VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

NÁSTROJ PRO USNADNĚNÍ TESTOVÁNÍ GUI WEBOVÝCH APLIKACÍ

PRODUCTIVITY TOOL FOR WEB APPLICATION TESTING

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. FILIP KALOUS**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. JAN PLUSKAL**
SUPERVISOR

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2018/2019

Zadání diplomové práce

Student: **Kalous Filip, Bc.**
Program: Informační technologie Obor: Informační systémy
Název: **Nástroj pro usnadnění testování GUI webových aplikací**
Productivity Tool for Web Application Testing
Kategorie: Web

Zadání:

1. Seznamte se s problematikou testování uživatelských rozhraní webových aplikací. Porovnejte stávající frameworky pro UI testování webových aplikací.
2. Definujte požadavky na nástroj, který automatizovaně vytvoří programový popis UI. Tento popis bude sloužit jako kostra pro psaní UI testů a zajištění statické kontroly v době překladu.
3. Proveďte návrh dle požadavků z bodu 2.
4. Implementujte v prostředí open-source frameworku DotVVM v jazyce C#.
5. Otestujte implementaci. Proveďte zhodnocení výsledku, demonstraci funkčnosti a popište další možnosti pro rozšíření.

Literatura:

1. Leotta, M., Clerissi, D., Ricca, F., & Spadaro, C. (2013, March). Improving test suites maintainability with the page object pattern: An industrial case study. In *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on* (pp. 108-113). IEEE.
2. Zhan, Z. (2015). Selenium WebDriver Recipes in C#. Apress.

Při obhajobě semestrální části projektu je požadováno:

- Body 1, 2 a 3.

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Pluskal Jan, Ing.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 22. května 2019
Datum schválení: 28. října 2018

Abstrakt

Cílem této práce je vytvoření nástroje pro testování grafického uživatelského rozhraní webových aplikací se zaměřením na weby implementované pomocí open-source frameworku DotVVM. Účel nástroje bude spočívat v automatizovaném tvoření programového popisu uživatelského rozhraní a jeho využití při psaní UI testů spolu se statickou typovou kontrolu testů v době překladu. V práci jsou definovány požadavky pro požadovaný nástroj, rozebrán jeho návrh a popsána názorová implementace v jazyce C#. Vytvořené řešení poskytuje uživatelům nástroj, který ulehčí jejich práci s tvorbou testů a také umožňuje zjistit nefunkční testy uživatelského rozhraní již v době překladu testů a tím urychlit její tvorbu a testování.

Abstract

The goal of this thesis is to create a tool for testing graphical user interface of web applications. The tool will focus on web applications implemented by open-source framework DotVVM. The main purpose of the tool is to automatically generate program description of a user interface which will then be used as a helper class to implement UI tests and for static type check of those tests at compile-time. The thesis is defining requirements for such a tool and describing its design with implementation in C# language. The created solution provides to its users a tool which will ease their work with tests creation. Also, it will bring detection of failing tests of the user interface at compile-time which will speed up testing and development.

Klíčová slova

Testování uživatelského rozhraní, Webové aplikace, C#, Page Object vzor, DotVVM, Selenium, .NET Framework, .NET Core

Keywords

User interface testing, Web applications, C#, Page Object pattern, DotVVM, Selenium, .NET Framework, .NET Core

Citace

KALOUS, Filip. *Nástroj pro usnadnění testování GUI webových aplikací.* Brno, 2019. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Jan Pluskal

Nástroj pro usnadnění testování GUI webových aplikací

Prohlášení

Prohlašuji, že jsem tento semestrální projekt vypracoval samostatně pod vedením Ing. Jana Pluskala. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Filip Kalous
17. května 2019

Poděkování

Chtěl bych poděkovat svému vedoucímu diplomové práce Ing. Janu Pluskalovi za odborné vedení práce a užitečné konzultace. Dále bych rád poděkoval kolegovi Tomáši Hercegovi za odbornou konzultaci.

Obsah

Kapitola 1

Úvod

Současný vývoj webových aplikací si nelze představit bez formy kontroly kvality. Testování se stalo akceptovanou a nedílnou součástí tvorby softwaru, které vývojářům poskytuje zpětnou vazbu k jejich implementaci. Při vývoji softwaru je vždy možné dopouštět se chyb, které nelze jednoduše odhalit, a tím snižovat kvalitu výsledného produktu. Tomuto je možné se vyhnout pomocí testování v průběhu vývoje, čímž lze ve výsledku dosáhnout funkčnějšího a hodnotnějšího produktu.

Grafické uživatelské rozhraní (dále GUI) postupem času nahradilo textové rozhraní aplikací a aktuálně je nejpoužívanějším prostředkem interakce s uživatelem. Tento pokrok uživateli přinesl intuitivnější a méně komplikovanou cestu, jak mu sdělit nebo naopak od něj získat informace. Všeobecně se GUI skládá z interaktivních grafických objektů, například tlačítek, menu, ikon atd., které jsou zobrazeny uživateli v oknech na obrazovce. Původně bylo GUI navrženo pro ovládání klávesnicí a myší, avšak s rozvojem dotykových zařízení a hlasově ovládanými technologiemi se GUI stále vyvíjí. Velké rozšíření GUI následně vedlo k jejich větší komplexitě a jejich testování funkčnosti se stalo rozsáhlým úkolem. I v jednoduchém textovém editoru WordPad je možné provést až 325 úkonů [18]. A tedy otestovat jejich veškeré možné kombinace je nejenom složitější než u rozhraní příkazové řádky, ale také velmi časově náročné. Urychlit tuto činnost a poskytnout nástroj k usnadnění tvorby testů je cílem této práce.

Motivací pro vznik této práce je usnadnění psaní testů UI. Tyto testy je obtížné vytvořit, často se testované GUI mění pod rukama, a to následně vede k nefunkčním testům. Avšak informaci o nefunkčních testech se tester dozví s velkým zpožděním až když testy spustí. Jejich běh přitom může trvat i několik hodin. Tomu se výsledná práce snaží předejít a zajistit testerovi zpětnou vazbu již v době jejich kompilace. Dále pomáhá testerovi ušetřit čas s tvorbou testů. A to pomocí generování selektorů jednotlivých elementů ve stránce a jejich zaobalení do programových popisů využitelných v testech.

V rámci mého řešení nástroje pro usnadnění testování GUI jsem se zaměřil na webové aplikace vytvořené pomocí frameworku DotVVM. První úlohou práce bylo navrhnout celý nástroj. Na základě tohoto návrhu jsem vyvinul konzolovou aplikaci, jež se stala součástí samotného frameworku, a kterou je možné spustit nad libovolnou DotVVM aplikací. Celé řešení je složeno ze dvou hlavních částí — generátoru a pomocných objektů pracujících nad frameworkem Selenium. Aby byla podporována kontrola funkčních testů během kompilace, vytvořil jsem úlohu pomocí MSBuild platformy. Zbytek textu práce popisuje testování implementace a funkčnosti.

V kapitole 2 je popsán úvod do testování grafického uživatelského rozhraní a probrány nástroje a techniky používající se k testování GUI. Společně s tím jsou popsány a po-

rovnány výhody a nevýhody testování GUI webových aplikací. Z tohoto srovnání vychází kapitola 4 v podobě definice požadavků a návrhu výsledného nástroje. V této kapitole je nástroj popsán a rozebrán z pohledu několika dílčích částí. Kapitola 5 se zabývá popisem implementace spolu s bližším pohledem na použité technologie. Před závěrem a výhledem do budoucna následuje kapitola 6, jenž popisuje průběh testování a demonstraci funkčnosti nástroje.

Kapitola 2

Testování grafického uživatelského rozhraní

Grafické uživatelské rozhraní je aktuálně možné najít kdekoliv. Od počítačů přes mobilní telefony až například k pračkám. Je to pochopitelné, textové uživatelské rozhraní je sice jednoduché na používání, ale uživatel musí znát spoustu textových příkazů sloužících k interakci a práce s aplikací může být ve výsledku velmi zdlouhavá. Přechodem na GUI mohly vzniknout mnohem složitější webové aplikace, které by s textovým rozhraním nemohly fungovat. Tímto pokrokem se ovšem testování uživatelského rozhraní stalo obtížnější úlohou a daleko důležitější v kontextu vývoje celé aplikace.

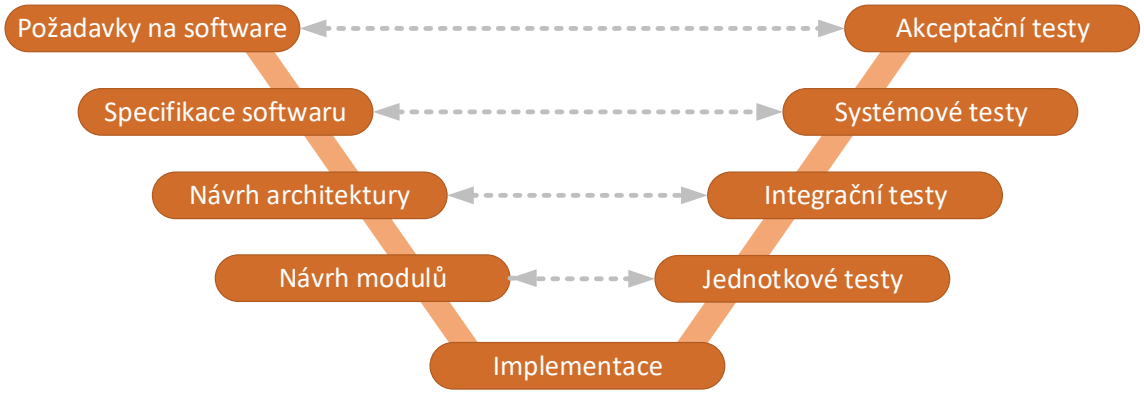
Testování GUI představuje kontrolu produktu v takové podobě, s jakou bude po dokončení pracovat uživatel. GUI se ovládá pomocí interaktivních (akčních) grafických prvků pro navigaci aplikací a manipulaci s daty, kam patří například různé typy tlačítek, menu a dialogy. Pro předání informací aplikaci a otestování funkčnosti je využito vstupních ovládacích prvků. K nim lze zařadit přepínače, seznamy a různé typy textových polí.

Je potřeba se zamyslet i nad tím, zda a kdy má smysl testování GUI provádět. V průběhu vývoje jsou využity již jiné způsoby testování, které zahrnují kontrolu funkcionality a aplikační logiky. Na tento problém je potřeba se podívat z pohledu uživatele. Ten ze začátku neřeší složitou logiku v pozadí, ale rozhodnutí zda bude používat aplikaci nadále, provede právě na základě jeho zkušeností s GUI. Jiné typy testů se navíc nezabývají tím, jak se bude aplikace chovat při běhu na jiných platformách, webových prohlížečích apod. Především automatizované testy GUI mohou nalézt regresní chyby, což přispívá k ulehčení a ušetření práce testerům.

I když výhod testování GUI je spousta, lze také nalézt řadu nevýhod, zejména u toho manuálního (více o manuálním testování v podsekci 2.2.1). Toto testování GUI je časově náročné a náchylné k chybám. Při nalezení problému v aplikaci je velmi těžké reprodukovat chybu, protože tester se nemusí řídit přesnými pokyny [11]. Během vývoje aplikace se může měnit uživatelské rozhraní aplikace, čímž se jak testy, tak testovací případy mohou rozbít a musí následovat jejich oprava.

2.1 Základní typy testů

Testování GUI je jen jedna z možností, jak webovou aplikaci testovat. Možností je hned několik, přičemž každá se zaměřuje na jiné pokrytí funkcionality a své uplatnění nalezne v jiné části vývoje. Spojení fází vývoje a odpovídajících testů lze vidět na obrázku 2.1.



Obrázek 2.1. V-Model tvorby softwaru představující, jak jednotlivé části vývoje souvisí s různými typy testování[1].

Jednotkové testy

Jednotkové testy (angl. Unit tests), kterými se detailně zabývá kniha *The Art of Unit Testing* [22], jsou části kódu zaměřující se na nejmenší testovatelné části jakékoliv aplikace. Pod pojmem jednotka je obvykle považována jedna nezávislá komponenta aplikace na nejnižší úrovní, například metoda. Následně se provede kontrola korektnosti nějakého předpokladu a na základě toho je test prohlášen za úspěšný nebo neúspěšný. Jednotkové testy si nejčastěji píší sami vývojáři, kteří znají požadovanou funkcionalitu, z důvodu ověření korektnosti jejich implementace.

Integrační testy

Kvůli tomu, že webové aplikace dosahují komplexnosti, bývají nejednou rozděleny do několika podsystémů. Integrační testy slouží ke zjištění, že výsledný program byl propojen správně a jednotlivé moduly mezi sebou mohou interagovat. Cílem integračního testování je prozkoumat, že produkt má funkční komunikační kostru. Splnění specifikačních požadavků na vyvíjený produkt není v této fázi klíčové [12, p. 130]. Na rozdíl od jednotkových testů využívají reálné závislosti a právě kvůli tomu jsou důležitým, avšak odděleným, doplňkem k jednotkovým testům [22, p. 7]. Testy jsou připravovány jak přímo testery[2], tak vývojáři.

Systémové testy

Tento druh testů se používá k ověření, že webová aplikace funguje jako funkční celek a splňuje požadavky na ni kladené. Testy se připravují podle předpřipravených scénářů, které jsou komplexní a simulují průchod aplikace běžným uživatelem. Obvykle se provádí v několika cyklech. K systémovému testování se v rámci vývoje dostává po dokončení integračních testů a končí, když je jasné, co aplikace umí, jsou opraveny všechny známé problémy a lze ji prohlásit za připravenou pro akceptační testování [12, p. 134]. Je to poslední fáze testování webové aplikace prováděnou vývojářským týmem, následuje zveřejnění nebo předání zákazníkovi k akceptačnímu testování.

[2]Člověk provádějící manuální testy nebo implementující jejich automatizaci.

Akceptační testy

Pokud není webová aplikace vyvíjena pro osobní potřebu, přicházejí na řadu akceptační testy. Ty obvykle neprobíhají v rámci vývojářského týmu, ale na straně zákazníka a jsou prováděny bez vědomí, jak aplikace funguje uvnitř. Zákazník se pomocí nich snaží zjistit, zda dodaná webová aplikace splňuje jeho vstupní požadavky, a pokud ano, je možné vývoj ukončit. Testy jsou opět vyhotoveny podle předpřipravených scénářů, na kterých spolupracuje jak zákazník, tak dodavatel.

Regresní testy

K výše probraným typům testů je nutné zmínit i regresní testování. To je možné využít ve spojitosti s každým dalším typem. Ve zkratce se jedná o opakované testování funkcí a vlastností aplikace. Jejich účelem je kontrola, zda změny provedené ve zdrojovém kódu nebo implementace nové funkcionality neovlivnila požadované chování aplikace [23, p. 427]. Jedná se tedy hlavně o kontrolu částí aplikace, které nebyly upraveny, jestli funkcionalita zůstala zachována a také jako kontrola, že se v aplikaci nevyskytují stejné chyby. Tento druh testů může být prováděn jak manuálně, tak automaticky, ačkoliv v praxi jsou regresní testy skoro vždy automatizované.

2.2 Testování GUI

Pojem testování GUI je možné definovat následujícím způsobem. „*Aplikace obsahující grafické uživatelské rozhraní je testována pouze pomocí sekvencí událostí (například kliknutím na tlačítko, zadáním textu, otevření menu) provedených nad GUI objekty (například tlačítko, vstupní pole, rozbalovací menu) [1]*“. Je tedy možné říci, že počet událostních sekvencí u komplexnějších GUI může být obrovský. Nárůst tohoto počtu se odvíjí od délky jednotlivé sekvence a je exponenciální [29, p. 3]. Z toho je patrné, že kontrola vyvíjeného produktu je složitý a časově náročný problém.

I když využívání GUI neustále roste, testování GUI dlouho zůstávalo nezkoumanou oblastí a to i přesto, že průběh a styl GUI testování je odlišný od technik konvenčního testování [15]. Neslouží jenom k zjištění, zda například kliknutí na tlačítko zobrazí nějaký formulář, ale v rámci toho může pokrývat i kontrolu komunikace a spolupráce několika vrstev aplikace, která je nutná pro úspěšné dokončení požadované události. Těchto praktik je tedy dobré využít v rámci systémového testování a následně i jako součást akceptačního.

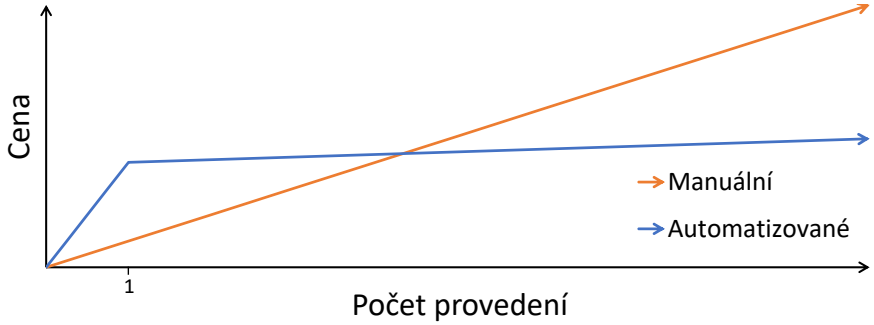
2.2.1 Přístupy k testování GUI

Během testování aplikace jsou tvořeny testovací scénáře a případy, které jsou spouštěny nad aplikací. Jejich provedení je možné zajistit dvěma způsoby — manuálně nebo automaticky. Každý z nich může být lepší za jiných okolností (například množina možných interakcí s aplikací), přičemž nejčastěji vychází kombinace obou způsobů jako nejlepší řešení [16].

Manuální testování

Ve své podstatě jde o „proklikání“ celé webové aplikace nebo provedení testovacího scénáře, které je vykonáváno jednotlivými testery. Obecně je ruční testování lepší pro scénáře, které

[2] `https://insights.sei.cmu.edu/sei_blog/2013/11/using-v-models-for-testing.html`



Obrázek 2.2. Graf reprezentující porovnání automatizovaného a manuálního testování, přesněji poměr ceny a počtu provedení testů v rámci vývoje softwaru převzatý od Alexe McPeaka [17]. Lze vidět, že cena manuálního testování roste lineárně s počtem provedení. V případě složitého testovacího scénáře, kde je větší pravděpodobnost vývojářských chyb a potřeby zjišťovat chyby, se vývoj aplikace využívající jen manuálního testování může velmi prodražit.

není nutné provádět pravidelně. Tester může manuální testování také využít pro zmapování nových funkcí a informací o vyvíjené aplikaci.

Nevýhodou manuálního testování je časová náročnost. Zaprvé, napsat testovací případy představuje složitou práci, pro kterou je potřeba kompletně porozumět chtěnému chování aplikace. Je nutné správně popsat kroky testu, navrhnout testovací data spolu s očekávanými výsledky [12]. Na obrázku 2.2 lze také vidět, že manuální testování zatěžuje nejenom časová náročnost, ale také náročnost finanční.

Zadruhé, interakcí proveditelných s aplikačním GUI je obrovské množství, přičemž s každou sekvencí příkazů může vést do jiného stavu aplikace. Toto vede k velkému množství kombinací vyžadující testování. Například společnost Microsoft vydala skoro 400 tisíc beta kopií systému Windows 95, aby našla programové chyby [18].

Automatické testování

U tohoto způsobu testování se využívá testovacího nástroje, který „proklikání“ provede za člověka. K automatickému provedení testu je potřeba testovací nástroj (pro webovou aplikaci webový prohlížeč), vstupní a výstupní data. Tester zde přebírá roli vývojáře a píše testovací skript[3]. Takto napsané testy je následně možné spouštět opakovaně jako součást ověření funkčnosti webové aplikace, například pokaždé před zveřejněním nové verze. Výsledkem automatizovaného testu ovšem není znalost, zda je systém nefunkční, jak by se mohlo zdát. Výsledkem je zaznamenaná změna od předchozího provedení stejného testu. Samotné vyhodnocení této změny už je na testerovi a jeho znalostech, jestli je chování požadované či nikoliv.

V rámci automatického testování se rozlišují dva hlavní přístupy, jak automatizace docílit. Zaprvé způsob „Capture and Replay“ vycházející z manuálního testování a zadruhé „Model Based Testing“ využívající znalostí o chování testované aplikace.

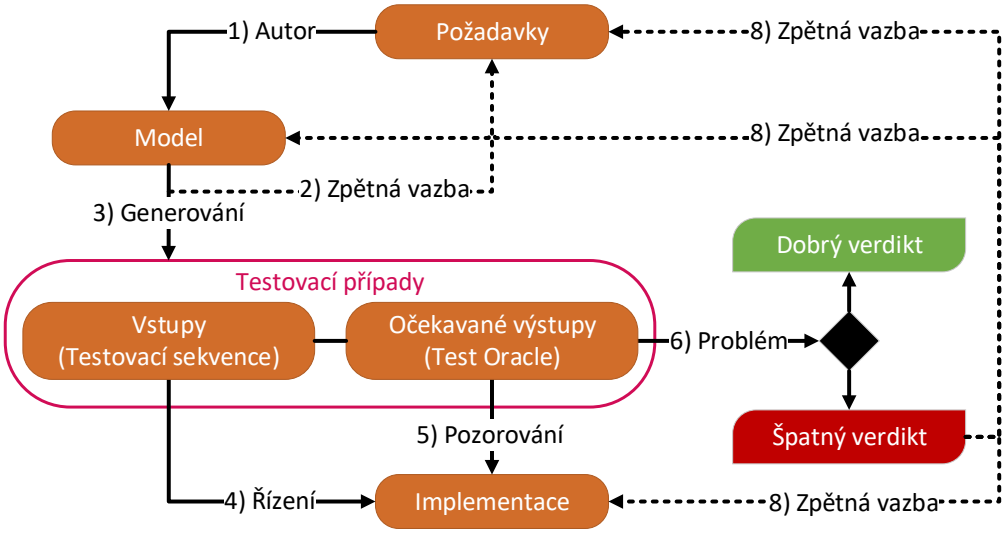
Capture and Replay (Zachytit a přehrát) testování je založeno na zaznamenání sekvence událostí provedené uživatelem (testerem) a její následné automatizované opa-

[3]Testovací skript — sekvence příkazů pro provedení testovaného případu v některém programovacím jazyce

kování nějakým nástrojem (například Selenium IDE[4]). Tento přístup je velmi jednoduchý na použití, například pro regresní testování, a tester si vystačí s minimální znalostí programování.

Model Based Testing (Testování založené na modelu) je přístup využívající k testování model GUI, který respektuje specifikaci a návrh aktuálního problému. Model popisující chtěné chování testovaného systému (angl. System Under Test, zkráceně SUT) představuje nejdůležitější část tohoto způsobu testování. Jde o další etapu v automatizovaném testování. Tento přístup je více popsán na obrázku 2.3.

Hlavním přístupem k testování založeném na modelu je využití stavů aplikace a přechodů (událostí) mezi nimi. Tento způsob stojí na potřebě otestovat všechny stavy, které popisují jednotlivé fáze aplikace [23]. Do těchto stavů se lze dostat pomocí provedení sekvencí událostí. Důležité ovšem nejsou tyto sekvence událostí, ale zjištění, zda získaný stav aplikace odpovídá předpokládanému stavu dle požadovaného chování.



Obrázek 2.3. Diagram zobrazující jak testování založené na modelu funguje. Testeři začínají s množinou požadavků, které je nutné přepracovat do strojově čitelného modelu vyjadřující veškerá chování systému splňující dané požadavky. Pro takto vytvořený model je pak možné použít nástroje pro vygenerování testovacích případů. Pomocí těchto test. případů je následně možné kontrolovat implementaci a výstupy programu [2]. Na základě této kontroly vzniká zpětná vazba. Tu je dále možné využít k opravě chyb v implementaci nebo k zjištění, že byl špatně vytvořen model chování systému.

2.2.2 Způsoby testování GUI

Na problém testování GUI se dá nahlížet z několika pohledů, přičemž pokaždé testera zajímá jiný aspekt aplikace. Nejčastěji se lze potkat s testy kontrolujícími, zda aplikace funguje tak, jak byla navržena a implementována. Za výsledek se považuje zjištění, jestli veškeré algoritmy a akce v pozadí pracují podle předpokladů. Tento přístup je nejjednodušší zautomatizovat, alespoň z větší části, a proto je využíván v největší míře. Kromě testů

[4] `https://www.seleniumhq.org/projects/ide/`

aplikační logiky, na kterou se tento postup nejvíce orientuje, se další přístupy zaměřují hlavně na samotné GUI.

Grafické uživatelské rozhraní je v aplikacích hlavně jako prostředník ke komunikaci s uživatelem. Zobrazuje akční a vstupní prvky a je tedy nutné zkontrolovat i to, že se všechny prvky zobrazují ve správnou dobu a na správném místě. Zobrazení těchto prvků může například ovlivňovat aktuální čas, stav aplikace nebo role uživatele. Jako součást testů je možné si představit otestování všech přechodů mezi jednotlivými stránkami webu pomocí navigačních prvků, validaci vstupů jednotlivých vstupních prvků atd.

Pokud už uživatel používá aplikaci s otestovanou aplikační logikou a ovládací prvky se správně zobrazují, je nutné zjistit, jestli je aplikace vůbec použitelná. Pohled na použitelnost aplikace a správný návrh je ovšem velmi subjektivní a zabírá se tím celé odvětví — User Experience[5]. Zde jsou tedy důležité i zkušenosti samotného testera, jestli dokáže odhadnout, že aplikace pracuje ideálním způsobem. Testování je také možné provádět za účasti koncových uživatelů, kde pomocí sledování jejich postupu aplikací a vyhodnocením odpovědí na různé dotazníky lze zjistit, jak se jim s produktem pracuje.

Ve výsledku je testování GUI komplikovaný problém, jež se odvíjí podle složitosti testovacích scénářů. Ty lze pokaždé pojmout z jiného pohledu. To lze vidět v následující podkapitole 2.3, kde se každý rozebraný framework zaměřuje na různé typy GUI testů. Pro shrnutí, výše probrané typy testování GUI jsou — testy aplikační logiky, testy zobrazení prvků a testy použitelnosti.

2.3 Frameworky pro testování GUI webových aplikací

Framework je pomocný nástroj programátora, který je sestaven ze skupiny knihoven tvořících kostru a zaměřuje se na konkrétní problematiku. Hlavním přínosem těchto nástrojů je, že poskytují řešení problémů pro zvolenou oblast a vývojář se následně může soustředit jen na tvorbu požadované funkcionality. Velmi často jsou frameworky založeny na některém z návrhových vzorů a přinášejí tak podporu pro tvorbu aplikace na základě tohoto vzoru. Jako příklad je možné uvést ASP.NET Framework[6] pro tvorbu webových aplikací od Microsoftu, který implementuje vzor Model-View-Controller[7] (MVC). Jak v jiných oblastech programování i pro testování grafického uživatelského rozhraní existuje několik frameworků, které slouží k usnadnění tvorby testů.

Příklady testovacích frameworků

I když je testování GUI možné provádět dvěma způsoby, manuálně a automaticky, frameworky jsou zaměřené hlavně na testování automatické. Testování manuální lze použít jako součást procesu automatizace u některých frameworků. Existuje velké množství nástrojů pro různé operační systémy, pro různé technologie a také jako otevřený software (open source) nebo jako proprietární produkt. Následující odstavce představují několik z nich.

Komerční frameworky

Proprietární software je vytvořen jednotlivcem, týmem nebo firmou, distribuovaný způsobem tak, že zdrojový kód, struktura produktu a know-how jsou tajemstvím. Na rozdíl

[5] `https://en.wikipedia.org/wiki/User_experience`

[6] `https://www.asp.net/mvc`

[7] `https://docs.microsoft.com/en-us/aspnet/core/mvc/overview`

```
1 Click "ChromeIcon"
2 TypeText "FIT VUT"
3 TypeText Return
4 WaitFor 15, "SearchResultsPage"
5 Click "FirstResult"
6 WaitFor 20, "Faculty of Information Technology"
```

Výpis 2.1. Ukázka testovacího skriptu v jazyce SenseTalk. Ukázka spočívá v otevření prohlížeče Google Chrome, zapsání řetězce „FIT VUT“ do textového pole pro vyhledávání. Následně se provede stisknutí klávesy „Enter“ pro spuštění vyhledávání. Po 15 sekundovém čekání na obrazovku s výsledky nástroj klikne na první vyhledaný záznam. Test je ukončen čekáním na načtení webové stránky FIT VUT a provedena kontrola, zda titulek obsahuje řetězec „Faculty of Information Technology“.

od softwaru dostupný jako open-source, u komerčních projektů je možné se spolehnout na podporu od vývojářské firmy. Není ovšem možné přesně říci, zda je komerční software bezpečnější než open-source.

Ranorex Studio je framework pro testování GUI vytvořený firmou Ranorex GmbH. Umožňuje testovat jak webové aplikace, tak i mobilní a desktopové. Uživateli nabízí možnost testovat GUI bez psaní kódu, využívat *Selenium WebDriver* a testovat web napříč různými prohlížeči. Ranorex Studio využívá k testování metody Capture and Replay, tedy zaznamenání uživatelova průchodu aplikací. Získaný záznam je možné libovolně upravovat, přidávat validaci nebo měnit identifikátory prvků UI (Ranorex využívá RanoreXPath[8] k identifikaci prvků ve webu).

Eggplant Functional slouží jako automatizační nástroj, který je vyvíjen od roku 2002 společností Redstone Software a následně koupený firmou Eggplant. Framework využívá k testování metodu rozpoznávání obrazu OCR. Díky tomu, testovaná aplikace může být vyvíjená jakoukoliv technologií a fungovat pod jakýmkoliv operačním systémem. Testovací skripty lze psát v jazyce SenseTalk[9] připomínajícím angličtinu (ukázka ve výpise 2.1. Testování aplikace probíhá přes RDP (Remote Desktop Protocol) nebo přes VNC server. Dle recenze *Forrester Research* je Eggplant aktuálně nejlepším automatizačním nástrojem [10].

Telerik Test Studio vzniklo v roce 2012 jako produkt společnosti Telerik. Nástroj funguje jen pod systémem Windows a je určen pro testování webových a desktopových aplikací. Produkt také umožňuje vytváření a editaci testů bez znalosti programování a funguje opět na principu Capture and Replay. Pomocí Test Studia je možné testovat funkcionalitu, REST[10] API nebo provádět výkonnostní testování. Pokud tester potřebuje změnit jednotlivé kroky testu, je toho možné docílit úpravou kódu testovacího scénáře pomocí programovacích jazyků C# a VB.NET.

[8] `https://www.ranorex.com/help/latest/ranorex-studio-advanced/ranorexpath/introduction/`

[9] `http://docs.testplant.com/ePF/SenseTalk/stk-about-sensetalk.htm`

[10] Representational state transfer — způsob jak vytvořit, číst, editovat nebo smazat informace ze serveru pomocí HTTP volání.

```
1  class SeleniumExample
2  {
3      static void Main(string[] args)
4      {
5          using (IWebDriver driver = new FirefoxDriver())
6          {
7              driver.Navigate().GoToUrl("http://www.fit.vutbr.cz/");
8
9              IWebElement query = driver.FindElement(By.Name("q"));
10             query.SendKeys("FIT VUT").Submit();
11
12             var wait = new WebDriverWait(driver,
               ↪ TimeSpan.FromSeconds(10));
13             wait.Until(d => d.Title.StartsWith("FIT VUT",
               ↪ StringComparison.OrdinalIgnoreCase));
14
15             Console.WriteLine("Title of the page is: " + driver.Title);
16         }
17     }
18 }
```

Výpis 2.2. Ukázka programu pro vyhledání řetězce „FIT VUT“ na webu `www.google.com` pomocí vstupu se jménem „q“ (vyhledávací pole) a následné čekání na zobrazení tohoto řetězce v titulku stránky. Tento titulek je poté vytisknut do konzole.

Open-source frameworky

Open-source software představuje produkt s nastavenou licencí tím způsobem, že může kdokoliv prohlížet, sdílet i upravovat jeho zdrojový kód. Za výhodu open-source frameworků lze považovat jejich distribuci zdarma, což může hrát velkou roli při výběru nástroje pro použití ve vyvíjené aplikaci. Jako další výhodu lze vzít již zmíněný zpřístupněný zdrojový kód. Stinnou stránkou pak může být chybějící podpora, zvláště u open-source projektů bez firmy v pozadí (například open-source projekt RoundCube[11] nemá firmu v pozadí). Oproti tomu stojí .NET Core[12], který je vyvíjen především společností Microsoft.

Selenium je aktuálně nejznámějším open-source frameworkem pro automatické testování. Vývoj byl započat v roce 2004. Je to multiplatformní balíček nástrojů, vyvíjený v jazyce Java, které jsou zaměřeny na různé přístupy k automatizaci testování skládající se ze Selenium IDE, Selenium WebDriver, Selenium RC (starší a již nepodporovaná verze Selenium WebDriver) a Selenium Grid (umožňuje distribuované testování[13]) [25].

První variantou jak vytvářet automatické testy pomocí tohoto frameworku, je Selenium IDE. Tento nástroj využívá metody Capture and Replay jako některé z komerčních produktů. Výsledný záznam představuje sada pokynů ve formě HTML tabulek. Dále je možné testovací scénář implementovat pomocí skriptu, případně upravit testovací skript, který byl

[11] RoundCube — webový emailový klient, více na `https://roundcube.net`.

[12] `https://dotnet.github.io`

[13] Distribuované testování — spočívá v paralelním spuštění několika testů proti různým strojům, operačním systémům i webovým prohlížečům ve stejném čase.

```
*** Settings ***
Documentation Single test for valid login.
Resource resource.robot

*** Test Cases ***
Valid Login
    Open Browser To Login Page
    Input Username demo
    Input Password mode
    Submit Credentials
    Welcome Page Should Be Open
    [Teardown] Close Browser
```

Výpis 2.3. Testovací případ pro validní přihlášení uživatele pomocí Robot frameworku.

```
*** Variables ***
${BROWSER} Firefox
${DELAY} 0
${LOGIN URL} http://${SERVER}/

*** Keywords ***
Open Browser To Login Page
    Open Browser ${LOGIN URL} ${BROWSER}
    Maximize Browser Window
    Set Selenium Speed ${DELAY}
    Login Page Should Be Open

Login Page Should Be Open
    Title Should Be Login Page
```

Výpis 2.4. Ukázka definic klíčových slov ze souboru *resource.robot*[14] pro případ z výpisu 2.3.

vytvořen ze záznamu uživatelovi interakce s aplikací. Nevýhodou tohoto nástroje je závislost na doplňcích do prohlížečů Google Chrome a Mozilla Firefox.

Druhou variantou a také nejpoužívanější pro vytváření automatických testů pomocí Selenium je Selenium WebDriver. Ten umožňuje psaní testovacích programů v několika jazycích (C#, Java, Python, Ruby, ...). Po spuštění programů Selenium provede veškeré operace stejným způsobem, jako by byly prováděny uživatelem. Selenium WebDriver je často základ pro další frameworky, které ho využívají jako svoji součást. Nevýhodou tohoto přístupu je skutečnost, že pro každý prohlížeč je nutné mít vlastní WebDriver, který s ním bude umět pracovat. Ukázku programu lze vidět ve výpisu 2.2.

Robot Framework poprvé vyšel v roce 2005 a vývoj pokračuje dodnes. Je nezávislý na operačním systému a technologii použité k vývoji aplikace. Implementovaný je v jazyce Python, pomocí knihoven Jython resp. IronPython beží i pod Javou resp. .NET frameworkem.

Syntaxe testů se liší od ostatních frameworků popsaných v této sekci. Je založena na klíčových slovech. Ta lze vytvářet nová nebo založená na již existujících klíčových slovech. Framework v základu obsahuje standardní knihovny s klíčovými slovy používané pro jednodušší testy. Další knihovny, například pro pokročilou práci s řetězci, lze stáhnout dodatečně.

Cypress představuje JavaScript framework pro testování webových aplikací zaměřený na „End-to-end“ testování. To představuje spuštění webové stránky v prohlížeči a provedení operací jako kdyby prohlížeč ovládal reálný uživatel. Dále je možné psát jednotkové a integrační testy. Cypress ke svojí činnosti nevyužívá Selenium, které by obalil vlastní nadstavbou a přidal funkcionalitu navíc, jako to dělá spousta jiných frameworků (např. Serenity[15], Selenide[16]). Je s ním ale často srovnáván. Protože Cypress využívá pro veške-

[14] `http://robotframework.org/#examples`
[15] `http://www.thucydides.info/#/`
[16] `https://selenide.org`

rou práci jenom JavaScript, funguje napříč platformami a nezávisle na technologii použité k tvorbě webové stránky. Přímo součástí frameworku Cypress je spousta jiných užitečných open-source JavaScript projektů (například Mocha[17], Chance[18], jQuery[19], Chai[20]).

Shrnutí

V rámci této kapitoly jsem představil několik typů testů, přístupů k testování a frameworků pro testování GUI webových aplikací. V rámci popisu přístupů k testování jsem se zaměřil především na automatizované testování, z důvodu jeho větší využitelnosti v této práci. Další části bylo porovnání open-source a komerčních frameworků, srovnání jejich výhod a nevýhod, spolu s několika příklady. Největší prostor byl věnován aktuálně nejpoužívanějšímu nástroji Selenium.

Ačkoliv testování GUI zní jako komplikovaný úkol, není radno ho opomínat v rámci vývoje aplikace. Uživatele nebude zajímat, že v pozadí běží složitá logika, s kterou vývojář strávil několik týdnů. Uživatel i zákazník vždy uvidí jen uživatelské rozhraní a pokud se vyskytne nějaký problém, například nezobrazení tlačítka, špatná navigace nebo složitá orientace v aplikaci, nebude spokojený a produkt nebude používat, případně nedoporučí ostatním. Tím pádem, otestování toho, že GUI funguje, jak má, může ušetřit spoustu problémů a času.

[17] `https://github.com/mochajs/mocha`
[18] `https://chancejs.com`
[19] `https://jquery.com`
[20] `https://www.chaijs.com`

Kapitola 3

DotVVM

DotVVM[1] je open-source webový framework vytvořen nad ASP.NET Core a ASP.NET pro tvorbu webových aplikací. Framework ke své práci používá návrhový architektonický vzor Model-View-ViewModel (dále jen MVVM).

3.1 Vzor Model-View-ViewModel

Architektonický vzor MVVM představuje oddělení definice uživatelského rozhraní od business logiky v pozadí. Tento vzor vznikl jako odpověď k tradičnímu vývoji UI, kdy View obsahovalo nejenom popis UI, ale i veškerou logiku potřebnou k chodu aplikace. Z toho plynou problémy s velikostí souboru obsahující View a velkou závislostí mezi UI, datovými operacemi a business logikou [28]. Tímto přístupem se zvyšuje náročnost úprav UI a složitost tvorby jednotkových testů kódu [21]. Jako řešení těchto problémů, vzor MVVM rozděluje aplikaci do tří vrstev — View, ViewModel a Model. Jednotlivé části jsou blíže popsány v obrázku 3.1.

Obrázek 3.1. Schéma spolupráce architektonického vzoru Model-View-ViewModel. Část View definuje strukturu, vzhled a rozložení uživatelského rozhraní. Každé View je navázáno na vlastní ViewModel. Pomocí příkazů a data bindingu[2] je View spojeno s ViewModelem, který představuje most mezi View a Modelem. Majorita frameworků postavených nad MVVM vzorem umožňuje definovat směr, kterým je možné data navázat z View na vlastnosti ViewModelu. Výchozí směr je pak obousměrný. ViewModel obsahuje stav aplikace a pracuje s daty. Typicky ViewModel využívá metody zprostředkované Modelem, získává z něj data a následně je předává View ve formě, s kterou View zvládne pracovat [21]. Model tedy představuje datovou část aplikace, společně s aplikační logikou.

[1] `https://www.dotvvm.com`

[2] Data binding — technika využívaná pro svázání zdroje dat s jejich použitím a vzájemnou synchronizací.

```
1  @viewModel DotVVMExample.ViewModels.TransportViewModel, DotVVMExample
2
3  <div>
4      <label>Text: </label> <dot:TextBox Text="{value: Text}" />
5  </div>
6  <div>
7      <dot:Button Text="Zjisti delku"
8                  Click="{command: CountChars()}" />
9  </div>
10 <div>
11     <label>Delka retezce: </label>
12     <dot:TextBox Text="{value: CharCount}" />
13 </div>
```

Výpis 3.1. Ukázka View ve frameworku DotVVM. Komponentu `<dot:TextBox>`, resp. `<dot:Button>` pak DotVVM přeloží do odpovídajícího elementu HTML `<input type="text">`, resp. `<input type="button">`.

Vzor MVVM v DotVVM

Pro realizaci architektonického vzoru MVVM ve frameworku DotVVM je využito knihovny Knockout[3]. Tato Javascriptová knihovna je využita na klientské straně aplikace pro vytvoření bindingů mezi hodnotami v UI a ViewModely přeložené do JavaScriptu. Také se stará o to, že při každé změně dat ve ViewModely jsou patřičně aktualizovány elementy v UI. Serverová část aplikace DotVVM obsahuje ViewModely naprogramované v jazyce C#. Ty jsou pak přenášeny mezi klientem a serverem v serializované podobě pomocí JSONu[4]. Server také provádí generování HTML kódu s již obsaženými Knockout bindingy, který je poté poslán na klienta [19].

Stránky webové aplikace v DotVVM jsou vždy tvořeny dvěma soubory, jeden pro View, druhý pro ViewModel. Pro definici View DotVVM využívá klasické HTML s přidanou syntaxí pro data binding výrazy a vlastní DotVVM komponenty [5]. Ukázka definice View je k vidění ve výpisu 3.1. ViewModel je pak klasická veřejná C# třída obsahující jakékoliv vlastnosti s hodnotami, které je možné serializovat do JSONu, tedy vlastnosti pro hodnoty zobrazené ve View a metody pro interakci s nimi. Jak vypadá ViewModel potřebný pro View z ukázky 3.1 lze vidět ve výpisu 3.2.

3.2 DOTHTML dokument (View)

View v DotVVM je definováno v souborech s příponou *.dothtml*, přičemž obsahuje převážně klasické HTML a CSS, jak je zvykem při tvorbě webových stránek. Framework DotVVM také obsahuje vlastní komponenty, které jsou následně přeložené do ekvivalentních HTML prvků spolu s bindingy knihovny *Knockout.js*. Každá komponenta může specifikovat několik vlastností ve formě atributů nebo dětských prvků, což je definováno vývojářem [4]. Zároveň je u většiny prvků dovoleno využít klasické HTML atributy (například *style, class* atd.).

[3] `https://knockoutjs.com`
[4] `https://www.json.org`

```
1  namespace DotVVMExample.ViewModels
2  {
3      public class TransportViewModel
4      {
5          public string Text { get; set; }
6          public int CharCount { get; set; }
7
8          public void CountChars() => CharCount = Text.Length;
9      }
10 }
```

Výpis 3.2. Ukázka třídy `TransportViewModel` reprezentující ViewModel. Ten koresponduje k View v ukázce 3.1 ve frameworku DotVVM.

Každé View nejdříve definuje direktivy, které se využívají například pro spojení View s odpovídajícím ViewModelem. Direktiva se značí znakem zavináče, po kterém následuje její typ a hodnota. Příklad direktivy `@viewModel` lze vidět v ukázce 3.1 na řádku č. 1.

3.2.1 Master pages

Aby framework umožňoval sdílet některé částí webové aplikace mezi více stránky, podporuje DotVVM mechanismus nazývaný Master Pages [7]. Pro stránky typu *Master Page* existuje vlastní souborová přípona *.dotmaster*. Tento soubor může také obsahovat jakoukoliv HTML i DOTHTML syntaxi a navíc jednu či více komponent typu `<dot:ContentPlaceHolder>` fungující jako placeholder pro dílčí obsah. Každý placeholder má vlastní atribut ID, pomocí nějž se na něj odkazuje. Pro vložení obsahu do placeholderu v *Master Page* je ve View využito elementu `<dot:Content>` s atributem `ContentPlaceHolderID` odpovídajícím ID definovanému v *Master Page*. Pro využití *Master Page* ve View je nutné využít direktivy `@masterpage` s hodnotou relativní cesty ke konkrétní *Master Page* využité ve View.

3.2.2 Uživatelské komponenty

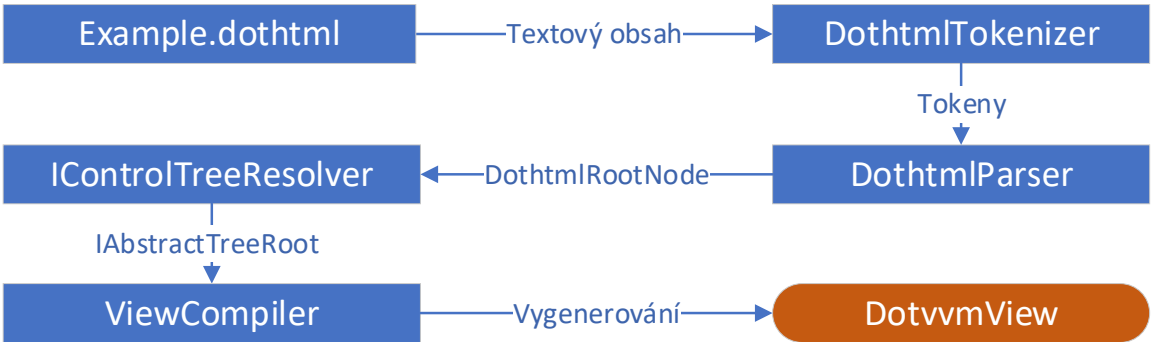
Další možností, kterou má uživatel při tvorbě DotVVM webové aplikace, je vytvoření si vlastních komponent. Ty se dělí na dva typy — *markup* a *code-only* komponenty [3]. *Markup* komponenty obsahují jen DOTHTML vložené do vlastního souboru s příponou *.dotcontrol*, které je následně možné využít v jakémkoliv View. *Code-only* prvky spočívají ve vytvoření výsledného DOTHTML pomocí jazyka C# a využívání nástrojů a metod, které poskytuje framework DotVVM. Příkladem *code-only* komponent jsou všechny elementy dostupné ve frameworku DotVVM [3]. Veškeré *uživatelské komponenty* je nutné zaregistrovat v DotVVM konfiguraci, aby je bylo možné využít při vlastní tvorbě webové stránky.

Každá *Markup* komponenta musí obsahovat direktivu `@viewModel` specifikující datový kontext, který v ní bude využit. Může to být jak rozhraní, tak specifická třída. Jestliže komponenta neobsahuje žádný *binding*, může být jako datový kontext použita bázová třída `System.Object`, čímž je dáno najevo, že datový kontext může být jakýkoliv [6]. V rámci *Markup* komponent lze také pomocí direktivy `@baseType` specifikovat třídu, která obsahuje logiku specifickou pro konkrétní komponentu. Tato třída se nazývá *code behind*. K jejím vlastnostem a metodám se pak přistupuje s připojením prefixu `_control.` nebo pomocí speciálních *binding* výrazů `controlProperty` a `controlCommand`.

3.2.3 Kompilace DOTHTML stránek

Samotná dokumentace frameworku DotVVM informace o kompilaci DOTHTML nepopisuje. Informace o tomto procesu nejsou dostupné jinak než prozkoumáním zdrojového kódu [8] v repozitáři na serveru GitHub. Následující odstavce pak obsahují shrnutí konverzace s autory DotVVM na toto téma.

Proces kompilace DOTHTML dokumentu začíná ve třídě `DothtmlTokenizer`, jenž provede lexikální analýzu. Rozdělí textový řetězec reprezentující obsah DOTHTML dokumentu na jednotlivé tokeny. S výstupem ve formě seznamu položek typu `DothtmlToken` dále pracuje třída `DothtmlParser`. Ta provede syntaktickou analýzu a z tokenů sestaví strom, který přesně reprezentuje strukturu HTML. Zatím nerozumí tomu, co je komponenta, co je šablona v elementu `<dot:Repeater>` apod. Produktem zpracování tokenů je sestavený strom dostupný z kořenu, jenž reprezentuje typ `DothtmlRootNode`. Další částí v procesu kompilace je *Resolver*, jehož úkolem je sestavit *ResolvedTree* (abstraktní strom), který každý element a atribut v HTML mapuje na konkrétní komponentu a její vlastnosti. V tomto kroku už je například možné zjistit, že uvnitř komponenty `<dot:Repeater>` neexistuje nic jako jeho potomci (*Children*), ale že k němu patří vlastnost *ItemTemplate*, která obsahuje šablonu reprezentovanou nějakou jinou částí stromu. V poslední fázi je spuštěn kompilátor `ViewCompiler`, který vezme zjištěný *ResolvedTree* a vygeneruje z něj C# třídu reprezentující vstupní DOTHTML stránku.



Obrázek 3.2. Schéma procesu DotVVM kompilace

Výsledkem je C# třída, která je zkompilována do *assembly* a při první návštěvě stránky nahrána do paměti. Dále je již využívána jen zkompilována verze stránky. Celý tento proces se provádí kvůli optimalizaci výkonu, jelikož parsování stránky pro každé její vyžádání by trvalo velmi dlouho. Jak probíhá proces kompilace DOTHTML stránky je možné vidět na obrázku 3.2.

Kapitola 4

Návrh

Tato kapitola popisuje návrh nástroje pro usnadnění testování GUI webových aplikací. První částí kapitoly je definice požadavků, v které se na základě teoretických znalostí a zkušeností pojednává o funkčních potřebách výsledného nástroje. V dalších sekcích je probrán návrh jednotlivých částí nástroje. A to automatická úprava HTML souborů pro identifikaci komponent v HTML a zároveň s tím koncept generátoru sloužící pro vytvoření programového popisu jednotlivých View v aplikačním GUI. Dále pak návrh využití získaného programového popisu k samotnému testování GUI.

Námět nástroje s definicí funkčních požadavků vychází především z nevýhod aktuálního stavu testování GUI webových aplikací. Cesta k funkčním požadavkům začíná popisem a ukázkou aktuálního stavu testování GUI. Z této ukázky a rozpoznání nevýhod vznikají nápady a potřeby, které definují požadavky na nástroj. Na základě nich jsou navrženy dílčí části nástroje s ohledem na přístupy, použitelné nástroje a frameworky popsané v předchozí kapitole 2. Výsledkem kapitoly Návrh je návrh řešení výsledného nástroje (generátoru) splňující veškeré definované funkční požadavky.

4.1 Definice požadavků

V předchozí kapitole je z několika pohledů popsáno, jak dělat testování GUI u webových aplikací. Testy GUI mají avšak i své nevýhody, které je potřeba vzít v potaz a pokusit se jim v průběhu návrhu nástroje vyhnout. V následujících odstavcích bude demonstrováno, jak probíhá postup testování GUI přihlašovacího formuláře nyní. Na základě ukázky a odhalených nevýhod bude výsledkem této části seznam definovaných funkčních požadavků.

Příklad testování GUI formuláře

Jako demonstrace testování GUI bude sloužit přihlašovací formulář z výpisu 4.1. Demonstrační test je napsán s využitím frameworku Selenium (podkapitola 2.3) ve výpisu 4.2. I když obsahuje jen několik interaktivních prvků, pro účely této demonstrace bohatě postačí.

Pro úspěšné provedení testu, ať už manuálně nebo automaticky, je nutné umět získat referenci na prvky, které budou potřeba. V demonstračním příkladu je využito HTML5[1] atributu „data-*“, přesněji atributu „data-ui“ (například na řádku č. 6 identifikátor data-ui=”signIn-email” pro emailový vstup). Atributy jsou následně využity pro nalezení jed-

[1] `https://www.w3schools.com/html/html5_intro.asp`

```
1  <form class="form-signin col-2">
2      <label for="inputEmail" class="sr-only">Email</label>
3      <dot:TextBox Text="{value: Email}"
4                   Validator.Value="{value: Email}"
5                   type="email" class="form-control" placeholder="E-mail"
6                   data-ui="signIn-email" />
7
8      <label for="inputPass" class="sr-only">Heslo</label>
9      <dot:TextBox Text="{value: Password}"
10                  Validator.Value="{value: Password}"
11                  type="Password" class="form-control" placeholder="Heslo"
12                  data-ui="signIn-pass" />
13
14     <dot:Button Text="Přihlásit se"
15                 Click="{command: OnSubmitClicked()}"
16                 type="submit" class="btn btn-primary btn-block btn-lg"
17                 data-ui="signIn-submit" />
18
19     <p IncludeInPage="{value: SignInSuccess}" data-ui="signIn-success">
20         Přihlášení bylo úspěšné.
21     </p>
22     <dot:ValidationSummary data-ui="signIn-errors" class="danger" />
23 </form>
```

Výpis 4.1. Ukázka formuláře sloužící pro přihlášení napsaného s využitím frameworku DotVVM. Formulář obsahuje dvě vstupní pole a tlačítko `<dot:Button>` pro odeslání formuláře. Vstupní pole `<dot:TextBox>` slouží pro zadání emailu (řádek č. 3) a hesla (řádek č. 9). Pod tlačítkem na řádku č. 14 se nachází odstavec, který je zobrazen v případě úspěšného přihlášení. Na řádku č. 23 je zobrazena komponenta `<dot:ValidationSummary>` pro výpis validačních chyb do seznamu, v případě neúspěšného přihlášení.

notlivých prvků v testu, jak je ukázáno ve výpisu 4.2 na řádcích č. 4, 6 a 8. Prvním krokem testu je odeslání dat metodou `SendKeys` nad Selenium elementy do jednotlivých vstupních polí a druhým, provedení kliku potvrzovacím tlačítkem. Posledním krokem je vyhodnocení testu, což v tomto případě znamená, že je možné nalézt ve stránce odstavec s identifikátorem „signIn-success“. Při kontrole zadání nevalidních informací by test obsahoval kontrolu, že seznam vnořený v komponentě `<dot:ValidationSummary>` (řádek č. 23) obsahuje alespoň jednu položku.

Již ze základního výpisu 4.1 lze poznat několik problémů, které mohou nastat při dalším vývoji. Vývojář může změnit strukturu HTML, například přesunem vstupního pole pro heslo do jiného View, změnit typ vstupního pole apod. Dalším problémem je změna nebo odstranění identifikátoru na komponentách. Všechny tyto možnosti povedou k nefunkčním testům, aniž by o tom vývojář nebo tester věděl. Lze ovšem udělat i změny, které neovlivní vyhodnocení testu. Ve výpisu 4.1 na řádku č. 6 lze vidět, že vstup pro email obsahuje identifikátor „signIn-email“. Pokud se ale smysl vstupního pole změní, například místo emailu by uživatel zadával uživatelské jméno a identifikátor nebude změněn, test bude sice nadále

```
1  [TestMethod]
2  public void TestLoginSuccess()
3  {
4      // Arrange
5      var emailInput = driver.FindElement(
6          By.CssSelector($"[data-ui*='signIn-email']"));
7      var passInput = driver.FindElement(
8          By.CssSelector($"[data-ui*='signIn-pass']"));
9      var submitBtn = driver.FindElement(
10         By.CssSelector($"[data-ui*='signIn-submit']"));
11     // Act
12     emailInput.SendKeys("test@test.com");
13     passInput.SendKeys("test");
14     submitBtn.Click();
15     // Assert
16     wait.Until(d => d.FindElement(
17         By.CssSelector($"[data-ui*='signIn-success']")));
18 }
```

Výpis 4.2. Testovací metoda v jazyce C#, která pomocí frameworku Selenium (2.3) testuje zadání validních dat do formuláře popsaného ve výpisu 4.1. Pro vyhledání jednotlivých vstupů je využito CSS selektorů, tak jak je popsáno v knize *Selenium WebDriver Recipes in C#* [30].

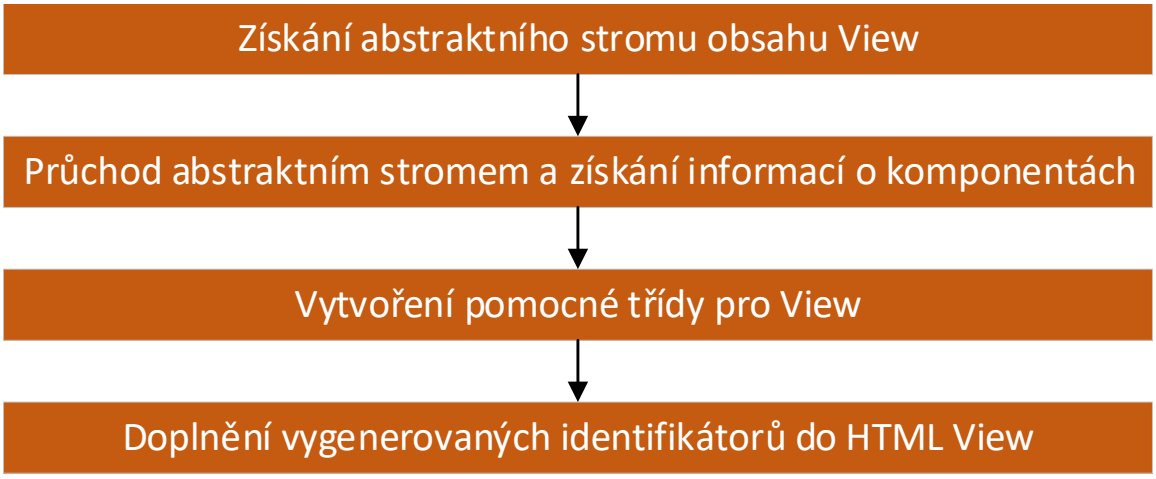
fungovat, ale identifikátor nebude dávat smysl. Pro vyřešení těchto nedostatků a problémů vzniká první funkční požadavek pro výsledný nástroj:

- Požadavek č. 1 — Nástroj musí být schopný automaticky generovat identifikátory pro známé komponenty View webové stránky. Identifikátory budou vpisovány přímo do View a budou generovány podle aktuálního kontextu a obsahu stránky pro lepší srozumitelnost.

Řešení problémů ze strany View už je definováno, další problémy jsou na straně testovacího kódu. Jak lze vidět ve výpisu 4.2 na řádcích č. 5, 7 a 9, je nutné identifikátory definované v UI webové stránky přepisovat i do testovací metody. Z tohoto důvodu se po změně identifikátoru stanou testy nefunkční vzhledem k vyvolání výjimky `OpenQA.Selenium.NoSuchElementException`, znamenající nenalezení elementu ve View. Aby se tomuto problému předešlo, je zapotřebí propojit vygenerované identifikátory ve View s elementy potřebnými k testování. Tím vznikají další dva funkční požadavky na výsledný nástroj související s referencemi na komponenty v uživatelském rozhraní:

- Požadavek č. 2 — Nástroj vygeneruje pomocné třídy využitelné v testovacích metodách pro každé View a komponenty v něm nástroji známé. Tato pomocná třída bude obsahovat reference na komponenty ve View.
- Požadavek č. 3 — Nástroj bude obsahovat základní balíček manuálně napsaných objektů obalující Selenium framework, který rozumí kontextu komponent.

Nástroj tedy zvládne vygenerovat identifikátory pro komponenty a k tomu pomocnou třídu, která bude schopná pomocí reference získat element a pracovat s ním. Pokud je



Obrázek 4.1. Obrázek ukazující navržený postup generátoru při generování pomocné třídy a identifikátorů do View.

pomocná třída využita v testech a následně je komponenta upravena nebo odstraněna, bude potřeba spustit nástroj znovu, aby vygenerované součásti reflektovaly aktuální stav webové stránky. V nejlepším případě bude úprava a generování identifikátorů spolu s pomocnými třídami spouštěna při kompilaci webové aplikace. Z toho plyne další funkční požadavek:

- Požadavek č. 4 — Nástroj bude spustitelný tak, aby zajišťoval statickou kontrolu testů v době překladu aplikace. Nebude tedy možné sestavit UI testy nad webovou aplikaci.

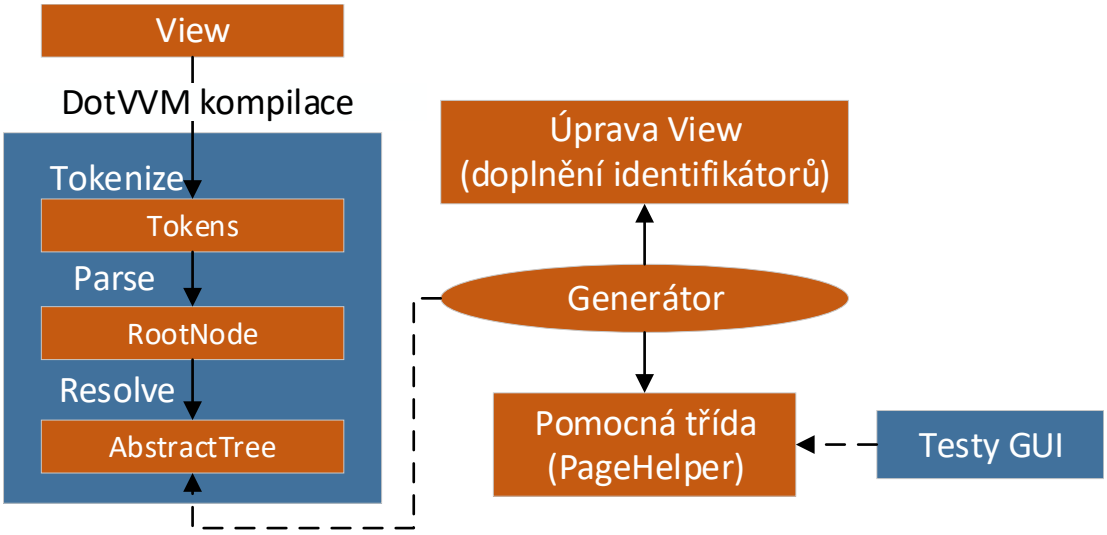
Tato sekce popsala skrze demonstrační příklad, jaké nevýhody a časově náročné úkony v sobě testování GUI obsahuje. Na ukázce HTML stránky s přihlašovacím formulářem 4.1 a ukázce testovací metody 4.2 byly představeny jednotlivé kroky testování GUI a na základě nich definovány funkční požadavky nástroje.

4.2 Návrh generátoru

V rámci návrhu generátoru budu vycházet z předchozí části, kde byly specifikovány funkční požadavky nástroje. Přesněji se ze začátku budu věnovat prvním třem požadavkům, které souvisí s tím, co bude nástroj generovat, nikoliv poskytovat testerům. Jak zajistit statickou kontrolu v době překladu a jak využít vygenerované programové popisy a pomocné objekty popisuje až podsekce 4.2.3.

Ze začátku této sekce je generátor popsán z hrubšího pohledu na věc, ještě tu nejsou řešeny veškeré detaily generování, ale spíš abstraktní napojení na framework a jak by generátor ve výsledku mohl fungovat. V dalších podčástech je již detailněji navrhnuto generování pomocných tříd a identifikátorů, spolu s návrhem výstupů. Samotný návrh pak bude probíhat po částech, jak jsou zobrazeny na obrázku 4.1. V rámci návrhu je přihlíženo k faktu, že nástroj bude vyvíjen v prostředí .NET pomocí jazyku C# a jako rozšíření pro open-source framework DotVVM.

Generátor pro svoji práci bude využívat vnitřní reprezentace webové stránky, tak jak ji definuje framework DotVVM. Jak bylo definováno v požadavcích, důraz bude kladen na automatičnost řešení a generátor by tedy neměl od uživatele vyžadovat žádné vstupy, kromě zadání projektu, nad kterým má být spuštěn.



Obrázek 4.2. Diagram zobrazující napojení generátoru na API DotVVM kompilace, konkrétně na zjištěný abstraktní strom webové stránky. Získaný abstraktní strom je následně využit pro průchod stránkou a nalezení využitých komponent.

Webové aplikace se většinou neskládají z jedné stránky, ale jsou složeny z několika View, kterými se dá různě navigovat. V rámci generátoru je ovšem zapotřebí získat informace o jedné konkrétní stránce a pro ni vytvořit programový popis, který bude moci tester využít. Informace z dalších částí aplikace jsou irelevantní a generátor tedy může stránky zpracovávat sériově. Jako vstupní parametr generátor bude potřebovat cestu ke konkrétnímu View, to bude následně využito pro získání jeho abstraktního stromu. Napojení generátoru na kompilaci DotVVM a výstupní abstraktní strom View je představeno na obrázku 4.2.

Abstraktní strom v DotVVM představuje vlastní reprezentaci stromu v HTML dokumentu a popisuje obsah a strukturu DOTHTML[2] dokumentu. Z důvodu, že součástí obsahu webové stránky nejsou jenom akční a vstupní prvky, bude nutné strom projít systematicky. Nemělo by význam přiřazovat identifikátor každé komponentě webové stránky, ale v rámci průchodu stromem vybírat komponenty, které jsou pro testera zajímavé. Z takto získaných komponent je následně možné sestavit pomocnou třídu. Ve výsledku tedy bude třída představovat celý jeden blok GUI, který může zastupovat například celou stránku nebo její podčást [24]. Tímto způsobem se sníží opakování psaní kódu pro vyhledání komponent ve stránce, tester může pracovat s větší úrovní abstrakce a jedna pomocná třída bude moci být využita ve více testech zároveň [14].

4.2.1 Generování pomocných tříd

Pomocná třída (dále `PageObject`) při testování GUI v podstatě představuje princip *PageObject*[3]. `PageObject` zaobalí veškeré komponenty ve View, pro které bude generován. Pro testování GUI již bude testerovi stačit použít takové pomocné třídy, které vyžaduje pro provedení konkrétního testovacího scénáře. Jednotlivé třídní vlastnosti u třídy `PageObject` budou představovat propojené komponenty ve View, ale spojení bude navázáno na název

[2] DOTHTML — formát dokumentu definující View ve frameworku DotVVM[5].
[3] `https://martinfowler.com/bliki/PageObject.html`

```
1 <body>
2     <dot:Literal Text="{value: SampleText}" data-ui="sampleSelector" />
3 </body>
```

Výpis 4.3. Obsah těla (body) webové stránky *Test.dothtml*. Komponenta `<dot:Literal>` slouží k vytisknutí hodnoty vlastnosti `SampleText` do webové stránky.

```
1 public class TestPageHelper : PageHelperBase
2 {
3     public Literal SampleText { get; }
4     public TestPageHelper(IWebDriver webDriver) : base(webDriver)
5     {
6         SampleText = new Literal(this, "sampleSelector");
7     }
8 }
```

Výpis 4.4. Výpis obsahující návrh pomocné třídy `TestPageHelper` představující `PageHelper` webové stránky Test.dothtml z výpisu 4.3. Veřejná třída `TestPageHelper` popisuje pomocí třídních vlastností jednotlivé komponenty (kontrolky) dostupné ve View. Například na řádku č. 3 lze vidět vlastnost `SampleText` představující komponentu `<dot:Literal>` na řádku č. 2 ve výpisu 4.3. V rámci konstruktoru třídy `TestPageHelper` pak bude nastaven identifikátor pro danou vlastnost. Identifikátor je nutný znát pro referencování elementu ve View pomocí atributu, jak bylo popsáno v příkladu testování (podkapitola 4.1).

identifikátoru komponenty, nikoliv na název třídní vlastnosti. Návrh a obsah pomocné třídy je blíže představen ve výpisu 4.4.

Aby bylo možné z konkrétních tříd `PageObject` s jednotlivými komponentami pracovat, bude nutné vytvořit další objekt, který zná kontext dané komponenty. Ve výpisu 4.4 na řádku č. 3 je tento objekt pojmenován `Literal`. Díky pomocnému objektu bude možné ovládat propojený element ve View. Například pro komponentu `<dot:Literal>`, která slouží pro zobrazení textu v dokumentu, by objekt `Literal` obsahoval metodu pro získání hodnoty zobrazeného řetězce (příklad lze vidět ve výpisu 4.5). Pro komponentu `<dot:Button>` by ideální metodu představoval `Click()`, pro provedení akce kliku na odkazovaném tlačítku.

Z důvodu, že pomocný objekt pro `PageObject` musí znát kontext konkrétní komponenty, bude nutné je připravit jako součást nástroje. Pro pokrytí základních operací (například vyhledání komponenty apod.) je ve výpisu 4.5 navržena bázová třída `ComponentBase`, čímž se sníží množství stejného kódu v jednotlivých třídách. Další součástí generátoru budou dílčí generátory pro každou komponentu. Ty využijí pomocných objektů jako `Literal`, informací o komponentách a kontextu aplikace pro vytvoření tříd `PageObject`.

> *Ve výsledku lze `PageObject` spolu s pomocnými objekty typu Literal, Button atd. a dílčími generátory považovat za splnění požadavků č. 2 a 3.*

4.2.2 Generování identifikátorů

Pro správnou funkci nástroje bude nutné najít a rozpoznat elementy ve stránce. Například v Selenium lze prvky vyhledávat pomocí osmi lokátorů. Z toho jsem vybral několik lokátorů, podle kterých bych mohl vyhledávat v mém řešení:

```
1 public class Literal : ComponentBase
2 {
3     public PageHelperBase Helper { get; set; }
4     public string Selector { get; set; }
5
6     public Literal(PageHelperBase helper, string selector)
7         : base(helper, selector) { }
8     public string GetText() => GetElement().Text;
9 }
```

Výpis 4.5. Návrh objektu (třídy) `Literal` s vlastnostmi pro uložení informací o elementu a metodou `GetText()` představující objekt s kontextem komponenty `<dot:Literal>`. Účelem je umožnit nalézt element ve stránce a provést nad ním odpovídající operace. V tomto případě operaci získání textu (řádek č. 8). V rámci metody `GetText()` bude nutné spustit metodu, která například s využitím frameworku Selenium nalezne element (v ukázce metoda `GetElement`).

- podle ID atributu,
- podle „name“ atributu (prvek ve formuláři),
- podle CSS[4] třídy elementu nebo CSS selektoru,
- pomocí XPath[5] (využití cesty stromem k elementu),
- podle textu v elementu.

Všechny způsoby jsou validní cestou, jak se dostat k požadovanému elementu. Ovšem v některých případech může fungovat skvěle vyhledání podle CSS selektoru, v jiném případě už bude potřeba využít alternativní způsob pro úspěšné vyhledání. Pro bezproblémové zjištění komponenty bude potřeba využívat stále stejný způsob. Atribut ID má již z definice jistotu, že v jednom HTML dokumentu bude unikátní [13]. Tento atribut je ale často měněn, případně generován při vývoji webu. To značí, že by nebylo rozumné se na něj spoléhat v rámci testování. Jeho vlastnosti by ovšem nástroji prospěšné byly.

Datový kontext	**Webová stránka**	**Tabulka**	**Sloupec**
Název (kontext) prvku	Page	People	Name
Výsledný identifikátor	Page	Page_People	Page_People_Name

Tabulka 4.1. Tabulka ukazující skládání generovaného identifikátoru na základě kontextu komponenty. Výsledný identifikátor pro Sloupec zobrazující jméno (Name) je složen z kontextu celé webové stránky spolu s kontextem tabulky vypisující informace o osobách.

V sekci 4.1 byly zmíněny `data-*` HTML atributy. Ty slouží pro definici vlastních atributů a uložení hodnot a mohou být využity u jakéhokoliv HTML elementu. Tím pádem, generované identifikátory lze ukládat například do atributu `data-ui`. Dalším problémem je hodnota atributu. Jednou z variant je generovat identifikátory jako náhodné hodnoty

[4] `https://www.w3schools.com/css/`
[5] `https://www.w3schools.com/xml/xpath_intro.asp`

se zajištěním, že budou ve zbytku stránky unikátní. K tomu se nabízí použití čísla UUID (GUID)[6], které představuje 128-bitové číslo a šance duplikace čísla je velmi blízko nule[7]. Nevýhodou UUID je jeho nesrozumitelnost. Tester se z náhodných čísel nedozví nic o kontextu elementu. Jako řešení tohoto problému je vygenerování identifikátoru přímo z kontextu DOTHTML komponenty s využitím názvu vlastnosti použité pro *data-binding* hodnoty, zobrazeného textového řetězce nebo nastaveného datového kontextu na komponentě. Příklad skládání identifikátoru pro jednotlivé komponenty na základě kontextu lze vidět v tabulce 4.1.

> *Těmito dvěma rozhodnutími je možné funkční požadavek č. 1 považovat za splněný.*

4.2.3 Využití programového popisu

Předchozí část o návrhu generátoru se zabývala návrhem, jak splnit první tři funkční požadavky. V této sekci zbývá rozebrat návrh řešení jak dosáhnout splnění čtvrtého funkčního požadavku a jak v testech upotřebit získané třídy `PageObject`.

```
1  public class TestHelper : PageHelperBase
2  {
3      public TextBox Email { get; }
4      public TextBox Password { get; }
5      public Button Přihlásitse { get; }
6      public ValidationSummary Summary { get; }
7
8      public TestHelper(IWebDriver webDriver): base (webDriver)
9      {
10         Email = new TextBox(this, "Email");
11         Password = new TextBox(this, "Password");
12         Přihlásitse = new Button(this, "Přihlásitse");
13         Summary = new ValidationSummary(this, "Summary");
14     }
15 }
```

Výpis 4.6. Ukázka jak může vypadat navrhovaný `PageObject` pro View v ukázce 4.1. Pro generování názvů atributů C# třídy budou využity hodnoty (DOT)HTML atributů, jména použitých vlastností nebo řetězce zobrazené v komponentách. Příkladem je řádek č. 5, v kterém je třídní atribut typu `Button` pojmenován „Přihlásit se“ z důvodu použití hodnoty DOTHTML atributu `Text` ve výpisu 4.1. Třída `TestHelper` obsahuje veškeré reference na DotVVM komponenty s využitím pomocných objektů (např. TextBox).

Výsledkem generátoru bude adresář obsahující všechny vygenerované pomocné třídy a upravené definice View. Následně může tester implementovat testy s aplikováním pomocných tříd nebo jen užít *Selenium WebDriver* z nich dostupný spolu s vygenerovanými identifikátory. Bude tedy jen na něm, jaký přístup k testování si zvolí. Příklad navrženého výsledku třídy `TestHelper` lze vidět v ukázce 4.6 a využití tohoto pomocného objektu je navrhnut ve výpisu 4.7.

[6] `https://en.wikipedia.org/wiki/Universally_unique_identifier`
[7] `http://www.h2database.com/html/advanced.html#uuid`

```
1  [TestMethod]
2  public void TestLoginSuccess()
3  {
4      //Arrange
5      var pageHelper = new TestHelper(driver);
6      //Act
7      pageHelper.Email.SetText("test@test.com");
8      pageHelper.Password.SetText("test");
9      pageHelper.Přihlásitse.Click();
10     //Assert
11     Thread.Sleep(2000);
12     var errors = pageHelper.Summary.GetErrors();
13     Assert.AreEqual(errors.Count, 0);
14 }
```

Výpis 4.7. Ukázka představující využití třídy `TestHelper` z výpisu 4.6. Tato ukázka není přepisem testovací metody z výpisu 4.2, ale představuje stejný testovací scénář, ovšem s využitím třídy `TestHelper` ve výpisu 4.6.

Posledním problémem k vyřešení je zajištění statické kontroly. Jeho vyřešením se nástroj stane důležitý nejenom pro testera, ale i pro vývojáře. A to z toho důvodu, že v případě úpravy View webu můžou nastat takové změny, kterými se testy stanou nefunkční. Pro zjištění nefunkčnosti testů by následně bylo potřeba testy spustit a čekat na jejich výsledek. To v případě GUI testů může být časově náročné. Pokud ale bude generátor spuštěn v době překladu webové aplikace, přegenerováním pomocných tříd budou editovány reference použité v testech a odstraněné komponenty nebude možné v testech použít. Výsledkem pak bude nezkompilovaná webová aplikaci i s testy. Toto chování lze zajistit více způsoby:

1. vytvořením konzolové aplikace, která se spustí po dokončení sestavení aplikace,
2. vytvořením *post build události (skriptu)*, který zajistí spuštění generátoru,
3. úpravou skriptu zajišťující automatické sestavení aplikace, například MSBuild[8] v nástroji Visual Studio[9].

Ideálním řešením pro výsledný nástroj by bylo zapojení spuštění nástroje po dokončení kompilace projektu webové aplikace a před spuštění kompilace aplikačních GUI testů.

> *Výběrem jakékoliv možnosti z předchozího seznamu bude splněn i poslední funkční požadavek č. 4.*

Za výsledek této kapitoly lze považovat dvě dílčí části — definici požadavků a návrh nástroje zodpovídající k definici požadavků. Požadavky jsou definovány čtyři, přičemž dva se zaměřují na generování programových popisů a dva na pomocné objekty v programových popisech. Výsledkem samotného návrhu bylo navrhnutí mechanismů a objektů, které definované požadavky splňují, což se úspěšně podařilo. V další části následuje převedení navržených principů do reality pomocí konkrétních technologií a programovacích technik.

[8] `https://docs.microsoft.com/cs-cz/visualstudio/msbuild/msbuild`
[9] `https://visualstudio.microsoft.com`

Kapitola 5

Implementace

Tato kapitola pojednává o implementační fázi vývoje nástroje pro usnadnění testování GUI webových aplikací. V první části kapitoly se nachází popis technologií použitých při implementaci pro dosažení navržené funkcionality z předchozí kapitoly. Součásti popisu technologií je i popis struktury nástroje. Dále je popsána implementace generátoru. Ten se stará o vytvoření programových popisů, vytvoření nebo získání unikátních selektorů elementů ve View a připravení změn, které je nutné provést v zdrojovém DOTHTML souboru. Poslední podsekcí je popis spuštění generátoru a jeho napojení na webovou aplikaci. V této podsekci také probírám popis a implementační detaily zajišťující kontrolu v době kompilace.

V druhé části kapitoly je popsána implementace a využití pomocných objektů, které generátor využívá v programových popisech. Spolu s tím, je rozebráno vyhledávání elementů ve View, řešení kolize selektorů a zohlednění uživatelových úprav selektorů. Na to navazuje sekce popisující možné uživatelské rozšíření těchto pomocných objektů. Tedy připravenost nástroje na využití testerem vytvořených pomocných objektů.

Použité nástroje a technologie

Před začátkem samotné implementace bylo nutné prvně rozhodnout, s jakým frameworkem a technologiemi bude nástroj pracovat. Z důvodu, že nástroj má fungovat jako rozšíření open-source frameworku DotVVM, vývojovou platformou se stala aktuální verze .NET Core frameworku spolu s jazykem C#.

Nástroj bude dostupný jako otevřený software. Proto framework pro testování GUI, který bude pracovat v pozadí, bylo nutné zvolit také některý z nekomerčních. Mezi nimi se stále největší popularitě těší framework Selenium, který je navíc velmi jednoduché a rychlé použít ve webové aplikaci. Ať už tvořené pomocí .NET Framework nebo .NET Core. Všechny zvolené technologie a nástroje jsou tyto:

- .NET Core 2.2 a C#,
- DotVVM,
- Selenium WebDriver,
- Visual Studio 2019[1] jako hlavní implementační IDE,[2]

[1] `https://visualstudio.microsoft.com`
[2] IDE — Vývojové prostředí

- Git[3] a GitHub[4] jako verzovací systém,
- MSTest Framework[5] jako *unit testing* framework,
- Google Chrome jako webový prohlížeč pro testování výstupů,
- Newtonsoft.Json knihovnu dostupnou z NuGet[6] galerie pro práci s formátem JSON.

Celý nástroj byl vyvíjen jako jediné Solution obsahující několik projektů. Pod pojmem projekt v rámci Solution je potřeba si představit samostatný prvek, jehož výstupem je většinou knihovna nebo aplikace. Rozdělení jednoho Solution do více projektů také přidává na přehlednosti a udržovatelnosti kódu. Osobně jsem si rozdělil práci do těchto projektů:

DotVVM.Framework.Tools.SeleniumGenerator je projekt fungující jako konzolová aplikace (více popsán v podkapitole 5.1). Obsahuje veškerou logiku potřebnou pro přečtení vstupů aplikace, průchod abstraktního stromu View, generování názvů, programových popisů atd.

DotVVM.Framework.Testing.SeleniumGenerator je projekt, který obsahuje datové struktury potřebné v předchozím projektu. Dále pak funguje jako projekt, který bude referencovaný z webové aplikace pro napojení se na generátor.

DotVVM.Framework.Testing.SeleniumHelpers projekt obsahuje pomocné objekty pro práci s DotVVM uživatelskými prvky UI (TextBox, CheckBox, Button atd.). Také je zde logika pro vyhledání odpovídajících elementů ve stránce pomocí frameworku Selenium podle selektorů z programového popisu. Tento projekt více rozeberu jako součást podkapitoly 5.2.

DotVVM.Testing.SeleniumGenerator.Tests je projekt sloužící jako testovací. Obsahuje testy pro kontrolu funkčnosti generátoru. Popis testování nástroje je probrán v samostatné kapitole 6.

SampleApp1 představuje DotVVM webovou aplikaci. Tento projekt slouží pro demonstraci funkčnosti vygenerovaných programových popisů. Tento způsob kontroly a demonstrace funkčnosti je také popsán v kapitole 6.

Ostatní projekty v rámci Solution jsou vložené projekty z frameworku DotVVM, například (`DotVVM.Framework` a `DotVVM.Framework.Hosting.AspNetCore`). Tyto projekty jsou do Solution vloženy hlavně kvůli lepší možnostem ladění nástroje. Projekt `DotVVM.Utils.ConfigurationHost` umožňuje získat konfiguraci DotVVM aplikace ze souboru dll a `DotVVM.Utils.ProjectService` slouží pro nalezení metadat o .NET projektu — zejména seznam referencovaných knihoven. Další projekt z DotVVM je `DotVVM.CommandLine`, který funguje jako konzolová aplikace. Zabývá se analyzováním vstupních parametrů a manipulací se známými příkazy. V tomto řešení je přes ní spouštěna konzolová aplikace generátoru. Generátor lze ovšem se správně zadanými argumenty programu spustit i bez `DotVVM.CommandLine` aplikace (více v podsekci 5.1.2). Projekt `DotVVM.CommandLine.Core` pak obsahuje objekty, s kterými pracuje jak moje aplikace, tak `DotVVM.CommandLine`.

[3] `https://git-scm.com`
[4] `https://github.com`
[5] `https://docs.microsoft.com/cs-cz/dotnet/core/testing/unit-testing-with-mstest`
[6] `https://www.nuget.org`

5.1 Generátor programových popisů

Hlavní složka celého nástroje. Pominu-li datové struktury, představuje jeden projekt v celém řešení, ale obsahuje veškerou logiku potřebnou pro generování programových popisů. Ke své funkčnosti potřebuje reference na projekty `DotVVM.CommandLine.Core`, `DotVVM.Utils.ConfigurationHost`, `DotVVM.Framework.Testing.SeleniumGenerator` a také `DotVVM.Framework`. Celý generátor funguje jako konzolová aplikace, jenž pracuje v adresáři webové aplikace, pro kterou chce tester vytvořit programové popisy. Výstupem generátoru jsou modifikované Views a vytvořené programové popisy pro všechny webové stránky a uživatelské komponenty ve webové aplikaci nebo jeden programový popis pro konkrétní View.

Samotná logika zodpovídající za generování programového popisu jednoho DOTHTML dokumentu se nachází v třídě `SeleniumPageObjectGenerator`, přesněji v její veřejné metodě `ProcessMarkupFile()`. Tato metoda funguje jako zapouzdřující všech nutných kroků k úspěšnému vygenerování programového popisu. Tyto kroky jsou následující:

1. zjistit abstraktní strom DOTHTML dokumentu
2. zjistit již využité selektory
3. spustit stejný proces generování pro *master pages*
4. spojit využité selektory se selektory vygenerovanými pro *master pages*
5. vytvořit programový popis DOTHTML dokumentu
6. spojit výsledky s výsledky z *master pages*
7. vygenerovat C# třídu *PageObject* pro programový popis
8. upravit DOTHTML dokument vygenerovanými selektory

Součástí třídy `SeleniumPageObjectGenerator` je metoda `ResolveControlTree()` vracející získaný abstraktní strom DOTHTML dokumentu. Tato metoda je spuštěna jak pro jednotlivé DOTHTML stránky, tak i pro zjištění datových typů *code behind* tříd u *Markup* komponent. Z důvodu, že některé *Markup* komponenty můžou být použity v jednom DOTHTML dokumentu několikrát, výsledky, které metoda vrací se ukládají do mezipaměti ve formě slovníku `ConcurrentDictionary<string, IAbstractTreeRoot>` s klíčem v podobě textového řetězce cesty k souboru.

5.1.1 Programový popis

Programový popis představuje v kapitole 4 navržený `PageObject`. Je to pomocná třída, kterou generátor generuje pomocí Roslyn frameworku pro každé webové View, uživatelské komponenty, případně pro složitější DotVVM uživatelské prvky. Programový popis je zamýšlen a jeho generování implementováno tak, aby odpovídal vzoru Page Object[7]. Vytvořené C# třídy se následně generují přímo do referencovaného testovacího projektu, kde je tester může využít.

Každý `PageObject` obsahuje vlastnosti odpovídající jednotlivým prvků UI ve webové stránce. Tyto vlastnosti jsou implementovány podle návrhového vzoru Proxy [9], který

[7] `https://martinfowler.com/bliki/PageObject.html`

```
1  public class TestPagePageObject : SeleniumHelperBase
2  {
3      public RepeaterProxy<UsersRepeaterHelper> Users { get; }
4      public ButtonProxy Refresh { get; }
5
6      public TestPagePageObject(OpenQA.Selenium.IWebDriver webDriver,
7          SeleniumHelperBase parentHelper = null, PathSelector
           ↪ parentSelector = null)
8          : base (webDriver, parentHelper, parentSelector)
9      {
10         Users = new RepeaterProxy<UsersRepeaterHelper>(this, new
           ↪ PathSelector { UiName = "Users", Parent = parentSelector });
11         Refresh = new ButtonProxy(this, new PathSelector { UiName =
           ↪ "Refresh", Parent = parentSelector });
12     }
13     public class UsersRepeaterHelper : SeleniumHelperBase
14     {
15         public LiteralProxy Name { get; }
16
17         public UsersRepeaterHelper(OpenQA.Selenium.IWebDriver webDriver,
18             SeleniumHelperBase parentHelper = null, PathSelector
               ↪ parentSelector = null)
19             : base (webDriver, parentHelper, parentSelector)
20         {
21             Name = new LiteralProxy(this, new PathSelector { UiName =
               ↪ "Name", Parent = parentSelector });
22         }
23     }
24 }
```

Výpis 5.1. Ukázka programového popisu (PageObject) s dalším vnořeným. Na řádku č. 3 a 4 jsou vlastnosti nadřazeného programového popisu. Od řádku č. 14 již následuje definice programového popisu `UsersRepeaterPageObject` pro vlastnost `Users` (řádek č. 3).

spadá do skupiny vzorů od Gang of Four[8]. V mém řešení má každý typ komponenty svoji dedikovanou třídu Proxy (např. `ButtonProxy`), která zaobaluje Selenium `IWebElement` a zapouzdřuje logiku hledání odpovídajícího elementu ve stránce. Více o jednotlivých `Proxy` a jejich využití je popsáno v sekci 5.2.

V případě, že webová stránka obsahuje složitější prvky GUI (`GridView, Repeater`), je součástí programového popisu definice dalšího. Tyto komponenty následně mají speciální typ `Proxy`, která jako generický parametr přebírá `PageObject` obsahu dané komponenty. Toto zanoření jednotlivých programových popisů není nijak omezeno. Ukázku vnoření programových popisů lze vidět ve výpisu 5.1.

[8]Gang of Four — skupina čtyř autorů knihy *Design Patterns: Elements of Reusable Object-Oriented Software*. Tato kniha se zabývá výhodami a nevýhodami objektově orientované programování a zároveň popisuje 23 návrhových vzorů.

Tvorba programového popisu musí brát v úvahu i prvky GUI obsažené v *master pages* (popsány v kapitole o DotVVM v podsekci 3.2.1). Zda je stránka vložena v *master page* lze zjistit pomocí direktivy `@masterPage`. Informaci o *master page* direktivě je možné zjistit ze získaného abstraktního stromu. Pokud se aktuální stránka nachází v *master page*, vygeneruje se samostatná definice programového popisu spolu s požadovanými úpravami DOTHTML souboru. Z důvodu, že *master page* může být také vložena do jiné *master page* atd., je celé generování definic programových popisů pro *master pages* implementováno pomocí rekurze. Výstupem generování programových popisů *master pages* je kolekce definic, které jsou následně zkombinované s definicí programového popisu pro aktuální View.

Zjištění známých prvků GUI

Aby byl generátor schopný vygenerovat výsledný `PageObject`, je nutné definovat komponenty, které zná a umí s nimi pracovat. V rámci implementace generátor podporuje prvky GUI, které jsou dostupné přímo z frameworku DotVVM. Jejich podpora je definována pomocí tzv. *dílčích generátorů*. Současně s tím, generátor podporuje i vlastní implementace dílčích generátorů (více popsáno v podsekci 5.3). Podpora jednotlivých komponent je specifikována ve třídě `SeleniumPageObjectVisitor` pomocí slovníku pro jednotlivé typy prvků UI a jejich odpovídající generátory. Každý dílčí generátor specifikuje způsob, jakým se s konkrétní komponentou má pracovat, jaké informace se pro ni mají vygenerovat do programového popisu atd. (konkrétnější využití je popsáno dále v sekci 5.2 a ukázka ve výpisu 5.2).

Samotná třída `SeleniumPageObjectVisitor` je instanciována již v době vytvoření instance `SeleniumPageObjectGenerator`. Je to z toho důvodu, že tento *Visitor* obsahuje referenci na objekt generátoru, kvůli podpoře spuštění vlastního procesu zjištění abstraktního stromu. Po vytvoření instance třídy `SeleniumPageObjectVisitor` je důležitou části najít veškeré dílčí generátory implementující rozhraní `ISeleniumGenerator`. Jejich hledání je provedeno nad sestaveními (*assemblies*) definovanými v třídě `SeleniumGeneratorOptions`. K nalezeným generátorům jsou následně přidány vlastní dílčí generátory (jejich registrace je popsána v sekci 5.3). Výsledek je zapsán do slovníku zmíněný v předchozím odstavci.

Vstupem generátoru je kořen abstraktního stromu (`IAbstractTreeRoot`) aktuálního View, s kterým generátor pracuje. Samotný abstraktní strom je získán způsobem popsaným v kapitole o DotVVM v podsekci 3.2.3. Průchod stromem je implementován pomocí třídy `SeleniumPageObjectVisitor`, která rozšiřuje abstraktní třídu `ResolvedControlTreeVisitor` dostupnou ve frameworku DotVVM. Jak je z názvu patrné, tyto třídy využívají návrhového vzoru *Visitor*, který je lehce popsán níže (podčást 5.1.1).

Jádro procházení abstraktního stromu se nachází v implementaci přepsané metody `VisitControl(ResolvedControl control)` v třídě `SeleniumPageObjectVisitor`. Ta je využita při navštívení definice komponenty z abstraktního stromu. Implementace obsahuje řešení sledování aktuálního datového kontextu pro generování názvů selektorů, tak jak bylo popsáno v návrhu v podkapitole 4.2.2. Následně předá získané informace o komponentě a aktuálním stavu programového popisu dílčímu generátoru uživatelské komponenty, který se stará o definici generovaných vlastností.

Návrhový vzor Visitor

Vzor Visitor je jeden z 23 návrhových vzorů pro objektově orientované programování popsaný v knize *Návrh programů pomocí vzorů* od skupiny autorů známých jako „Gang of Four“. Tento vzor je ideální k využití pro průchod hierarchických struktur. *Visitor* patří

mezi takzvané vzory chování (behavioral patterns) a umožňuje rozšířit funkcionalitu množiny tříd, aniž by bylo nutné modifikovat jejich implementaci [9]. Abych této možnosti docílil, musí navštěvované třídy být schopné přijmout „návštěvníka“. Návštěvník následně musí umět navštívit a provést požadovanou operaci nad třídou, kterou navštěvuje.

Získání unikátního selektoru

Generátor využívá několik cest, jak získat název, který použije jako identifikátor a zároveň jako název vlastnosti v C# programovém popisu. Generátor všechny cesty kombinuje a využívá aktuálně nejvýhodnější. Mezi tyto způsoby patří:

- **získání již zadaného nebo vygenerovaného selektoru,**

  Tato možnost je preferována. Selektor může existovat již z předchozího běhu generátoru a je žádoucí tyto selektory neměnit, jak z důvodu výkonu (muselo by se znovu zasahovat do DOTHTML souboru), tak z důvodu přívětivosti pro uživatele. Selektor ovšem nemusí být jen generovaný, ale uživatel může zvolit vlastní. Ten se následně použije i pro název vlastnosti v programovém popisu.

- **využití některé vlastnosti prvku UI,**

  Jak bylo v kapitole 3 vysvětleno, každý element má svoje definované vlastnosti. Z nich lze vytvořit unikátní identifikátor jen za využití jejich hodnot. Například hodnota vlastnosti `Text` u komponenty `dot:Button` může obsahovat řetězec „Click me“, jenž bude testerovi napovídat, co komponenta dělá a zároveň může posloužit jako unikátní selektor.

- **využití obsahu prvku UI,**

  U některých elementů může být rozumné získat hodnotu pro selektor přímo z jeho obsahu. Zda je povolené tuto možnost využít, je definováno pro každý element zvlášť v jeho dílčím generátoru.

- **využití jména uživatelské komponenty.**

  Tato varianta přichází v úvahu jako poslední, a to u prvků, jež nemají žádné vlastnosti nebo je nelze využít, protože by nebyly unikátní. Využívá se například pro vývojářem definované *Markup* komponenty.

Aby vlastnosti pro generování názvu nebyly vybírány čistě náhodně, existuje pro každou DotVVM komponentu odpovídající C# třída (dílčí generátor), jež specifikuje, které vlastnosti se mají při volbě názvu (selektoru) vzít v potaz (např. pro `Button` C# třída `ButtonControlGenerator`). Tato třída také specifikuje, zda lze využít obsah elementu ke generování identifikátoru. Dále i k specifikaci, který pomocný objekt přísluší dané komponentě. Implementaci této třídy lze vidět v ukázce 5.2.

I přes všechny vyjmenované způsoby není zaručeno, že získaný selektor bude unikátní. Pokud by tento případ nastal, připojí se na konec identifikátoru inkrementující se číslo. Další situace, kterou bylo potřeba řešit, bylo přidání komponenty ve View, které by se mohl vygenerovat stejný identifikátor jako jiné. Problém je řešen pomocí průchodu získaného stromu View a uložení všech již použitých selektorů, ještě před začátkem samotného generování. Implementace průchodu spočívala v rozšíření třídy `ResolvedControlTreeVisitor`, kterou poskytuje DotVVM, vlastní třídou `SeleniumSelectorFinderVisitor` a přepsanou implementací metody `VisitControl()`. V této metodě se pro každou nalezenou komponentu

```
1  public class ButtonControlGenerator : SeleniumGenerator<Button>
2  {
3      private static readonly DotvvmProperty[] nameProperties
4          = { ButtonBase.TextProperty, ButtonBase.ClickProperty };
5
6      public override DotvvmProperty[] NameProperties => nameProperties;
7      public override bool CanUseControlContentForName => true;
8
9      protected override void AddDeclarationsCore(
10         PageObjectDefinition pageObject,
11         SeleniumGeneratorContext context)
12     {
13         const string type = "ButtonProxy";
14         AddPageObjectProperties(pageObject, context, type);
15     }
16 }
```

Výpis 5.2. Ukázka třídy `ButtonControlGenerator` definující vlastnosti elementu `<dot:Button>` použitelné pro získání selektoru. V tomto případě vlastnosti `Text` a `Click`. Pravdivostní proměnná na řádku č. 7 definuje, že je možné zkusit použít obsah prvku UI pro vygenerování selektoru.

ve View uloží nalezený selektor do setu `HashSet<string>`. Ten je následně předán jako parametr do generátoru.

V rámci generování názvu je nutné pokrýt pojmenování, jak selektoru, tak názvu vlastnosti v programovém popisu. Každý z názvů se vyskytuje v jiném prostředí (C#, resp. HTML), kde panují jiné zvyky ohledně pojmenovávání identifikátoru apod. Použité řešení je vysvětleno v ukázce 5.3.

Po získání jména probíhá ještě mírná normalizace řetězce, aby bylo zajištěno, že řetězec obsahuje jen znaky, které je možné použít v jazyce C# jako název vlastnosti. Tyto úpravy se musejí speciálně dělat při získání selektoru zadaného uživatelem, který nemusí odpovídat pravidlům syntaxe jazyka C#.

Generování vlastností v programového popisu

S ohledem na to, že generátor pracuje s informacemi, které jsou dostupné v době kompilace, je nutné i výsledné programové popisy generovat v době kompilace. Toho je docíleno pomocí frameworku Roslyn (více popsáno v části 5.1.1). Pro každou DotVVM komponentu je pomocí dílčího generátoru (ukázka 5.2) definováno, jaký výstup se do programového popisu vygeneruje. Největší rozdíl v implementaci jednotlivých generátorů je v tom, zda `Proxy` bude využívat generický typ definující vnořený `PageObject`. Podle toho je do definice programového popisu (třída `PageObjectDefinition`) zaregistrována vlastnost a výraz pro konstruktor pro aktuální `PageObject` jako syntaktická struktura. V rámci toho je použit vytvořený selektor, datový typ a název vlastnosti.

Specifický případ je komponenta `GridViewTemplateColumn`. Ta představuje tabulkový sloupec v komponentě `GridView`, který může obsahovat složitější HTML strukturu. S ohledem na lepší orientaci testera v programovém popisu se v tomto případě vyplatí pro šablonu vygenerovat celý `PageObject` a zároveň na něj nemít referenci pomocí objektu `Proxy`. Ta

```
1 <dot:Button Text="Calculate It" Click="{command: CalculateIt()}"
2             UITests.Name="calculate-it" />
3
4 <dot:Button Text="Calculate It" Click="{command: CalculateIt()}"
5             UITests.Name="CalculateIt" />
6
7 public ButtonProxy CalculateIt { get; }
```

Výpis 5.3. Ukázka výsledků dvou implementací generování názvu selektoru a vlastnosti. Na řádku č. 2 lze vidět vygenerovaný selektor v podobě, která zapadá do HTML a na řádku č. 7 vlastnost, která odpovídá prvku `<dot:Button>`. Problém ovšem nastává při opakovaném spuštění generátoru. Získaný řetězec (zde „calculate-it“) není možné vždy bezchybně naparsovat, aby bylo možné obnovit název vlastnosti v programovém popisu a testerovi se neměnil pod rukami. Z tohoto důvodu se ve výsledné implementaci generuje selektor ve formátu jako na řádku č. 5. V tomto tvaru se následně využívá i v programovém popisu.

by neumožnila intuitivní přistoupení ke konkrétnímu šablonovému sloupci v rodičovském programovém popisu. Nadřazený `PageObject` tedy obsahuje přímo referenci na vnořený `TemplatePageObject`.

.NET Compiler Platform — Roslyn[9]

Jde o projekt od společnosti Microsoft, který má vývojářům zpřístupnit informace, získané během procesu kompilace. Data vzniklá v době kompilace obsahují spoustu užitečných informací o kódu, které se ovšem hned po dokončení zahodí. S těmito daty se dá ovšem provádět statická analýza kódu nebo je mohou využít IDE k nabízení různých funkcí pro zvýšení efektivity programátora.

V době kompilace se kompilátor nachází v několika fázích, ke kterým je možné pomocí Roslynu získat přístup. V mém projektu využiji operací použitelných nad syntaktickým stromem. S tím lze provést tři typy operací. Je možné procházet již vytvořené syntaktické struktury, měnit je, a také lze vytvářet vlastní syntaktické stromy. Pro práci se syntaktickými strukturami je možné využít dvou tříd. Zaprvé `SyntaxGenerator`, kterému není nutné specifikovat, s jakým jazykem pracuje, nýbrž si ho zjistí z objektu `Project` a následně generuje kód v jazyce používaném v konkrétním projektu. Druhou možností je použít statickou třídu `SyntaxFactory`[10] z jmenného prostoru `Microsoft.CodeAnalysis.CSharp` a využívat její statické metody pro tvorbu syntaktických struktur. Tuto možnost jsem následně zvolil pro mé účely generování programových popisů. Ukázku generování syntaktické struktury pomocí `SyntaxFactory` je možné vidět na výpisu 5.4.

Úprava DOTHTML souborů

V rámci generování názvů selektorů, syntaktických struktur pro `PageObject` apod. je také potřeba připravit úpravy samotných DOTHTML souborů. V mém řešení zapisuji změny přímo do DOTHTML souboru pomocí třídy `File` z jmenného prostoru `System.IO`. Pro

[9] `https://github.com/dotnet/roslyn/wiki`
[10] `https://docs.microsoft.com/en-us/dotnet/api/microsoft.codeanalysis.csharp.syntaxfactory`

```
protected StatementSyntax GenerateInitializerForTemplate(string
↪ propertyName, string typeName)
{
  return SyntaxFactory.ExpressionStatement(
    SyntaxFactory.AssignmentExpression(SimpleAssignmentExpression,
      SyntaxFactory.IdentifierName(propertyName),
      SyntaxFactory.ObjectCreationExpression(
          SyntaxFactory.ParseTypeName(typeName))
          .WithArgumentList(SyntaxFactory.ArgumentList(
            SyntaxFactory.SeparatedList(new[] {
              SyntaxFactory.Argument(
                SyntaxFactory.IdentifierName("webDriver")),
              SyntaxFactory.Argument(SyntaxFactory.ThisExpression()),
              SyntaxFactory.Argument(
                SyntaxFactory.IdentifierName("parentSelector"))
          })))
  ));
}
```

Výpis 5.4. Ukázka vytvoření syntaktické struktury pro výraz specifikující inicializaci vlastnosti programového popisu pro prvek `GridViewTemplateColumn`. Jako vstupní parametr metody vstupuje jméno vlastnosti, pro kterou tvoří inicializační výraz, a název datového typu konkrétní třídy `PageObject`.

každý nově vygenerovaný selektor jsou vytvořena data o modifikaci, která obsahují zapisovaný řetězec a pozici v souboru. Informace o umístění, kam je potřeba atribut se selektorem zapsat, je získávána z vlastnosti `DothtmlNode` ve třídě `ResolvedControl`. Přístup k této třídě umožňuje `ResolvedControlTreeVisitor` prostřednictvím metody `VisitControl()`. Vlastnost `DothtmlNode` popisuje, jak vypadá prvek v DOTHTML souboru a v rámci výčtu jednotlivých tokenů obsahuje informaci o pozici uzavíracího tagu (`>` nebo `\>`).

Samotné zapisování do souboru probíhá po vygenerování programového popisu. Veškeré modifikace jsou seřazeny sestupně podle pozice. Sestupně z toho důvodu, aby se při zapisování jednotlivých modifikací neměnila pozice zápisu kontrolek v dokumentu vzhledem k pozicím zjištěným při průchodu stránky generátorem.

Časté zapisování do DOTHTML souborů by bylo časově náročné. Ovšem nástroj generuje nové selektory jedině v případě, že pro daný prvek uživatelského rozhraní ještě neexistuje selektor z předchozího běhu nástroje nebo ho uživatel sám nespecifikoval. Tím pádem je počet přístupů k souborům a zápisu modifikací velmi rapidně snížen.

5.1.2 Spuštění generátoru a napojení na webovou aplikaci

Nástroj je možné spustit více způsoby. Především pak prostřednictvím projektu `DotVVM.CommandLine`, do kterého v rámci implementace výsledného nástroje byla přidána podpora jeho spuštění. Výhoda spuštění přes `DotVVM.CommandLine` projekt spočívá ve vyřešení problémů spojených s konfigurací metadat a cest k jednotlivým projektům. Tento projekt funguje jako rozšíření .NET Core příkazové řádky.

V rámci spuštění generátoru bylo nutné zjistit následující data — cestu k DotVVM aplikaci, .NET framework, nad kterým je aplikace postavena, a také najít cestu k samotné apli-

kaci generátoru. Většina nalezených informací se ukládá do třídy `DotvvmProjectMetadata` a předána jako aplikační parametr do nástroje. Nástroj je možné spustit pomocí příkazu `gen uitest`, případně pomocí jeho zkrácené verze `gut`. Módy spuštění tohoto příkazu přes DotVVM příkazovou řádku jsou následující:

- bez parametrů (spustí generátor pro všechny komponenty a *DOTHTML* soubory registrované v DotVVM konfiguraci,
- s jedním a více parametrů ve formě relativní cesty ke konkrétním DOTHTML dokumentům nebo komponentám.

V případě spuštění příkazové řádky s jedním z těchto příkazů, je zavolán *handler* `GenerateUiTestStubCommandHandler` a poté `SeleniumGeneratorLauncher`, což jsou třídy zprostředkovávající start samotné aplikace. Výsledkem těchto kroků je nalezení dat nutných pro spuštění, jejich serializace do JSON, připravení nového aplikačního procesu a jeho spuštění jako konzolové aplikace projektu `DotVVM.Framework.Tools.SeleniumGenerator`.

Druhým způsobem, jak zprovoznit implementovaný nástroj, je s pomocí IDE Visual Studio. Zde je nutné nastavit projekt `DotVVM.Framework.Tools.SeleniumGenerator` jako startovací projekt. Pro úspěšné spuštění projektu je dále nutné nastavit aplikační parametry v podobě DotVVM metadat v textovém formátu JSON (parametr `--json ""`). Tato metoda se ovšem více hodí pro spouštění v rámci vývoje. a to z důvodu, že projekt je nastartován jako jeden proces, což urychluje možnosti ladění programu.

Výsledkem zapnutí nástroje nad webovou aplikací je vygenerování programových popisů do projektu s UI testy a úprava DOTHTML dokumentů. Programové popisy jsou vkládány do složky *PageObjects*, kde je dodržena struktura podsložek stejná jako u samotných DOTHTML dokumentů v aplikaci. Programové popisy pro *Markup* komponenty budou k nalezení v podsložce *Controls*.

Nástroj jako součást DotVVM

I když v této práci je výsledný nástroj popsán především jako individuální část, chtěným výsledkem bylo nástroj vydat jako součást open-source frameworku DotVVM. Toho se dosáhlo již v době publikace této práce. A to právě formou přidání podpory spuštění nástroje pomocí příkazové řádky a zapojení generátoru jako součásti frameworku DotVVM. Projekt příkazové řádky v DotVVM se instaluje jako globální nástroj, a to pomocí následujícího příkazu:

```
$ dotnet tool install dotvvm.commandline -g --version 2.3.0-preview01
    -41917-selenium-generator
```

Prerekvizitou úspěšného spuštění příkazu je nainstalovaný .NET Core framework. Specifikovaná verze byla aktuální v době publikace práce. V případě instalace doporučuji zaměnit za aktuální verzi. Po dokončení instalace lze využívat příkazy `gen uitest` a `gut` stejným stylem, jak bylo popsáno výše. Jak využívat staticky typové kontroly v době kompilace je popsáno v sekci 5.1.2. Více informací, jak využívat DotVVM příkazové řádky, spustit nástroj nad nějakou webovou aplikací je popsáno na přiloženém CD (příloha A).

Vytvoření testovacího projektu

Pro správné fungování nástroje musí existovat projekt, do kterého může generátor vygenerovat jednotlivé programové popisy. Při spuštění nástroje jsou zkontrolována metadata

webové aplikace, zda obsahují relativní cestu k složce obsahující .NET C# knihovnu s testovacím projektem. Jestliže je cesta nalezena, uživatel už nemusí nijak interagovat s nástrojem. Nástroj ještě zkontroluje, zda v testovacím projektu existuje složka *PageObjects* (pokud ne, je vytvořena). Poté už začíná samotný proces generování — začne se číst obsah webové aplikace a generovat programové popisy.

V opačném případě je uživatel dotázán na zadání cesty k testovacímu projektu, přičemž uživatel dostane připravenou výchozí cestu. Ta směřuje do stejné složky, kde se nachází kořenový adresář webová aplikace. Následuje kontrola, zda zadaný adresář existuje. Pokud ne, je nutné vygenerovat nový .NET C# projekt. Součástí každého projektu musí být soubor s příponou *.csproj*, který obsahuje informace o souborech v projektu, sestavení, nastavení projektu, verzi projektu apod. Struktura tohoto souboru je získána pomocí *T4 textové šablony*[11]. Ta spočívá v mixu textových bloků a logiky v jazyce C#. Výsledkem šablony je vygenerovaný textový soubor, jež může být jakéhokoliv typu — webová stránka, soubor se zdroji nebo soubor se zdrojovým kódem v jakémkoliv jazyce [27].

Nástrojem připravená šablona a z ní vygenerovaný *.csproj* obsahuje informace o cíleném frameworku, projektovou referenci na `DotVVM.Framework.Testing.SeleniumHelpers` a webovou aplikaci. Dále pak balíčkové reference pro využití `MSTest` frameworku a adaptéru spolu s referencí na balíček `Selenium WebDriver`. Výsledkem je textový řetězec s obsahem souboru *.csproj*, který je vytvořen ve složce s testovacím projektem. Po vytvoření testovacího projektu a ukončení generování se občas nemusí správně obnovit balíčky v projektu. Pro opravu závislostí v projektu je následně nutné pustit příkaz `dotnet restore` ve složce s testovacím projektem.

Kontrola při kompilaci

Jedním z definovaných požadavků na implementaci bylo zajištění staticky typové kontroly testů v době překladu UI testů a webové aplikace. V návrhu jsem představil několik nápadů, jak by se toho dalo docílit, přičemž rozhodnutí padlo až v implementační fázi. Výsledkem je částečná kombinace prvního a třetího námětu (návrhy v podsekci 4.2.3). Výsledný nástroj byl celý vyvinut jako konzolová aplikace. Je tím pádem možné jej pustit kdykoliv, nejen během překladu aplikace. Abych ovšem dosáhl i kontroly během samotné kompilace, využil jsem nástroje MSBuild[12].

MSBuild, neboli celým názvem Microsoft Build Engine, je platforma pro sestavování aplikací. Pomocí XML schématu definující projektový soubor kontroluje, jak řídit kompilaci a sestavit software [20]. Nejlepší ukázkou, kde je nástroj MSBuild využit, představuje IDE Visual Studio, které ho využívá pro sestavování projektů v rámci Solution. MSBuild ovšem není závislý jen na programu Visual Studio. Lze ho využít pro sestavení projektů, i když není Visual Studio nainstalováno, a to pomocí příkazové řádky nebo když je potřeba projekt sestavit pomocí 64-bitové verze MSBuild [20].

Mezi další příklady využití patří potřeba modifikovat, přesněji specifikovat, pořadí, v jakém se má Solution sestavit. Přesně této vlastnosti jsem potřeboval využít v rámci zajištění kontroly funkčních testů během kompilace.

Výsledné řešení se skládá ze dvou částí. První je nutná závislost testovacího projektu na webové aplikace. Jestliže již tester má projekt vytvořen, je nutné tuto závislost do projektu přidat. V realitě je tato závislost velmi běžný postup, protože v rámci testů chceme využívat

[11] `https://docs.microsoft.com/en-us/visualstudio/modeling/code-generation-and-t4-text-templates`

[12] `https://docs.microsoft.com/cs-cz/visualstudio/msbuild/msbuild`

```
1 <Target Name="SeleniumGeneratorTarget" AfterTargets="BeforeBuild">
2     <Exec Command="dotvvm gut"
3           WorkingDirectory="..\\SampleApp1" />
4 </Target>
```

Výpis 5.5. Ukázka `Target` elementu z testovacího projektu, pro který jsou tvořeny programové popisy. Tento `Target` se jménem *SeleniumGeneratorTarget* specifikuje událost, která se má provést před samotným sestavením projektu, čehož je dosaženo pomocí specifikování parametru `AfterTargets` s hodnotou `BeforeBuild`. Účelem této úlohy je spustit příkaz `dotvvm gut` v příkazové řádce s pracovním adresářem směřovaným na umístění webové aplikace.

dostupné možnosti z webové aplikace (například definici směrovacích cest atd.). V případě, že nástroj bude vytvářet testovací projekt v rámci jeho spuštění, T4 šablona pro soubor *.csproj* již obsahuje projektovou referenci na projekt DotVVM webové aplikace.

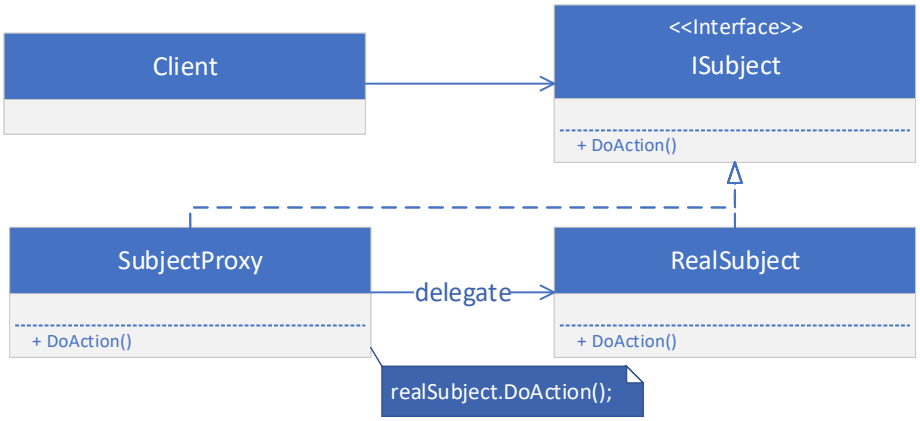
Druhá část řešení spočívá v použití `Target` elementu, který definuje MSBuild. `Target` seskupuje různé úlohy sestavovacího procesu dohromady do pojmenovaného celku. Soubor *.csproj* pak může obsahovat několik elementů `Target`, které rozdělují sestavení aplikace do logických celků [20]. Jak je využit `Target` element v rámci implementovaného nástroje je názorně popsáno ve výpisu 5.5. Důvod specifikace tohoto pořadí spočívá v zajištění spuštění nástroje hned po sestavení webové aplikace, ale před spuštěním kompilace projektu s UI testy. Pokud je tedy provedena nějaká změna, která ovlivní výsledný programový popis, může kompilace testů „spadnout“. To se stane v případě, že programový popis byl již instanciován v nějakém testu a bylo přes něj přistoupeno k vlastnosti reprezentující určitou komponentu. Tato vlastnost ale přegenerováním programového popisu byla odstraněna nebo změněna. Příklad tohoto chování lze nalézt v kapitole 6 o testování funkčnosti nástroje.

S tím, že v rámci běhu nástroje je vytvářen i testovací projekt, umožnilo mi to upravit T4 šablonu pro vytvoření souboru *.csproj* tak, aby úlohu popsanou ve výpisu 5.5 obsahoval již od založení (šablona popsána v bloku 5.1.2). V případě, že testovací projekt již existoval, je nutné tuto úlohu doplnit. Jestliže si bude tester přát vypnout kontrolu v době kompilace, stačí danou úlohu zakomentovat či odstranit.

Projekt `DotVVM.CommandLine` a výsledný nástroj využívají souboru *.dotvvm.json*. Ten obsahuje metadata o webové aplikace a testovací projektu, které je nutné získat od uživatele. Jestliže je první spuštění generátoru provedeno pomocí `Target` elementu v *.csproj* souboru, použijí se výchozí hodnoty těchto dat. Pokud potřebuje tester tyto údaje přesněji specifikovat, musí první spuštění provést pomocí příkazové řádky příkazem `dotvvm gut`. Toto je nutné provést z důvodu, že při založení projektu webové aplikace, není soubor *.dotvvm.json* vytvořen. Vytváří se až v době spuštění rozšíření příkazové řádky. V rámci sestavení aplikace ovšem není možné získat od uživatele data, protože nelze přijímat uživatelův vstup prostřednictvím konzole. Druhou testerovou možností je upravit přímo soubor *.dotvvm.json* v adresáři webové aplikace.

5.2 Pomocné objekty jako Selenium wrappery

V této části se budu zabývat projektem `DotVVM.Framework.Testing.SeleniumHelpers` a jeho využitím v UI testech webové aplikace. Projekt obsahuje pomocné objekty pro programové popisy, tak jak byly navrhnuto v ukázce 4.5 a v kapitole 4. Další součástí této



Obrázek 5.1. Ukázka návrhového vzoru Proxy. V případě, že `Client` bude požadovat volání metody `DoAction()` na rozhraní `ISubject`, vykonání metody bude provedeno pomocí Proxy třídy `SubjectProxy` zastupující třídu `RealSubject`. V rámci vykonání metody v třídě `SubjectProxy`, pak bude spuštěna původní implementace metody `DoAction()` na objektu `RealSubject`.

sekce bude popis využití frameworku Selenium v rámci pomocných objektů a popis implementace vyhledávání elementů a jejich selektorů ve View. Závěrem následuje vysvětlení, jak je nástroj připraven pro možné rozšíření ze strany uživatele.

5.2.1 Pomocné objekty

V předchozí části jsem zmínil pomocné objekty odpovídající návrhovému vzoru Proxy (popsáno níže), které vycházejí z předchozí kapitoly a sekce 4.2. Pomocí nich pak může uživatel pracovat s elementy ve webové stránce.

Aktuálně projekt obsahuje pomocné objekty (Proxy) pro základní komponenty z frameworku DotVVM. Každá `Proxy` dědí z abstraktní třídy `WebElementProxyBase`, která implementuje některé základní operace potřebné u každého elementu (viditelnost a možnost pracovat s uživatelským prvkem ve View) a také metodu `FindElement()` zodpovědnou za nalezení prvku UI podle selektoru ve webové stránce. Vysvětlení, jak tato metoda funguje a její implementace, je probráno v podsekci 5.2.2. Dále třída obsahuje referenci na aktuální `PageObject` a selektor elementu, s kterým tester pracuje. Abstraktní třída `WebElementProxyBase` tedy slouží jako zástupce rozhraní `IWebElement` z frameworku Selenium, které reprezentuje každý HTML element a jsou přes něj prováděny veškeré zajímavé operace nad elementy [26]. Operace nad samotnými elementy jsem čerpal z knihy *Selenium WebDriver Recipes in C#*.

Výjimkou jsou komponenty, s kterými by se při takto zvoleném návrhu pomocného objektu špatně pracovalo. Konkrétně jde o uživatelské *Markup* komponenty a prvky reprezentující sloupce v prvku `GridView`. V těchto případech se v podstatě stále jedná o `Proxy` objekt, ovšem ve formě vlastního programového popisu. Styl, jakým se vlastní `PageObject` generuje, byl představen v ukázce 5.1.

Návrhový vzor Proxy

Návrhový vzor opět spadá mezi 23 návrhových vzorů pro objektově orientované programování představených v knize *Návrh programů pomocí vzorů*. Samotný vzor Proxy se v praxi

```
1  <body>
2     <h1>Stránka s markup control hierarchií</h1>
3     <fieldset>
4        <legend>Control B</legend>
5        <dot:Repeater DataSource="{value: Sections}"
6                      UITests.Name="Sections">
7           <cc:ControlB UITests.Name="ControlB" />
8           <dot:TextBox Text="{value: Language}" UITests.Name="Language" />
9        </dot:Repeater>
10
11       <label>Jméno</label>
12       <dot:TextBox Text="{value: Name}" UITests.Name="Name" />
13    </fieldset>
14 </body>
15
16 // obsah ControlB.dotcontrol definující strukturu kontrolky ControlB
17 <dot:Literal Text="{value: Name}" UITests.Name="Name" />
18 <span DataContext="{value: Counter}">
19    <cc:Counter UITests.Name="Counter_Counter" />
20 </span>
```

Výpis 5.6. Ukázka DOTHTML dokumentu obsahující několik prvků uživatelského rozhraní společně s uživatelskou komponentou `<cc:ControlB>`. Od řádku č. 16 pak následuje obsah souboru ControlB.dotcontrol specifikující vnitřní strukturu tohoto elementu. Lze vidět, že `<ControlB>` element obsahuje použití textového prvku `<dot:Literal>` s vygenerovaným selektorem „Name“ (řádek č. 16). Samotné View pak také obsahuje komponentu, tentokrát `<dot:TextBox>` se selektorem „Name“ (řádek č. 11).

využívá k vytvoření zástupného objektu jinému objektu, který se navenek bude jevit stejně jako původní. Obaluje tedy jeho funkcionalitu a může přidávat vlastní, čímž lze zajistit kontrolu nad tímto objektem. Častým využitím je *logger*, který obalí konkrétní objekt a *loguje* jeho volání, provedené operace apod. Návrh použití návrhového vzoru Proxy je možné vidět na obrázku 5.1.

5.2.2 Vyhledávání selektoru ve View

Vyhledávat elementy ve View jde několika způsoby, což bylo popsáno v podsekci 4.2.2 v rámci návrhu nástroje. První volba při implementaci padla na vyhledávání podle CSS selektorů. V nástroji toto vyhledávání fungovalo dobře do té doby, než jsem začal řešit generování uživatelských *Markup* komponent a tabulkových komponent z DotVVM. Zde jsem narazil na omezení CSS selektorů, a to ve formě vyhledávání předchůdců elementů spolu s indexováním specifických elementů. Byl jsem tedy donucen k využití jazyka *XPath*. Tento jazyk se využívá pro adresaci částí XML dokumentu, tedy i HTML. Pomocí XPath je možné vyhledávat v osách, což byla vlastnost, kterou jsem potřeboval k úspěšnému nalezení předchůdců, sourozenců elementu atd.

Následně bylo potřeba vyřešit problém vyhledávání selektoru ve View, který vznikl oddělením generování selektorů pro uživatelské *Markup* komponenty od generování programového popisu konkrétního DOTHTML dokumentu. V rámci práce s elementy mohlo dojít

k situaci, popsané v ukázce 5.6. V ukázce lze vidět dvě komponenty se stejným selektorem. To by při snaze vyhledat výsledný HTML element pomocí frameworku Selenium dopadlo vrácením dvou elementů, a to je nechtěný výsledek, který by přinesl uživateli komplikace.

Řešením tohoto problému je vlastní implementace metody `FindElement()`, která využívá metod `FindElement(), FindElements()` frameworku Selenium. Vlastní implementace funguje tak, že nalezne veškeré elementy vyhovujícímu danému selektoru. Následně jimi iteruje a pro každý element vyhledá všechny jeho předchůdce, kromě `body` a `html`, kteří obsahují argument `UITests.Name`. Každá `Proxy` má referenci na jeho rodiče, což je možné využít pro porovnání mezi aktuálním předchůdcem a očekávaným rodičovským selektorem. V případě, že se hodnoty neshodují, automaticky se přechází na další ze získaných elementů pro aktuální selektor. Speciálním případem jsou prvky UI reprezentují tabulkové zobrazení (`Repeater, GridView`), u kterých je nutné řešit ještě požadovaný řádek dle indexu. V této situaci je ještě nutné najít pro aktuálního předchůdce všechny „sourozence“ požadovaného řádku a zjistit, zda element v cyklu patří pro aktuálního předchůdce.

5.3 Rozšíření pomocných objektů

Aby použití nástroje nebylo vázáno jen na prvky dostupné z frameworku DotVVM, bylo nutné uživatelovi poskytnout možnost, jak využít vlastní pomocné objekty (*Proxy*) nebo programové popisy. Přičemž existují dvě hlavní varianty, jakými toho může dosáhnout:

- **Rozšiřující metoda[13] nad programovým popisem nebo *Proxy***
- **Vytvoření vlastního pomocného objektu**
  - Rozšíření existujícího pomocného objektu
  - Využití vlastního pomocného objektu

Rozšiřující metody nad pomocným objektem

Pomocí rozšiřujících metod si může tester doimplementovat jakoukoliv logiku proveditelnou přímo nad konkrétním programovým popisem nebo pomocným objektem (*Proxy*). Tímto způsobem není závislý na předpřipravených dílčích generátorech v rámci samotné webové aplikace nebo různých knihoven. Příklad rozšiřující metody nad programovým popisem komponenty lze vidět ve výpisu 5.7. Tento způsob rozšíření objektů bych osobně využil hlavně pro jednodušší metody.

Vytvoření vlastního pomocného objektu

Jak jsem popsal v části 5.1.1, každá komponenta má dílčí generátor, který specifikuje jaký *Proxy* objekt má využít, jaké vlastnosti budou vygenerovány atd. Je tedy na vývojáři jednotlivých kontrolek, aby připravil i jejich generátory. Pro testera to pak znamená, že je závislý na připravených generátorech od vývojáře.

Příkladem bych uvedl vlastní implementaci generátoru `ControlBSeleniumGenerator` z projektu webové aplikace `SampleApp1`. Tento dílčí generátor slouží uživatelské komponentě

[13] rozšiřující metoda — anglicky extension method, je to statická metoda, jejíž první parametr je reference na objekt, který ji volá. Umožňuje přidat třídě instanční metodu, aniž by bylo nutné upravovat samotnou třídu (více na `https://docs.microsoft.com/cs-cz/dotnet/csharp/programming-guide/classes-and-structs/extension-methods`)

```
1 public static void IncrementByValue(
2     this CounterPageObject counter, int value)
3 {
4     for (int i = 0; i < value; i++)
5     {
6         counter.Increment.Click();
7     }
8 }
```

Výpis 5.7. Příklad implementace rozšiřující metody nad objektem `CounterPageObject`. Metoda `IncrementByValue` spočívá v provedení kliknutí na tlačítko *Increment* tolikrát, kolikrát definuje celočíselná proměnná `value`.

`ControlB`, jejíž definici je možné nalézt ve složce *Controls* ve stejném projektu. Ukázka implementace je ve výpisu 5.8. Při implementaci je potřeba brát v potaz několik podmínek. Konkrétní třída musí dědit z třídy `SeleniumGenerator`, jemuž musí předat jako generický parametr datový typ komponenty pro níž je dílčí generátor implementován. Z toho vyplývá, že uživatelská komponenta musí mít definovanou *code-behind* třídu a v rámci DOTHTML specifikovanou direktivu `@baseType` (více popsáno v kapitole 3). Pokud tento typ nebude existovat nebude možné si vytvořit vlastní dílčí generátor a použije se výchozí generátor pro uživatelské *Markup* komponenty.

Tyto generátory obsahuje samotná webová aplikace, jednotlivé vlastní *Proxy* objekty a programové popisy se ovšem musí nacházet v testovacím projektu. A to zaprvé z důvodu, že oba typy se budou využívat jenom v testech, čímž nedává smysl mít tyto typy přímo ve webové aplikaci. Tester tedy musí dodržet jmennou konvenci, která je specifikována ve vlastním generátoru. A zadruhé kvůli nechtěné referenci projektu s *Proxy* objekty `DotVVM.Framework.Testing.SeleniumHelpers`, kterou ale testovací projekt má.

V rámci vytvoření vlastního programového popisu nebo pomocného objektu má tester na výběr, zda využije i výchozí `PageObject` generovaný výsledným nástrojem či nikoliv. Z mého pohledu bude častějším případem „podědění“ výchozí třídy `PageObject` a přidání vlastní implementace. Tester musí ovšem počítat s tím, že se mu výchozí dílčí generátor může kdykoliv přegenerovat.

Aby nástroj mohl využít vlastních dílčích generátorů, musí je vývojář zaregistrovat. Toho dosáhne skrze metodu `ConfigureServices()` ve třídě `DotvvmStartup` s pomocí metody `AddSeleniumGenerator()`. To je rozšiřující metoda nad rozhraním `IDotvvmServiceCollection`, jenž vytváří instanci třídy `SeleniumGeneratorOptions` jako *Singleton* a zaregistruje jí do kolekce služeb DotVVM. Tuto metodu poskytuje projekt `DotVVM.Framework.Testing.SeleniumGenerator`. V rámci jejího volání je možné jako druhý parametr předat akci, v jejímž těle lze přistoupit k třídě `SeleniumGeneratorOptions` a skrz poskytnutou veřejnou metodu `AddCustomGenerator()` zaregistrovat vlastní generátor.

5.4 Shrnutí

V závěru kapitoly bych rád shrnul implementovanou funkcionalitu vzhledem k definovaným požadavků v kapitole Návrh. Celkem byly ustanoveny čtyři. Prvním požadavkem na nástroj byla schopnost automaticky generovat identifikátory pro známé komponenty webové

```
1  public class ControlBSeleniumGenerator : SeleniumGenerator<ControlB>
2  {
3     public override DotvvmProperty[] NameProperties { get; } = { };
4     public override bool CanUseControlContentForName => false;
5     protected override void AddDeclarationsCore(
6        PageObjectDefinition pageObject, SeleniumGeneratorContext context)
7     {
8        // generator for user control so it's using PageObject not Proxy
9        // it's required create class ControlBPageObject
10       // in UI test project in PageObjects folder
11       string type = "PageObjects.MyControlBPageObject";
12       AddControlPageObjectProperty(pageObject, context, type);
13    }
14 }
```

Výpis 5.8. Ukázka vlastní implementace dílčího generátoru pro uživatelskou komponentu `ControlB`. V rámci implementace je nutné dědit z třídy `SeleniumGenerator`, která se nachází v projektu definující datové struktury generátoru. Jako její generický parametr je následně vložena třída reprezentující komponentu (řádek č. 1). Řádek č. 11 specifikuje typ objektu, jenž bude využit jako datový typ odpovídající vlastnosti v programovém popisu. Při specifikaci typu je důležité nezapomenout na celý jmenný prostor datového typu. Řádek č. 12 obsahuje volání metody, která připraví za pomoci frameworku Roslyn informace ke generování.

stránky. Součástí toho byly nastaveny pravidla pro vytváření názvů selektorů. Tento požadavek byl implementačně splněn a jeho implementace je převážně popsána v části 5.1.1.

Druhým požadavkem bylo definováno, že nástroj musí být schopen generovat pomocné třídy pro jednotlivá View a jemu známé komponenty. Pomocí nich je možné zpřístupnit jednotlivé elementy ve View. Tento požadavek byl naimplementován pomocí generování programových popisů, které obsahují vlastnosti ve formě *Proxy* objektů. Pomocí nich a frameworku Selenium je následně v testech umožněno provádět operace s odpovídajícími prvky UI.

S druhým požadavkem byl zároveň definován i třetí, a to nutnost nástroje obsahovat základní balíček objektů, které vědí s jakou komponentou pracují. Proto výsledný nástroj obsahuje zmíněné *Proxy* objekty pro komponenty z frameworku DotVVM, které jsou automaticky použity při generování programových popisů pro stránky s DotVVM elementy. Implementace obou požadavků je popsána v částech 5.1.1 a 5.2.

Posledním požadavkem nástroje je zajištění statické kontroly funkčnosti testů při kompilaci webové aplikace. Tato funcionalita byla zajištěna s využitím nástroje MSBuild a vytvoření nástroje jako konzolové aplikace. Popis funkčního mechanismu se nachází v sekci 5.1.2.

Kapitola 6

Testování a demonstrace funkčnosti nástroje

Tato kapitola pojednává o testování implementovaného nástroje. Nejprve se zaměřuji na kontrolu samotné implementace nástroje, tedy zda dokáže vygenerovat a zapsat požadované výsledky. V další části se zaměřím na demonstraci funkčnosti nástroje nad vzorovou webovou aplikací implementovanou pomocí frameworku DotVVM a využití vyprodukovaných výsledků a porovnání s klasickým přístupem. Poslední část této kapitoly je popis uživatelského testování.

Samotná kontrola funkčnosti je implementována v projektu `DotVVM.Testing.SeleniumGenerator.Tests`. Webová aplikace pro kontrolu vygenerovaných výsledků a jejich využití se nachází v projektu `SampleApp1`. Uživatelské testování bylo provedeno s několika UI testery, kteří znají a využívají framework DotVVM. Testování bylo prováděno z důvodu získání zpětné vazby. Na základě ní bylo provedeno několik úprav v implementaci nástroje.

6.1 Testování implementace nástroje

V rámci testování implementace bylo potřeba zkontrolovat tři nejdůležitější funkcionality nástroje. Zaprvé, zda nástroj správně generuje programové popisy. Zadruhé, jestli programový popis obsahuje správné *Proxy* vlastnosti odpovídající elementům v GUI. Zatřetí, zda ke známým komponentám byl doplněn atribut s vytvořeným selektorem. Za účelem těchto kontrol jsem založil .NET Core projekt `DotVVM.Testing.SeleniumGenerator.Tests`, který využívá testovací framework společnosti Microsoft — *MSTest*.

Součástí projektu je testovací třída `SeleniumGeneratorTests` obsahující testy pro výsledky generování programového popisu a úprav DOTHTML souboru. Další součásti jsou testovací třídy `UnitTests` a `ExtensionsTests`, které obsahují několik jednotkových testů (teoreticky popsáno v podsekci 2.1). Ty jsou zaměřeny na otestování jednoduchých metod bez komplexních závislostí, které následně lze využít v komplexnějších scénářích.

6.1.1 Kontrola výsledků generování

Pro testování výsledků vzešlých z generátoru bylo nutné odstínit testy od jakýkoliv předchozích běhů generátoru. Z toho důvodu veškeré testy z třídy `SeleniumGeneratorTests` využívají vlastní *pracovní plochy* v podobě třídy `WebApplicationHost`. Jejím vstupem je cesta k projektu s webovou aplikací, z níž si sestaví informace nutné pro pokračování testu, například cestu k metadatům DotVVM aplikace a cestu k testovacímu projektu.

```
1  [TestMethod]
2  public async Task SimplePage_CheckGeneratedProperties()
3  {
4    using (var workspace = new WebApplicationHost(TestContext, webAppDir))
5    { // process and compile project
6      workspace.ProcessMarkupFile("Views/SimplePage/Page.dothtml");
7      workspace.FixReferencedProjectPath(proxiesCsProjPath);
8      var compilation = await workspace.CompileAsync();
9
10     var pageObject = compilation.AssertPageObject(
11         "SampleApp1.Tests.PageObjects.SimplePage", "PagePageObject");
12     pageObject.AssertPublicProperty(typeof(RadioButtonProxy), "Person");
13     pageObject.AssertPublicProperty(typeof(TextBoxProxy),
       ↪ "Name_LastName");
14     pageObject.AssertPublicProperty(typeof(ButtonProxy), "Click");
15   }
16 }
```

Výpis 6.1. Ukázka jednoho z testů funkčnosti generátoru. Řádky č. 4–8 představují inicializaci, spuštění generátoru a kompilaci vygenerovaného testovacího projektu. Řádek č. 10 obsahuje kontrolu, zda projekt obsahuje třídu s programovým popisem pro požadovaný DOTHTML dokument. Následuje kontrola přítomnosti správných vlastností v programovém popise dle typu *Proxy* objektů a názvu vlastnosti.

Dalším krokem je inicializace třídy `WebApplicationHost`. Ta je provedena pomocí metody `Initialize()`, v jejímž rámci je celá složka s obsahem webové aplikace zkopírována do mnou vytvořené dočasné pracovní plochy s využitím Windows příkazu *xcopy*[1]. Cesta k dočasné pracovní složce je získána pomocí třídy `Path` a metody `GetTempPath()`, která vrátí tuto cestu pro aktuálně přihlášeného uživatele. Následně je připojeno náhodné GUID číslo, čímž vznikne kompletní cesta k mnou vytvořenému dočasnému adresáři. Tento adresář se pak nastavuje jako aktuální pracovní adresář pro testovací konzolovou aplikaci v třídě `Environment`[2], jenž reprezentuje aktuální pracovní prostředí.

Po dokončení inicializace je provedeno samotné zpracování vstupního DOTHTML souboru, generování programových popisů a úprava DOTHTML dokumentu, pomocí metody `ProcessMarkupFile`. Návratová hodnota metody spočívá v textovém obsahu zpracovávaného DOTHTML dokumentu po skončení běhu nástroje.

Následujícím krokem je kompilace nástrojem vygenerovaného testovacího projektu pro UI testy webové aplikace. Toho dosáhnu opět pomocí frameworku Roslyn. Výsledkem kompilace je třída `Compilation` z jmenného prostoru `Microsoft.CodeAnalysis`[3], nad kterou je vzápětí spuštěna diagnostika syntaxe apod. pro zjištění potencionálních chyb v projektu. V případě, že diagnostika vrátí nějakou diagnózu s úrovní problému — chyba — je test automaticky prohlášen za neúspěšný kvůli nemožnému sestavení projektu.

Jestliže se povedlo zkompilovat testovací projekt, je možné přistoupit k samotným testům. Ukázka jednoho z testů je ve výpisu 6.1. Ta obsahuje testovací metodu frameworku

[1] `https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/xcopy`
[2] `https://docs.microsoft.com/cs-cz/dotnet/api/system.environment`
[3] `https://docs.microsoft.com/en-us/dotnet/api/microsoft.codeanalysis`

MSTest, kontrolující vygenerování správných typů vlastností a názvů v programovém popisu. Kontrola přítomnosti třídy pro programový popis je implementována pomocí *rozšiřující metody* `AssertPageObject()` na třídě `Compilation`. Ta se snaží nalézt v zkompilovaném projektu symbol s předpokládaným názvem (např. *PagePageObject*) a provést kontrolu jmenného prostoru této třídy s testovaným. Její volání je vidět v ukázce 6.1 na řádku č. 10.

Další důležitý test obsahuje kontrolu vygenerování atributů „UiTests.Name“ se selektory přímo do DOTHTML dokumentů (metoda `SimplePage_CheckGeneratedUiNames()`. Test se skládá ze dvou částí. Zaprvé obsahuje inicializaci popsanou v předchozích odstavcích a kontrolu vygenerování třídy s programovým popisem. V druhé části je potřeba získat DotVVM konfiguraci webové aplikace a získat abstraktní strom webové stránky. Dále je nutné strom projít, opět pomocí vlastní implementace třídy `ResolvedControlTreeVisitor` z DotVVM frameworku, a získat všechny komponenty s požadovaným atributem (práce s abstraktním stromem a nalezení komponent bylo více popsáno v části 5.1.1). Samotné otestování následně spočívá v počtu objevených elementů a porovnání textových řetězců selektorů s předpokládanými výsledky.

Mezi další testy funkčnosti generátoru patří kontrola vygenerování separátního programového popisu pro komponenty, u kterých je to očekávané (jmenovitě `Repeater`) nebo kontrola připojení názvu datového kontextu rodičovských prvků v textovém řetězci vytvořeného selektoru u vnořených komponent.

6.1.2 Jednotkové testy

V předchozí podsekci jsem probíral testy celého nástroje, které bych zařadil především mezi systémové nebo integrační typy testů. Abych se mohl spolehnout, že složitější metody zaobalující více logiky budou bez problémů fungovat, rozhodl jsem se otestovat i několik jednodušších metod. Ty se skládají hlavně z metod s malým množstvím vstupů nebo bez složitých závislostí, které by se špatně napodobovaly. Tyto požadavky ovšem hodně vyfiltrovaly množinu metod vhodných k provedení jednotkových testů.

S tím, že jednoduché metody byly většinou využívány jen jako součást komplexnějších scénářů, byly všechny psány jako neveřejné. Pro lepší testovatelnost bylo nutné upravit modifikátory přístupů na `internal` a ve vlastnostech projektu `DotVVM.Framework.Tools.SeleniumGenerator` (soubor *AssemblyInfo.cs*) doplnit atribut `[assembly: InternalsVisibleTo("DotVVM.Testing.SeleniumGenerator.Tests")]`. Atribut mi umožní v testovacím projektu přistoupit k metodám, které mají nastavený modifikátor přístupu jen pro vlastní *assembly* (`internal`).

Nejvíce jsem se zaměřil na otestování třídy `SelectorStringHelper` zajišťující provádění operací nad textovými řetězci pro úspěšné vytvoření selektoru. Většina z těchto metod byla volána z metod `AddDeclarations()` a `DetermineName()`, které k vytvoření jména vlastnosti programového popisu a názvu selektoru slouží. Dalšími jednotkovými testy byly kontroly metod v třídě `SeleniumGeneratorOptions` pro správné přidání vlastních dílčích generátorů a sestavení (*assembly*), v kterých se mají hledat dílčí generátory. Posledními jednotkovými testy jsem pokryl kontrolu rozšiřující metody `AddRange()` nad třídou `Dictionary`.

6.2 Demonstrace funkčnosti

Po kontrole korektní implementace nástroje přišlo na řadu testování a demonstrace funkčnosti nástroje. V této části se chci především zaměřit na porovnání aktuálního formátu testů

s ukázkami předvedenými v předchozích kapitolách. Dále pak na srovnání programových popisů a pomocných objektů.

Vygenerování jednotlivých programových popisů bylo provedeno pomocí spuštění příkazu `dotvvm gut`. Další variantou bylo využití automatického spuštění generování programových popisů během kompilace. Z důvodu, že tato operace může pro více DOTHTML dokumentů a komponent trvat několik sekund, měl jsem tuto možnost během vývoje a testování vypnutou.

Jestliže je kontrola během kompilace zapnuta, což lze udělat pomocí úlohy z výpisu 5.5, je kompilace a sestavení testovacího projektu závislé na úspěšném běhu celého nástroje. Jak je tohoto mechanismu využito přímo v praxi jsem předvedl a popsal ve výpisu 6.2.

```
1 <dot:Button Text="Klik" Click="{command: Click()}" UITests.Name="Klik"/>
2
3 var pageObject = new TestPageObject(driver);
4 pageObject.Klik.Click();
5
6 <dot:Button Text="Klik" Click="{command: Click()}"
7            UITests.Name="Click" />
```

Výpis 6.2. Příklad DOTHTML a vygenerovaných selektorů v různých bězích generátoru. Na řádku č. 1 má selektor hodnotu „Klik“. Ten samý název má vlastnost reprezentující tlačítko v programovém popise `TestPageObject`. Na řádcích č. 3–4 je ukázka použití této vlastnosti v testu. Následně po ruční úpravě selektoru na hodnotu „Click“ (řádek č. 6) a spuštění kompilace testovacího projektu, neskončí toto sestavení úspěšně. Stane se to z důvodu přegenerování programového popisu nástrojem a změnu názvu vlastnosti na „Click“. Celý tento proces se provede automaticky, díky specifikaci úlohy *SeleniumGeneratorTarget* z výpisu 5.5.

Příklady použití nástroje

Jako první příklad v demonstraci použití je zařazen DOTHTML dokument ukázaný ve výpisu 4.1. Na tomto formuláři jsem v kapitole 4 vysvětloval, jak by proces UI testování probíhal bez využití navrhovaného nástroje, přičemž ukázku takového testu (bez inicializace) jsem představil ve výpisu 4.2. Do jakého stavu se dostala testovací metoda (také bez inicializace Selenium WebDriver) po spuštění nástroje nad vypsaným DOTHTML lze vidět ve výpisu 6.3. Výsledkem jsou veškeré vlastnosti zapouzdřené v programovém popisu `SignInPageObject`, který může tester využít v jakémkoliv testu, v němž by potřeboval přístup k tomuto přihlašovacímu formuláři.

Druhým příkladem je ukázka práce s programovým popisem, který obsahuje ve více nezávislých úrovních stejné selektory. Tato situace nastává při generování programového popisu pro *Markup* komponentu. Ten se vytváří nezávisle na View, v kterém je použita, a proto se může tato kolize vyskytnout. Pro zachování čitelnosti nejsou tyto kolize řešeny na úrovni generování, ale při vyhledávání požadované komponenty ve View. Tento proces byl popsán v sekci 5.2.2. Tento test tedy spočívá v zadání a získání jména v první úrovni formuláře a přečtení textového řetězce titulku druhého řádku v komponenty `Repeater` (oboje selektor „Name“). Kvůli tabulkové komponentě `Repeater` je tedy selektor „Name“ ve stránce tolikrát, kolik položek obsahuje, čímž příklad ještě komplikuje. Celý test s popisem a počátečním stavem stránky lze vidět na obrázku 6.1.

```
1  [TestMethod]
2  public void SignInPage_TestLoginSuccess()
3  {
4      // Initialize
5      var pageObject = new SignInPageObject(driver);
6
7      // Do
8      pageObject.Email.SetText("test@test.com");
9      pageObject.Password.SetText("test");
10     pageObject.Login.Click();
11
12     // Assert
13     Assert.IsFalse(pageObject.ValidationSummary.IsVisible());
14 }
```

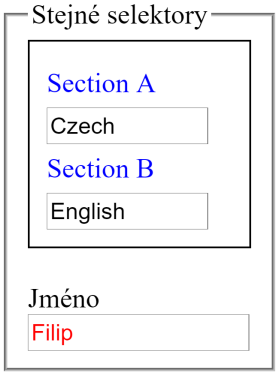
Výpis 6.3. Ukázka testovací metody, která odpovídá testovací logikou výpisu 4.2. V aktuální verzi je využit vygenerovaný programový popis `SignInPageObject` pro DOTHTML dokument z ukázky 4.1. Zároveň je možné si všimnout názvu selektoru „Login“ pro tlačítko „Přihlásit se“. Toho bylo dosáhnuto pomocí testerova přepsání selektoru přímo v DOTHTML, aby se vlastnost definující tlačítko nejmenovala „PrihlasitSe“.

Oba příklady jsou součástí projektu webové aplikace `SampleApp1`. Ta představuje aplikaci vytvořenou v DotVVM a postavenou nad .NET Core frameworkem. Testy UI se nacházejí v projektu `SampleApp1.Tests`. Tento projekt je také postaven nad .NET Core platformou a byl vygenerován při běhu nástroje. Webový projekt obsahuje několik webových stránek. Veškeré DOTHTML definice se nacházejí ve složce *Views* a definice *Markup* komponent ve složce *Controls*. Testovací projekt pak obsahuje adresář *PageObjects* s programovými popisy všech webových stránek i komponent pro webovou aplikaci.

Adresář *Views* shromažďuje několik složek, jenž obsahují webové stránky různé složitosti. Složka *SimplePage* obsahuje testovací případy, které nepracují s uživatelskými *Markup* komponentami. Součástí jsou ovšem příklady zaobalení komponent do vnitřního datového kontextu nebo přidávání více kontrolek se stejnými vlastnostmi, pro které by mohla vzniknout kolize vygenerovaných selektorů. V adresáři *MasterPages* lze nalézt soubory představující *Master pages*. Pomocí těch a samotných DOTHTML dokumentů jsem testoval generování programových popisů stránek zanořených ve vícero rodičovských stránek. Ve složce *Controls* jsou DOTHTML Views, které obsahují testovací případy pro generování programových popisů s využitím již vygenerovaných programových popisů pro *Markup* komponentu. Součástí jsou i stránky pro kontrolu využití vlastních dílčích generátorů pro specifikaci použití testerem implementovaných pomocných objektů a programových popisů.

Veškeré DOTHTML stránky i *Markup* komponenty byly připraveny především pro využití v testech kontrolujících funkčnost. Proto jsem nevěnoval tolik pozornosti vzhledu, přípravě kaskádových stylů apod. Samotný testovací projekt sloužil především pro kontrolu vývoje nové funkcionality výsledného nástroje.

```
1  [TestMethod]
2  public void SameSelectorsForControls()
3  {
4    var pageObj = new SelectorsPageObject(driver);
5
6    pageObj.Name.SetText("Filip Kalous");
7
8    var section = pageObj.Sections.GetItem(1);
9    var secText = section.ControlB.Name.GetText();
10   var languageText = section.Language.GetText();
11
12   var nameText = pageObj.Name.GetText();
13
14   Assert.AreEqual(secText, "Section B");
15   Assert.AreEqual(languageText, "English");
16   Assert.AreEqual(nameText, "Filip Kalous");
17 }
```



Obrázek 6.1. V levé části se nachází test pracující se stejnými selektory komponent, ale v různých kontextech. V pravé části obrázek stavu stránky před provedením testu. Test má následující průběh. Nejdříve se manipuluje s textovým vstupem pro jméno, jenž se nastaví na „Filip Kalous“ (řádek č. 6). Řádky č. 8–10 pracují s komponentou `Repeater` a jejím 2. řádkem (metoda `GetItem()` indexuje od 0). Po získání hodnoty z uživatelské komponenty `ControlB`, jenž reprezentuje titulek řádku, následuje získání řetězce z textového vstupu pro jméno (řádek č. 12) a porovnání očekávaných a dosažených výsledků (řádek č. 14–16).

6.3 Uživatelské testování

Jednou z fází testování bylo uživatelské testování, které jsem provedl za účelem zjištění zpětné vazby a vylepšení nástroje. Testování jsem prováděl pravidelně v průběhu vývoje, přičemž nejintenzivněji po dokončení implementace generování programových popisů. Na testování jsem spolupracoval s týmem vývojářů frameworku DotVVM. Výstupem byla řada upozornění na nedostatky ve vygenerovaných výstupech nebo návrhy na vylepšení, které byly postupně zapracovávány do nástroje.

První připomínka byla zaznamenána již v době návrhu nástroje a týkala se generování řetězce, který bude využit jako selektor. Původní plán byl řešit názvy selektorů jen podle samotné komponenty. Na základě zpětné vazby jsem se následně rozhodl, že do názvu selektoru by měl být propagován i datový kontext nadřazených komponent. Výsledný návrh je pak vidět v tabulce 4.1.

Důležitým implementačním detailem, který vznikl ze získané zpětné vazby, bylo využití vlastních programových popisů pro prvky reprezentující tabulkové sloupce v komponentě `GridView`. A to ve formě vlastnosti v programovém popisu. Původní implementace spočívala ve využití klasické generické *Proxy* vlastnosti, která poskytovala skrze metodu `GetItem()`instanci konkrétního programového popisu pro sloupec. Tato změna vylepšila čitelnost a orientaci testera v kódu samotného testu. Mezi další zjištěné nedostatky nebo návrhy na zlepšení patřily tyto:

- umožnit získat `IWebElement`, tedy rozhraní reprezentující prvek GUI ve frameworku Selenium,
- využití jiných atributů a vlastností komponent,
- nevyužít syntaxe selektorů ve formátu zapadající do HTML (např. `name-selector`),
- návrhy metod pro pomocné objekty komponent (*Proxy*),
- využít *selenium-utils*,
- odstranění nepotřebných jmenných prostorů u vlastností v programovém popisu.

Samotné testování probíhalo, jak přímo s webovou aplikací `SampleApp1`, tak nad jinými DotVVM projekty. Například s aplikací CheckBook[4], což je aplikace pro sledování výdajů mezi skupinou lidí, dostupnou na portálu GitHub.

Na závěr bych se zaměřil na zhodnocení mnou provedeného testování. I když jsem testování prováděl pravidelně během implementace, určitě bych můj přístup nenazýval programováním řízeném testy (TDD[5]). Testy zaměřující se na implementaci jsem využíval především ve chvíli změn logiky generování, což zjednodušilo kontrolu vygenerovaných výsledků. Avšak převážně jsem se zaměřil na využití vygenerovaných programových popisů pro UI testování nad webovou aplikací. Velkým přínosem bylo určitě uživatelské testování. Díky této zpětné vazbě jsem odhalil několik nedostatků, které jsem z osobního pohledu za problémy nepovažoval.

[4] `https://github.com/riganti/dotvvm-samples-checkbook`
[5] TDD — Test Driven Development, více v knize *Test-Driven Development by Example*

Kapitola 7

Závěr

Cílem diplomové práce bylo seznámit se s problematikou testování uživatelských rozhraní webových aplikací a následně porovnat aktuální frameworky, které se na toto téma specializují. Další fází bylo definovat požadavky pro výsledný nástroj a na základě definovaných požadavků provést jeho návrh. Jako další součást práce následovalo vytvoření názorné implementace v jazyce C# pro framework DotVVM. Tuto implementaci jsem následně podrobil kontrole funkčnosti. Poslední fází bylo testování výsledného nástroje nad DotVVM webovou aplikací a ověření funkčnosti vygenerovaných výstupů.

Dle poznatků z teoretické části práce a aktuální situace testování GUI, která byla vysvětlena na praktickém příkladu, byly v kapitole 4 definovány funkční požadavky. V souladu s nimi vznikl návrh nástroje, respektive generátoru, sestávající se z dvou částí. Zaprvé, z části, která navrhuje generování identifikátoru ke komponentám a úpravu DOTHTML dokumentu. Zadruhé, návrh fungování generování pomocných tříd použitelných v GUI testech. Součástí návrhu je i nastínění využití výsledků nástroje a porovnání tvorby testů bez a s použitím nástroje. Jediné dané stanovisko bylo využití frameworku DotVVM. Jeho výsledkem byla definice požadavků, které je nutné splnit pro úspěšné hodnocený výsledek, společně s návrhy jejich možného řešení.

Návrh byl proveden spíše abstraktní formou, pro větší svobodu v době implementace. Nezabýval se implementačními detaily, jako je použití konkrétních implementačních technologií, vývojových prostředí a návrhových vzorů. Tato rozhodnutí byla provedena až jako součást implementační fáze. Byly vybrány technologie a nástroje, s kterými jsem pracoval v době vývoje a dále pak technologie, které jsou využity k běhu nástroje. Výsledkem úspěšné implementace je splnění definovaných požadavků, a to ve formě konzolové aplikace. Tu lze pustit na vyžádání nebo jako úlohu v rámci sestavování UI testů.

Funkčnost výsledku byla otestována pomocí několika testů, jak pro účely kontroly implementace, tak pro účely kontroly využití vygenerovaných výsledků v UI testech. Role testů byla již během implementace velmi přínosná, především v době přepracování již naprogramovaných řešení. Pro zjištění možných chyb a vylepšení, jsem navíc provedl i uživatelské testování s vývojáři frameworku DotVVM. Na základě získané zpětné vazby byly opraveny různé nedostatky a náležitě upravena implementace pro rychlejší a jednodušší práci s generovanými programovými popisy.

Výhled do budoucna

Implementovaný nástroj aktuálně funguje jako vlastní projekt a také je dostupný jako repozitář na serveru GitHub[1]. Současně také jako součást frameworku DotVVM. Hlavním cílem do budoucna je vyladit nástroj na takovou úroveň, aby ho mohlo být možné vydat jako součást hlavní větve DotVVM. Jednou z věcí, která tomuto cíli může pomoci, bude využití frameworku *selenium-utils*[2]. Ten slouží k ulehčení práce s frameworkem Selenium a funguje na principu obalení funkcionality, kterou poskytuje Selenium.

Dalším vylepšením bude přidání podpory více komponent, s kterými zvládne generátor pracovat. Aktuálním nedostatkem je například neumožnění práce s komponentami, které nevyužívají obalovacích značek (tzv. *wrapper tagů*). Jedno z možných řešení by například spočívalo ve využití komentářů ve View, které by implikovaly přítomnost vygenerovaného selektoru pro konkrétní komponentu.

V rámci frameworku DotVVM je možné využívat i rozšíření pro IDE Visual Studio. To umožňuje rychlejší práci s DOTHTML dokumenty apod. Další možnou funkcionalitou implementovaného nástroje by bylo přidání jeho podpory do tohoto rozšíření. Tato podpora by spočívala v nabídce generování jednotlivých programových popisů na vyžádání přes kontextové menu rozšíření.

[1] Repozitář nástroje — `https://github.com/riganti/dotvvm-selenium-generator`
[2] `https://github.com/riganti/selenium-utils`

Slovník

GUI Graphical user interface (Grafické uživatelské rozhraní). 3–5, 7–11, 14, 19, 27, 28, 31, 32, 45, 51, 52, 54

Model je vrstva architektonického vzoru MVVM, která popisuje data, se kterými aplikace pracuje. 15

Model-View-ViewModel je architektonický používaný k tvorbě aplikací. Jeho podstatou je rozdělení aplikace na tři vrstvy — Model (popisuje datovou doménu), View (definuje UI) a ViewModel (reprezentuje aplikační logiku). 15, 16

OCR zkratka pro Optical Character Recognition – technologie umožňující konverzi obrazu zpět na text. 11

RDP značí síťový protokol umožňující uživateli ovládat vzdálený počítač prostřednictvím počítačové sítě. 11

Solution v českém překladu řešení. Slouží jako zastřešující kontejner více souvisejících projektů, v programu Visual Studio 2019. 29, 38

View reprezentuje uživatelské rozhraní webové aplikace podle architektonického vzoru MVVM. 15–17, 19–21, 23, 24, 26–30, 32–34, 40, 41, 44, 48, 49, 53

ViewModel slouží jako most mezi vrstvami View a Model v architektonickém vzoru MVVM. 15–17

VNC zkratka pro Virtual Network Computing — program umožňující vzdálené připojení ke GUI pomocí počítačové sítě. VNC server vytváří grafickou plochu v operační paměti a komunikací po počítačové síti s klientem zobrazuje plochu uživateli. 11

Literatura

[1] Banerjee, I.; Nguyen, B.; Garousi, V.; aj.: Graphical user interface (GUI) testing: Systematic mapping and repository. *Information and Software Technology*, ročník 55, č. 10, 2013: s. 1679 – 1694, ISSN 0950-5849.

[2] Brockmeyer, U.: Model-Based Testing: what it is and what does it entail? *IBM Community*, online, [cit. 2019-01-16].
URL `https://www.ibm.com/developerworks/community/blogs/35dfcb99-111b-423a-aaa4-50f3fddae141/entry/Model_Based_Testing_what_it_is_and_what_does_it_entail?lang=en`

[3] Introduction to Control Development. *DotVVM Docs*, online, [cit. 2019-04-15].
URL `https://www.dotvvm.com/docs/tutorials/control-development-introduction/latest`

[4] Creating the First Page. *DotVVM Docs*, online, [cit. 2019-04-16].
URL `www.dotvvm.com/docs/tutorials/basics-first-page/latest`

[5] Introduction. *DotVVM Docs*, online, [cit. 2018-12-17].
URL `https://www.dotvvm.com/docs/tutorials/introduction/2.0`

[6] Markup-Only Controls. *DotVVM Docs*, online, [cit. 2019-04-23].
URL `https://www.dotvvm.com/docs/tutorials/control-development-markup-only-controls`

[7] Master Pages. *DotVVM Docs*, online, [cit. 2019-04-15].
URL `https://www.dotvvm.com/docs/tutorials/basics-master-pages/latest`

[8] Zdrojový kód frameworku DotVVM. *GitHub*, online, [cit. 2019-04-27].
URL `https://github.com/riganti/dotvvm`

[9] Erich, G.; HELM, R.; JOHNSON, R.; aj.: Proxy, Visitor. In *Návrh programů pomocí vzorů*, Praha: Grada, 2003, ISBN 80-247-0302-5, s. 207–217, 331–344.

[10] The Forrester Wave^{TM}: Omnichannel Functional Test Automation Tools, Q3 2018. *Forrester*, 2018, online, [cit. 2019-05-05].
URL `https://www.forrester.com/report/The+Forrester+Wave+Omnichannel+Functional+Test+Automation+Tools+Q3+2018/-/E-RES140737#`

[11] Gandhi, G. M. D.; Pillai, A. S.: Challenges in gui test automation. *International Journal of Computer Theory and Engineering*, ročník 6, č. 2, 2014: str. 192.

[12] Hetzel, W. C.: *The complete guide to software testing*. Wellesley, Mass.: QED Information Sciences, druhé vydání, 1988, ISBN 08-943-5242-3.

[13] HTML id Attribute. *W3schools.com*, online, [cit. 2019-01-12].
URL `https://www.w3schools.com/tags/att_global_id.asp`

[14] Leotta, M.; Clerissi, D.; Ricca, F.; aj.: Improving Test Suites Maintainability with the Page Object Pattern: An Industrial Case Study. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, March 2013, s. 108–113, doi:10.1109/ICSTW.2013.19.

[15] M. Memon, A.; Soffa, M.; E. Pollack, M.: Coverage Criteria for GUI Testing. *ACM SIGSOFT Software Engineering Notes*, ročník 26, 07 2001, doi:10.1145/503271.503244.

[16] Striking a Balance Between Manual and Automated Testing: When Two Is Better Than One. *AltexSoft*, online, [cit. 2019-05-04].
URL `https://www.altexsoft.com/blog/engineering/striking-a-balance-between-manual-and-automated-testing-when-two-is-better-than-one/`

[17] McPeak, A.: The Difference Between Manual vs Automated Testing.
URL `https://crossbrowsertesting.com/blog/test-automation/difference-manual-automated-testing/`

[18] Memon, A. M.; Pollack, M. E.; Soffa, M. L.: Using a Goal-driven Approach to Generate Test Cases for GUIs. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, New York, NY, USA: ACM, 1999, ISBN 1-58113-074-0, s. 257–266, doi:10.1145/302405.302632.

[19] Mrnuštík, M.: *Mezijazykový překladač C#-JavaScript pro DotVVM*. Bakalářská práce, Vysoké učení technické v Brně, Brno, 2018.

[20] MSBuild. *Visual Studio Docs*, online, [cit. 2019-04-25].
URL `https://docs.microsoft.com/cs-cz/visualstudio/msbuild/msbuild`

[21] The MVVM Pattern. *Microsoft Docs*, 2018, online, [cit. 2018-12-15].
URL
`https://docs.microsoft.com/en-us/previous-versions/msp-n-p/hh848246(v=pandp.10)`

[22] Osherove, R.: *The art of unit testing with examples in C#*. New York: Manning, druhé vydání, 2014, ISBN 978-1-617290-89-3.

[23] Pezzè, M.; Young, M.: *Software testing and analysis*. Hoboken: John Wiley, 2008, ISBN 978-0-471-45593-6.

[24] Schiller, K.: Getting Started with Page Object Pattern for Your Selenium Tests. *PluralSight*, online, [cit. 2019-01-11].
URL `https://www.pluralsight.com/guides/getting-started-with-page-object-pattern-for-your-selenium-tests`

[25] Introduction. *SeleniumHQ*, online, [cit. 2018-12-19].
URL `https://www.seleniumhq.org/docs/01_introducing_selenium.jsp`

[26] IWebElement Interface. Online, [cit. 2019-04-14].
URL https://seleniumhq.github.io/selenium/docs/api/dotnet/html/T_OpenQA_Selenium_IWebElement.htm

[27] Code Generation and T4 Text Templates. *Visual Studio Docs*, online, [cit. 2019-04-17].
URL https://docs.microsoft.com/en-us/visualstudio/modeling/code-generation-and-t4-text-templates

[28] Understanding the basics of MVVM design pattern. *Microsoft Developer*, online, [cit. 2018-12-15].
URL https://blogs.msdn.microsoft.com/msgulfcommunity/2013/03/13/understanding-the-basics-of-mvvm-design-pattern/

[29] Xie, Q.; Memon, A. M.: Using a Pilot Study to Derive a GUI Model for Automated Testing. *ACM Trans. Softw. Eng. Methodol.*, ročník 18, č. 2, Listopad 2008: s. 7:1–7:35, ISSN 1049-331X.

[30] Zhan, Z.: *Selenium WebDriver Recipes in C#*. California: Apress, second edition vydání, [2015], ISBN 978-1-484217-41-2.

Příloha A

Obsah CD

V této příloze se nachází popis obsahu CD, které je přiloženo k originálnímu výtisku této práce. Na přiloženém CD se nachází elektronická práce ve formátu PDF, originál zdrojových souborů LaTeX a zdrojové kódy vytvořeného nástroje. Struktura disku z pohledu kořenového adresáře odpovídá git repozitáři, kde byl nástroj vyvíjen a vypadá následovně:

- *dotvvm-selenium-generator/* — adresář obsahující zdrojové kódy nástroje a frameworku DotVVM
  - *dotvvm/* — adresář s upraveným zdrojovým kódem frameworku DotVVM
  - *src/* — adresář se zdrojovými kódy implementované nástroje. V adresáři se nachází pět podadresářů. Podadresář *Samples* obsahuje webovou aplikaci `SampleApp1`. Ostatní podadresáře jsou součástí implementovaného nástroje
- *docs-src/* — adresář obsahující zdrojové kódy této zprávy ve formátu LaTeX;
- *docs-build/xkalou03.pdf* — dokument ve formátu PDF obsahující diplomovou práci ve stavu, v jakém byla odevzdána v informačním systému.
- *README.md* — textový soubor ve formátu *markdown* reprezentující manuál pro spuštění nástroje