byte>() );
                if ( vp != null )
                {
                        Rectangle bbox = CvInvoke.BoundingRectangle( vp );
                        src.ROI = bbox;

                        var img = src.Copy();
                        src.ROI = Rectangle.Empty;

                        Image<Gray, float> resultImage = img.MatchTemplate(
                                        tmp,TemplateMatchingType.SqdiffNormed );

                        double[] minValues, maxValues;
                        Point[] minLocations, maxLocations;
```

```
                    resultImage.MinMax( out minValues, out maxValues,
                                        out minLocations, out maxLocations );

            //this will be percentage of difference of two images
                    var result = 100 - ( maxValues[0] * 100 );
                    return result;
                }
                return 0;
        }
        catch ( Exception ex )
        {
                Console.WriteLine( ex );
        }

        return 0;
}
```

4.5 Optimalizace

Podle návrhu je po nalezení chyby v testu provedena optimalizace, která se pokusí najít nejnižší počet testů, které spolu nefungují a vrátit je ve správném pořadí. K provádění dané optimalizace se využívá genetický algoritmus. Hlavními částmi algoritmu jsou třídy: GeneticHandler, Chromosome a Fitness.

4.5.1 GeneticHandler

GeneticHandler je třída, která slouží k řízení genetického algoritmu. Třída obsahuje jedinou metodu, tou je metoda Go(), která v parametrech bere list s testy pro optimalizaci a poslední test, který je po provedení ostatních testů nefunkční. Metoda nejdříve vytvoří instance tříd, které bude genetický algoritmus používat. Těmi jsou instance tříd: EliteSelection() pro selekci, OrderedCrossover pro křížení, ReverseSequenceMutation pro mutaci, Fitness pro fitness funkci, Chromosome pro chromozom a Population pro populace. Následně jsou tyto třídy předány nové instanci genetického algoritmu (GeneticAlgorithm) a algoritmus se spouští. Algoritmus je nastaven na 100 až 120 generací, protože bylo experimentálně zjištěno, že dané rozmezí produkuje dobré výsledky v ne příliš dlouhém časovém okně. Nakonec se algoritmus provede a vrací nejlépe hodnocený chromozom. Metoda vrací list testů z nejlepšího chromozomu.

4.5.2 Chromosome

Chromosome je třída zastupující chromozom genetického algoritmu. Chromozom daného algoritmu obsahuje generátor náhodných čísel, dva listy s testy a poslední test. První list (InitTests) slouží pouze pro správné ohodnocení genů ve fitness funkci a obsahuje testy, které jsou chromozomu předány při inicializaci. V druhém listu (GeneTests) jsou uloženy testy, které se postupně přibližují výsledku. Pro jednoduchost je v chromozomu uložen i poslední test, který by měl během optimalizace zůstat nefunkční (když nezůstane, ohodnocení daného chromozomu je 0). Součástí chromozomu jsou 2 metody, metoda GenerateGene() pro

vytvoření nového genu a metoda CreateNew() pro vytvoření nového chromozomu. Při vytváření nového genu je určitá pravděpodobnost (zajištěna náhodným generátorem čísel) odstranění náhodného testu. To zajistí lepší ohodnocení v případě, že poslední test zůstane nefunkčí i po této operaci. Mezi danými geny následně probíhá mutace i křížení, které zajišťuje knihovna GeneticSharp.

4.5.3 Fitness

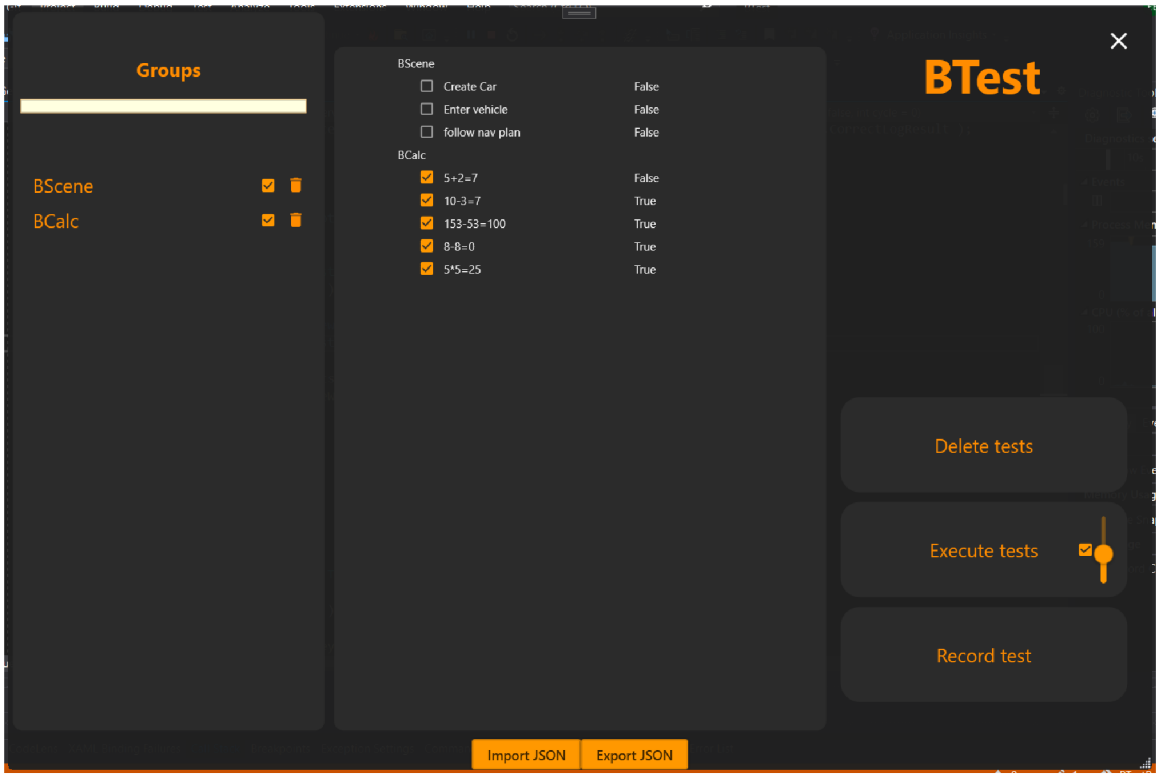
Třída Fitness obsahuje fitness funkci genetického algoritmu, která slouží k hodnocení daných genů algoritmu. Hlavní metodou třídy je metoda Evaluate, která vrací ohodnocení daného chromozomu. Největším problémem algoritmu v implementovaném frameworku byla doba trvání optimalizace. To bylo způsobeno tím, že pro správné ohodnocení chromozomu bylo nutné provést všechny testy v listu GeneTests a následně provést poslední test pro zjištění, jestli je stále nefuknční. Metoda evaluate je při běhu algoritmu volána asi osm tisíc krát, což způsobilo nesmyslně dlouhou dobu optimalizace. Řešením tohoto problému je provádění testů chromozu pouze, pokud je jeho možné ohodnocení lepší než ohodnocení nejlepšího dřívějšího chromozomu a pokud jeho hodnocení již není nejlepší možné. Ohodnocení se počítá odečtením testů, které v chromozomu zbyly (v Listu GeneTests), ale jen v případě, že po jejich provedení je poslední test stále nefunkční. V opačném případě je ohodnocení chromozomu nula.

4.6 Pohledy

V této sekci budou popsány nejdůležitější pohledy aplikace a jejich možnosti. Ke každému pohledu je přiřazen také view-model, těmi se však práce nebude zaobírat, protože ty obvykle slouží čistě k obsluze daného pohledu.

4.6.1 MainView

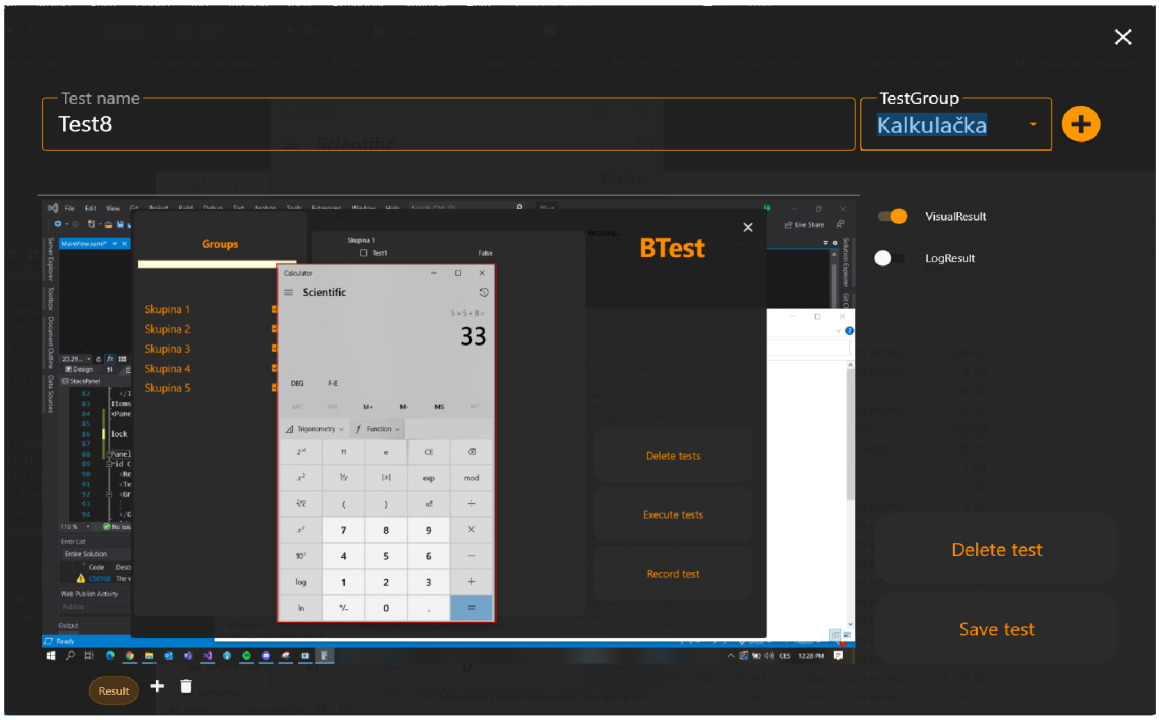
Prvním pohledem je pohled MainView. Ten se zobrazí okamžitě po spuštění frameworku a obsahuje základní rozcestník pro uživatele. Pohled zobrazuje dříve vytvořené skupiny testů a testy samotné. Uživatel si může vybrat, které skupiny chce zobrazit a z daných skupin vybrat testy, které bude chtít spustit, popřípadě vymazat. V levém horním rohu framework při testování názorně vyobrazuje kolik testů již proběhlo a v pravém dolním rohu má uživatel možnost vybrat, zda chce spustit vybrané testy, smazat je, čí nahrát nový. Vedle tlačítka pro spuštění testů leží CheckBox, který zapíná či vypíná optimalizaci genetickým algoritmem a hned vedle je Slider nastavující kolikrát se dané testy spustí (první spuštění je pořadí testů dané, před každým dalším se pořadí testů promíchá). Poslední možností na hlavním pohledu jsou tlačítka na spodní straně pohledu, které umožňují vyexportovat vybrané testy s jejich skupinami do souboru formátu JSON, či daný soubor importovat.



Obrázek 4.1: Hlavní okno

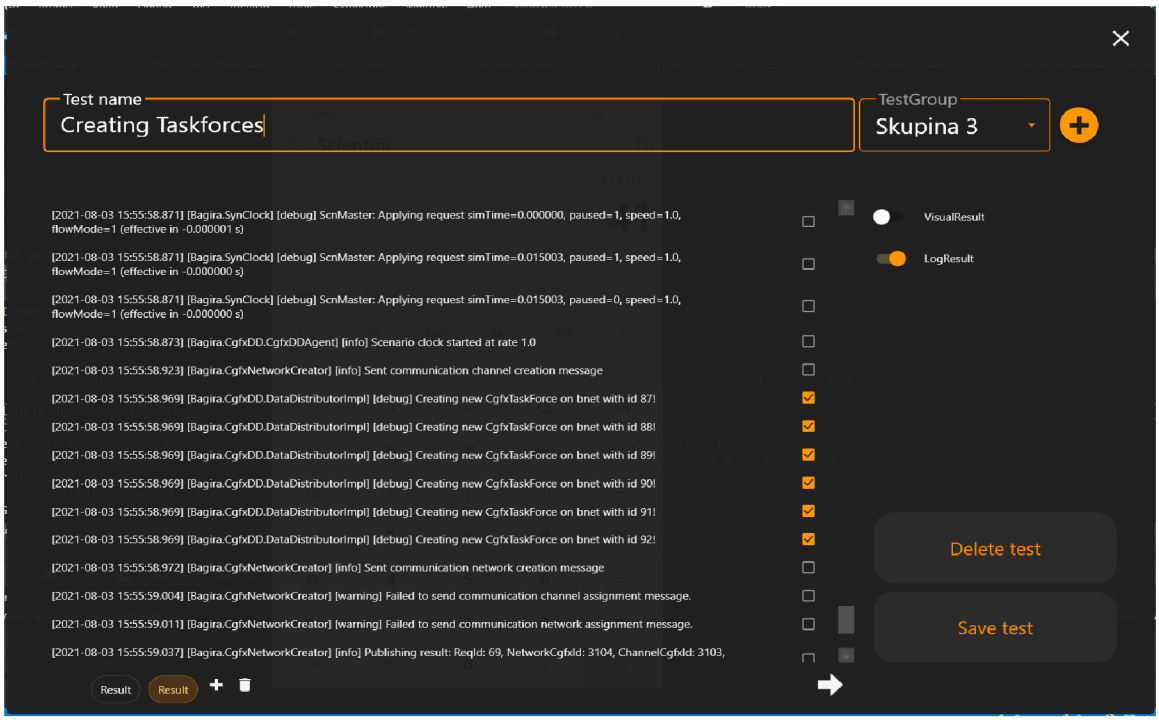
4.6.2 ResultView

Po stisknutí tlačíka Record test začne framework zaznamenávat akce uživatele a hlavní okno frameworku se skryje. Po naklikání testu uživatelem framework přejde do dalšího pohledu, který se zabývá přidáním důležitých informací k danému testu. V horní části pohledu je pole s názvem daného testu (ten zadá uživatel), hned vedle je pole pro zadání testové skupiny, které je možné přidávat či upravovat po stisknutí tlačítka +, ležícím za ním. Největší část obrazovky zabírá výsledek, který je buď vizuální nebo výsledek logu, což mění dva ToggleButtony v pravé části pohledu. Výsledky lze také přidávat či mazat pomocí tlačítek ve spodní liště. Poslední součástí pohledu jsou tlačítka Delete test, které test zahodí a tlačítko Save test, které ho uloží. Obě tlačítka po vykonání této akce vrátí okno do hlavního pohledu.



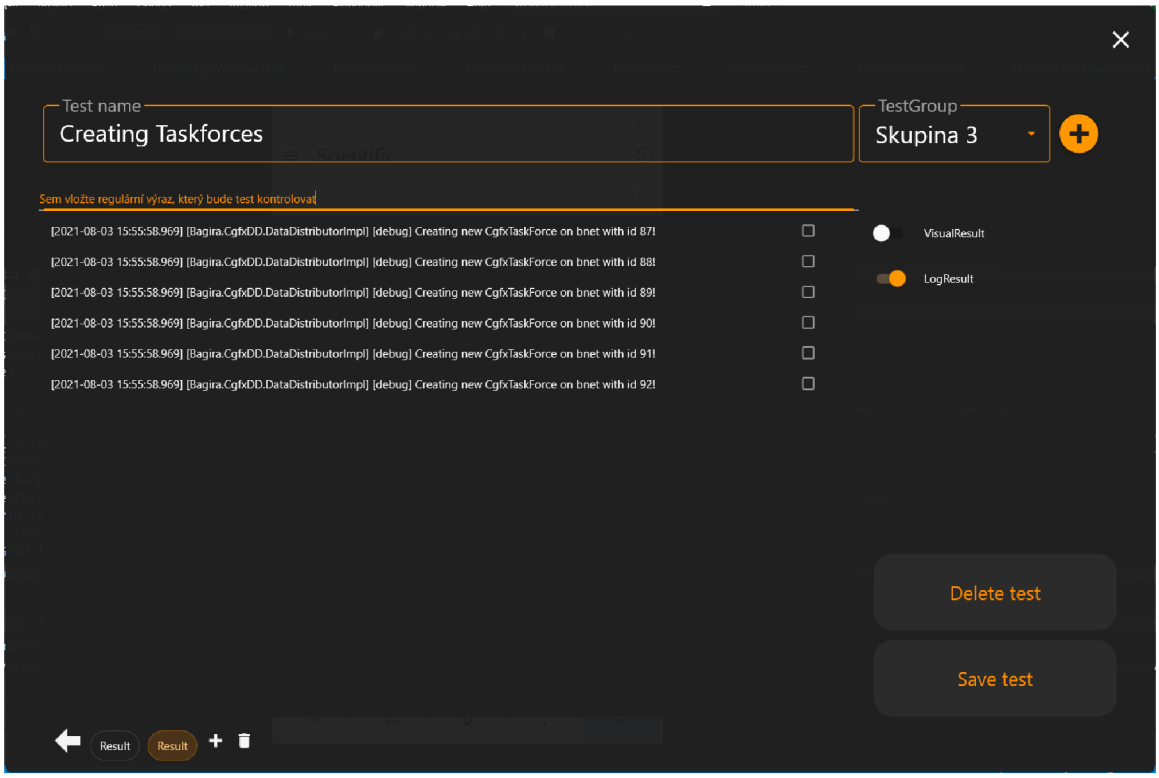
Obrázek 4.2: Výsledkové okno - první pohled

Na prvním obrázku (4.2) je možné vidět výsledkový pohled, test s názvem Test8, patřící do skupiny Kalkulačka, s vizuálním výsledkem. V tomto případě bude framework po provedení testu hledat na obrazovce okno kalkulačky (část ve výsledku ohraničenou červeně).



Obrázek 4.3: Výsledkové okno - druhý pohled

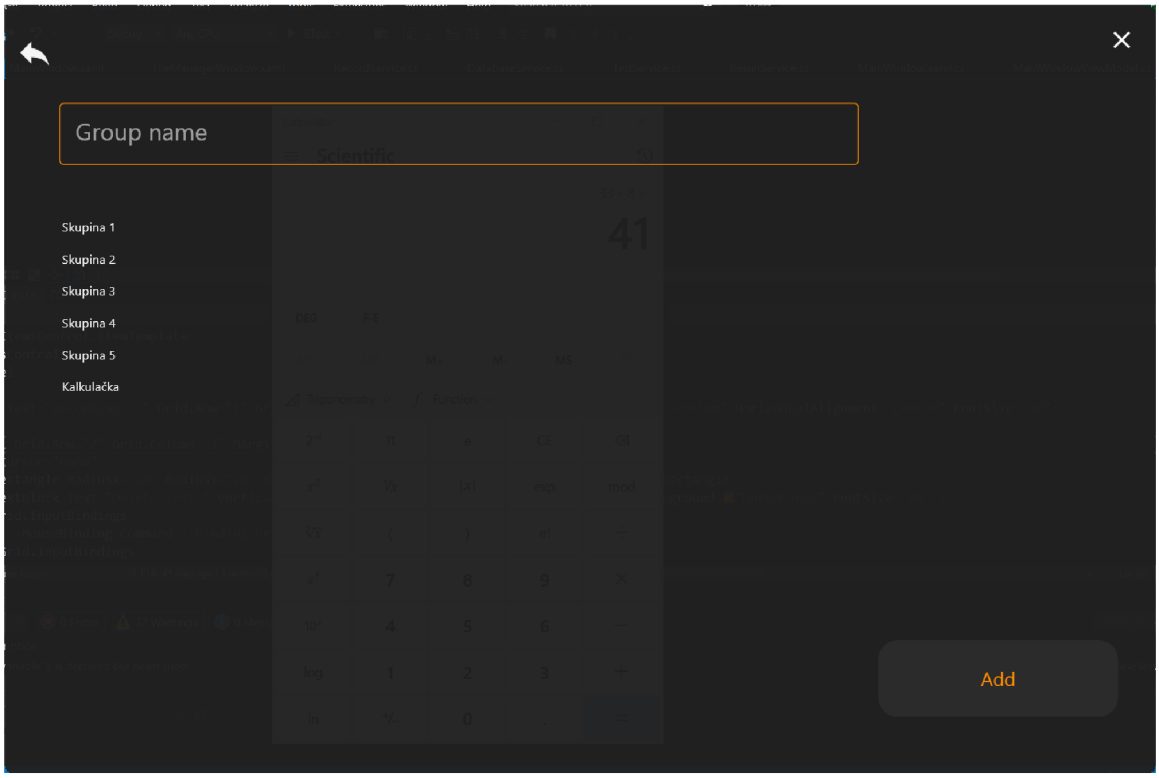
Obrázek 4.3 dále zachycuje druhou možnost výsledku. Vlevo dole je možnost vybrat počet výsledků, které se budou kontrolovat (v tomto případě 2). Narozdíl od předchozího obrázku je zde vyobrazen výsledek logu aplikace. Log nahrál uživatel, vybral z něj výpisy, které chce kontrolovat a pokračuje stisknutím šipky vpravo. Další postup popisuje obrázek 4.4.



Obrázek 4.4: Výsledkové okno - třetí pohled

Obrázek 4.4 vyobrazuje aktuální stav frameworku při stisknutí dříve zmíněné šipky. Zde jsou vyobrazeny vybrané výpisy logu a nad ně přibyl řádek, do kterého uživatel zadá regulární výraz, kterým bude kontrolovat výpisy pod ním. Dále už nezbývá nic než test uložit a vrátit se na hlavní obrazovku.

4.6.3 GroupsView

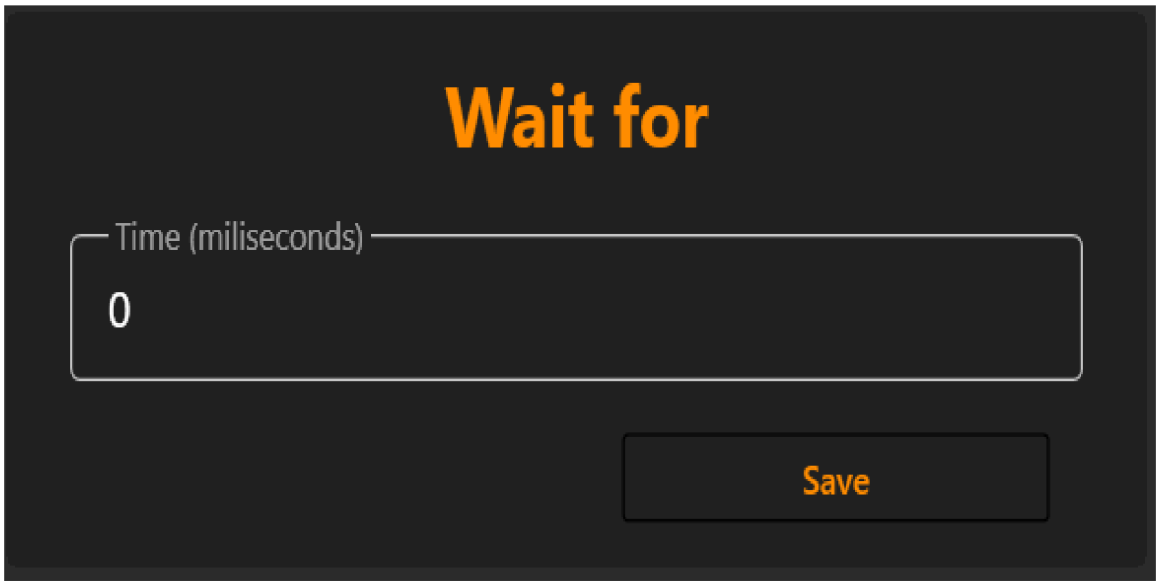


Obrázek 4.5: GroupsView

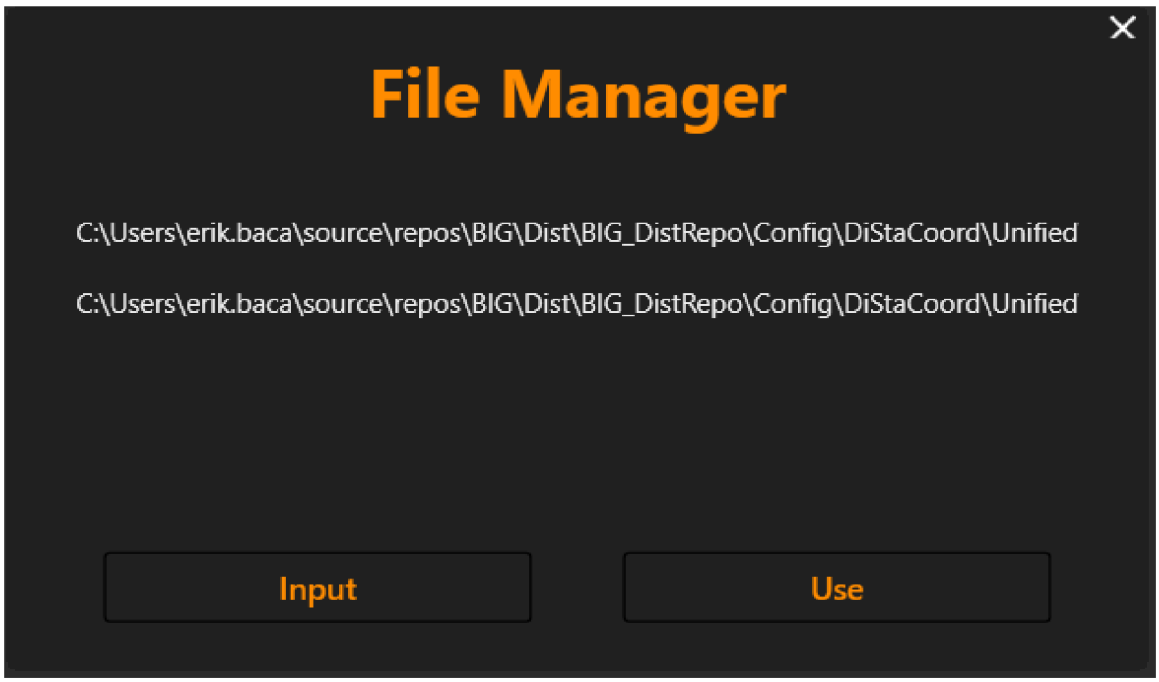
Posledním pohledem je pohled GroupsView, který řeší přidávání, úpravu, či mazání testových skupin. Hned nahoře je kontrolka pro zadávání textu, když do ní napíšeme nové jméno a stiskneme tlačítko Add, vytvoříme novou skupinu. V případě kliknutí na již existující skupinu, přibude tlačítko Delete, kterým se skupina maže a tlačítko Add se změní na tlačítko Edit. Nyní uživatel může přepsat jméno skupiny a kliknutím na tlačítko Edit upraví jméno skupiny v databázi.

4.7 Okna

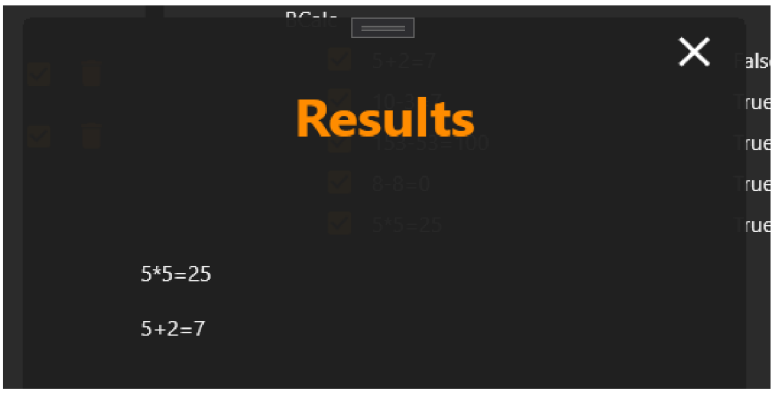
Poslední částí frameworku, ještě nebyla zmíněna jsou okna. Framework obsahuje čtyři okna. Prvním je hlavní okno frameworku, ve kterém se zobrazují jednotlivé pohledy. Dalším oknem je WaitTimeWindow (obrázek 4.6), které umožňuje uživateli určit čas, který má framework čekat před určitým kliknutím. Následujícím oknem frameworku je okno FileManagerWindow (obrázek 4.7), které se stará o import souborů potřebných pro daný test a posledním oknem je GeneticResultsWindow (obrázek 4.8), které vypisuje výsledky genetického algoritmu.



Obrázek 4.6: WaitTimeWindow



Obrázek 4.7: FileManagerWindow



Obrázek 4.8: GeneticResultsWindow

4.8 Aplikace pro zkoušku frameworku

Kalkulačka

Kalkulačka nazvaná podle frameworku BTestCalc je přesnou implementací podle návrhu. Stejně jako framework byla napsána ve WPF. Kalkulačka zvládá jednoduché operace mezi čísly a byla do ní zavedena chyba, přesně jak je popsáno v návrhu.

Obrázek 4.9: Kalkulačka

Kapitola 5

Zhodnocení výsledků

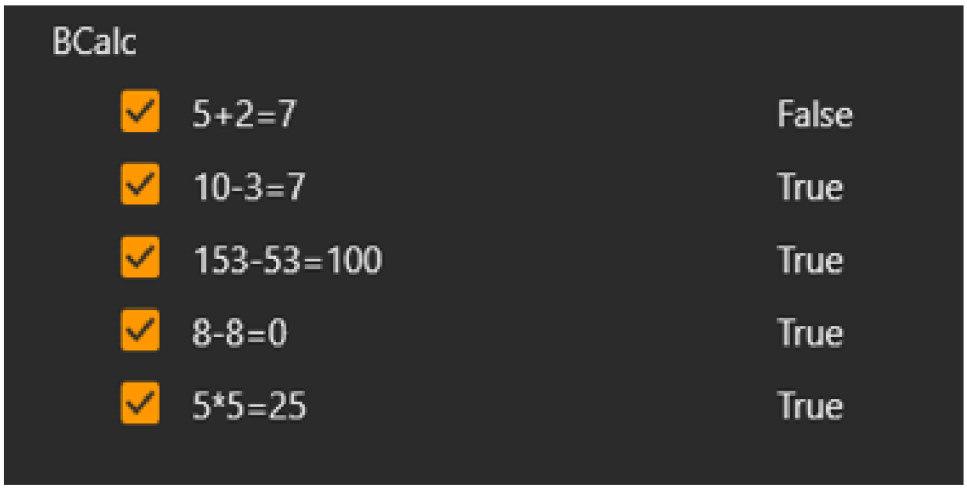
5.1 Testy aplikace vytvořené pro zkoušku frameworku

BTestCalc

Pro testování implementované kalkulačky bylo vytvořeno několik sad testů. Každá z těchto sad testů odhalila chybu, kterou následně framework správně optimalizoval. Jediné co se lišilo, byl průměrný čas hledání chyb a optimalizace. Každá sada byla koncipována tak, že testy samostatně fungovaly správně, nicméně ne všechny fungovaly společně.

Z důvodu, že všechny sady měly velice podobné výsledky a lišily se pouze v času běhu, pro zhodnocení postačí detailněji popsat jen dvě sady testů. Čas celkového běhu testů není vhodné měřit, protože je nalezení hluboké chyby závislé na náhodě. Pořadí testů v jednotlivých cyklech je náhodné, což způsobuje, že se chyba může objevit například již ve druhém cyklu a celkový běh bude v této situaci výrazně kratší, než když se chyba objeví až v běhu dvacátem. To však nelze kvůli náhodě přesně simulovat. Co naopak ale měřit lze je rychlost, za kterou framework dokáže chybu optimalizovat.

První popisovaná sada obsahovala pět testů. Každý test z dané sady zkoušel jednu početní operaci. Testy zkoušely postupně všechny početní operace a z jejich názvu (viz obrázek 5.1) je patrné jaké příklady pro to byly použity. Z návrhu kalkulačky je zřejmé, že v její implementaci je zavedena chyba. Ta se ve zmíněné sadě projeví v případě, že framework provede test s příkladem $5 * 5 = 25$ těsně před testem s příkladem $5 + 2 = 7$. Při nastavení frameworku na více než deset cyklů průchodu dané sady, framework ve velké většině případů chybu našel (chybu nenašel jen v případech, kdy byl počet cyklů v nízkých jednotkách, například dva cykly a dva dříve zmíněné testy se během testování nikdy neprovedly po sobě). Po nalezení chyby následovala optimalizace, jenž fungovala velmi dobře, z deseti měřených optimalizací našly všechny správný výsledek a každá trvala průměrně 5,5 minut (přesné časy jsou vyobrazeny v tabulce 5.1).

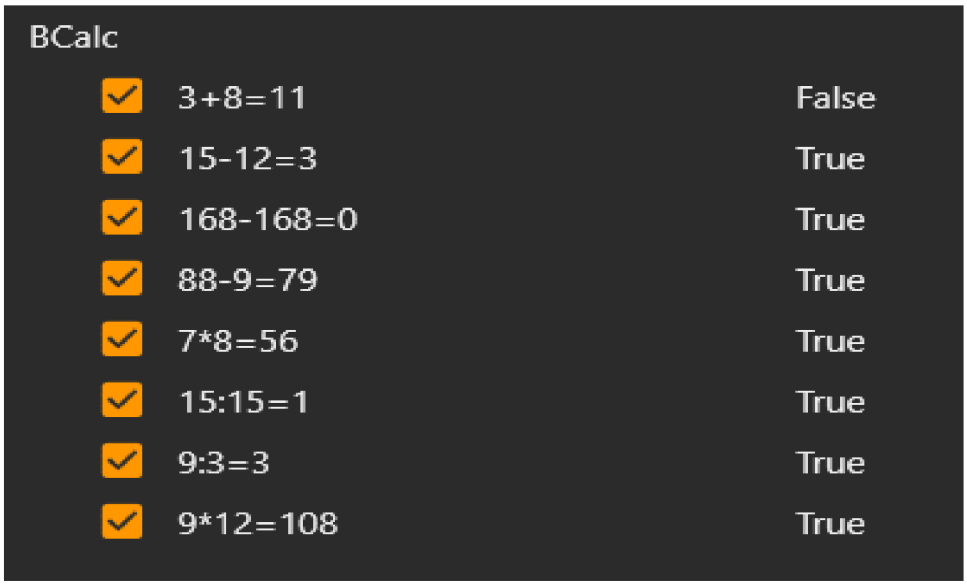


Obrázek 5.1: 1. Sada testů

Číslo běhu	Čas (počet minut)	Počet cyklů	Původní počet testů	Konečný počet testů	Nalezen nejlepší výsledek
1.	4,5	10	5	2	Ano
2.	5.8	10	5	2	Ano
3.	6.5	10	5	2	Ano
4.	0.5	10	5	2	Ano
5.	5.3	10	5	2	Ano
6.	2.5	10	5	2	Ano
7.	3.9	10	5	2	Ano
8.	12.1	10	5	2	Ano
9.	4.3	10	5	2	Ano
10.	6.7	10	5	2	Ano

Tabulka 5.1: Tabulka výsledků 1. sady testů

Druhá testovací sada byla založena na stejném principu jako ta první. Sady se lišily v tom, že každá používala zcela jiné testy. Součástí druhé sady bylo osm testů (viz obrázek 5.2). Z názvu každého z nich je opět jasně vidět, jakou početní operaci s jakými čísly testovala. Nehledě na odlišné testy optimalizace fungovala velmi dobře s jediným problémem, kterým je rychlost. Framework zvládá velmi dobře odhalit chyby, včetně těch skrytých, které se objevují jen zřídka, nicméně spolu s počtem testů výrazně roste i čas optimalizace. Přesné časy měřených pokusů jsou vyobrazeny v tabulce 5.2.



Obrázek 5.2: 2. Sada testů

Číslo běhu	Čas (počet minut)	Počet cyklů	Původní počet testů	Konečný počet testů	Nalezen nejlepší výsledek
1.	16	15	8	2	Ano
2.	8	15	8	2	Ano
3.	65	15	8	2	Ano
4.	30	15	8	2	Ano
5.	15	15	8	2	Ano
6.	62	15	8	2	Ano
7.	40	15	8	2	Ano
8.	12	15	8	2	Ano
9.	7	15	8	2	Ano
10.	39	15	8	2	Ano

Tabulka 5.2: Tabulka výsledků 2. sady testů

Z výsledků je zřejmé, že čím více testů framework prováděl, tím déle mu trvalo najít chybu a optimalizovat ji. Nicméně se jedná o automatické testy, které nikdo nemusí sledovat a vzhledem k velmi dobrým výsledkům je přínos frameworku neoddiskutovatelný.

5.2 Testy na reálných systémech

Framework byl testován v reálném prostřední na dvou systémech armádního simulátoru firmy Bagira Systems. Framework byl také zkoušen testerem v již zmíněné firmě. To přineslo objektivní pohled na to, jak framework ve skutečnosti funguje a jak je nebo není využitelný v praxi. Výsledkem těchto testů je zjištění, že framework je dobře použitelný, nicméně není

dokonalý a během testování bylo nalezeno dost možných zlepšení, ať už ohledně funkčnosti, či použití lidmi, kteří nejsou obeznámeni s tím, jak funguje. Hlavním problémem, zjištěným až na reálném systému, je problém s hodnocením provedených testů. Vizuální výsledky nemají, z důvodu vysoké složitosti hledaných obrazců, dokonalé výsledky a ačkoli výsledky logu fungují velmi dobře, vzhledem k tomu, že framework pouze hledá daný regulární výraz v určitém logu, nastává problém, když log obsahuje daný řádek z předchozích testů. To je možné vyřešit hledáním pouze v části logu pomocí časové značky, což ale framework zatím nedělá. Testování tudíž odhalilo nedokonalosti, které by bylo dobré odstranit, hlavním zjištěním však je, že se framework dá dobře použít i na reálném systému, nejen v simulovaném prostředí.

5.3 Prostor pro zlepšení

Framework je možné velmi dobře rozšířit a i přes dobré výsledky je možností, jak ho posunout na další úroveň, stále dost. Hlavními z nich mohou být například zlepšení vyhodnocování výsledků, jak vizuálních (úpravou algoritmů), tak výsledků logu (přidáním časové značky), zlepšení uživatelské přívětivosti frameworku (framework dost unavuje oči a pro člověka, který ho vidí poprvé, není úplně snadné ho používat), přidání možnosti používat dvojklik či zapisovat text na klávesnici v průběhu nahrávání testu.

Kapitola 6

Závěr

Cílem bakalářské práce bylo navrhnout a implementovat framework pro testování grafického uživatelského rozhraní, který bude využívat algoritmů Soft-Computingu. Úkolem bylo podívat se na problém netradičním způsobem a přinést možnosti, které existující software nenabízí. Tento vytyčený cíl byl splněn.

Během řešení práce se bylo nutné nejdříve seznámit s již existujícím softwarem pro testování GUI. Následně bylo nezbytné zaměřit se na zjištění, jaké jsou jeho nevýhody, jaké možnosti nenabízí, a jak by bylo možné ho vylepšit. Cílem softwaru pro testování GUI je nahradit práci člověka tam, kde to jde, tím výrazně zvýšit rychlost testování a umožnit člověku pracovat na něčem jiném. Velmi důležité bylo, aby tohle framework umožňoval.

Dalším krokem bylo na základě získaných znalostí navrhnout framework. V úvodních fázích návrhu se bakalářská práce zaměřuje na vytyčení cílů, které by měl framework splnit a výběr vhodných technologií. Pro implementaci byl zvolen programovací jazyk C#. Následoval výběr vhodných Soft-Computing algoritmů, rozebíraných v teoretické části této práce. Poslední částí této sekce bylo navrhnout nejdůležitější části frameworku spolu s návrhem aplikace, na které byl framework testován.

Po návrhu přišla na řadu implementace a zhodnocení výsledků. Implementace, až na drobné úpravy, blíže popsané v implementační části práce, odpovídá návrhu. Při hodnocení výsledků byl vytvořený framework testován nejdříve na simulovaném prostředí. To využívalo aplikaci (vytvořenou v rámci práce), která obsahovala chybu, výsledky a měření jsou popsány v části se zhodnocením. Následně byl framework otestován na reálných systémech firmy Bagira Systems, a ačkoli toto testování odhalilo nedostatky frameworku, bylo zjištěno, že je framework použitelný nejen v simulovaném prostředí, ale i v praxi na reálných systémech.

Jak již bylo zmíněno v předchozích odstavcích tak framework funguje dobře i při použití na reálných systémech, Nicméně možností, jak ho posunout na vyšší stupeň kvality, je stále dost. Hlavní z nich jsou například zlepšení vyhodnocování výsledků, jak vizuálních (úpravou algoritmů), tak výsledků logu (přidáním časové značky), či zlepšení uživatelské přívětivosti frameworku.

Literatura

[1] *Appium*. Dostupné z: https://github.com/appium/appium.

[2] *Get started with desktop Windows apps that use the Win32 API*. Written: 2021-07-01. Dostupné z: https://docs.microsoft.com/en-us/windows/win32/desktop-programming.

[3] *Rapise*. Dostupné z: https://www.inflectra.com/Rapise/?utm_source=GoogleAd&gclid=EAIaIQobChMIytSB1_7S9wIV1-F3Ch1qbgA9EAAYASAAEgIZyvD_BwE.

[4] *Sahi*. Dostupné z: https://resources.sahipro.com/docs/introduction/index.html.

[5] *The Selenium Browser Automation Project*. Last modified: 2021-12-11. Dostupné z: https://www.selenium.dev/documentation/.

[6] *Squish - Automated GUI Testing that works*. Dostupné z: https://www.froglogic.com/squish/.

[7] *Telerik Test Studio*. Dostupné z: https://www.telerik.com/teststudio.

[8] *TestComplete*. Dostupné z: https://smartbear.com/product/testcomplete/overview/.

[9] *TestStack.White*. Dostupné z: https://teststackwhite.readthedocs.io/en/latest/.

[10] Aleezaadeh, A. Indian Academicians and Researchers Association. In:. únor 2018.

[11] Bc. Štěpán Karásek. *NEURONOVÉ SÍTĚ A GENETICKÉ ALGORITMY*. Brno, CZ, 2016. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií, Ústav inteligentních systémů. Dostupné z: https://www.fit.vut.cz/study/thesis-file/7525/7525.pdf.

[12] Corporation, E. *Emgu CV*. Last edit: 12 January 2022. Dostupné z: https://www.emgu.com/wiki/index.php/Main_Page.

[13] Dernoncourt, F. Introduction to fuzzy logic. Leden 2013.

[14] Giacomelli, D. *GeneticSharp*. Dostupné z: https://github.com/giacomelli/GeneticSharp.

[15] Gurney, K. *An introduction to neural networks*. University of Sheffield, 1997.

[16] Kitner, R. *Typy testování software (třídění testů)*. Written: 2021-07-01. Dostupné z: https://kitner.cz/testovani_softwaru/typy-testovani-software-trideni-testu/.

[17] Kumari, B., Chauhan, N. a Syed, H. H. A COMPARISON BETWEEN MANUAL TESTING AND AUTOMATED TESTING. *SSRN Electronic Journal*. Prosinec 2018, sv. 5, s. 323–331.

[18] Lee, T. G. *Co je Windows Presentation Foundation (WPF)?* Written: 2021-12-03. Dostupné z: https://docs.microsoft.com/cs-cz/visualstudio/designers/getting-started-with-wpf?view=vs-2022.

[19] Mamaladze, G. *Processing Global Mouse and Keyboard Hooks in C.* Written: 31 Aug 2011. Dostupné z: https://www.codeproject.com/Articles/7294/Processing-Global-Mouse-and-Keyboard-Hooks-in-C.

[20] Panda, M. *Soft Computing: Concepts and Techniques.* Leden 2014. ISBN 978-93-81159-66-8.

[21] Qureshi, I. a Nadeem, A. GUI Testing Techniques: A Survey. *International Journal of Future Computer and Communication.* Leden 2013, s. 142–146. DOI: 10.7763/IJFCC.2013.V2.139.

[22] Robert C. Martin, P. W.-W. a contributors, F. *FitNesse User Guide.* Dostupné z: http://fitnesse.org/FitNesse.UserGuide#introduction.

[23] Steegmans, E., Bekaert, P., Devos, F., Delanote, G., Smeets, N. et al. Black and White Testing: Bridging Black Box Testing and White Box Testing. Leden 2004.

[24] Taylor Sakyi, K. Reliability Testing Strategy - Reliability in Software Engineering. Květen 2016.

[25] Thede, S. An introduction to genetic algorithms. *Journal of Computing Sciences in Colleges.* Říjen 2004, sv. 20.

[26] Young, D. Software Testing Overview. *White Paper.* Březen 2015.

[27] Zahradníčková, J. *Automatizované testování informačního systému s využitím testů řízených daty.* Brno, CZ, 2019. Diplomová práce. Masarykova univerzita, Fakulta informatiky. Dostupné z: https://is.muni.cz/th/bdwxr/DP_Zahradnickova.pdf.

[28] Zayegh, A. a Albassam, N. Neural Network Principles and Applications. In:. Listopad 2018. DOI: 10.5772/intechopen.80416. ISBN 978-1-78984-540-2.

..

Příloha A

Obsah CD

Adresář BTestCalc - Obsahuje soubory s testovanou aplikaci pro zhodnocení výsledků frameworku.

Podadresář BTestCalc/BTestCalc - Obsahuje soubory se zdrojovým kódem aplikace.

Podadresář BTestCalc/Binaries - Obsahuje spustitelné soubory aplikace.

Adresář BTest - Obsahuje soubory frameworku.

Podadresář Btest/Btest - Obsahuje soubory se zdrojovým kódem frameworku.

Podadresář Btest/Binaries - Obsahuje spustitelné soubory frameworku.

Adresář TextBP - Obsahuje zdrojové soubory textu bakalářské práce.

Soubor xbacae00-bp.pdf - Obsahuje vysázený text bakalářské práce.

Soubor README - Obsahuje informace o adresářové struktuře, obsahu přiloženého CD a rychlokurz ovládání frameworku.