Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Bakalářská práce

Generování testovacích datasetů

Plzeň 2018 Lenka Šimečková

Místo této strany bude zadání práce.

Prohlášení

Prohlašuji, že jsem bakalářskou práci vypracovala samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 2. května 2018

Lenka Šimečková

Abstract

The goal of this thesis is creating an application for automatic creation of web application testing datasets, usable in automated testing. Input of the application is graph of states and transitions of tested application created in Oxygen tool, respectively XML files exported from this tool. Output will be testing datasets directly usable in generated automatized tests. For testing, the Selenium WebDriver library is used.

Abstrakt

Cílem této práce je vytvořit aplikaci, jenž bude automaticky vytvářet testovací datasety pro webové aplikace, použitelné v automatizovaném testování. Vstupem aplikace je graf stavů a přechodů testované aplikace vytvořený v nástroji Oxygen, respektive XML soubory exportované z tohoto nástroje. Výstupem budou testovací datasety přímo využitelné v generovaných automatizovaných testech. Pro testování je využita knihovna Selenium WebDriver.

Obsah

1 Úvod

Testování webových aplikací by mělo být nedílnou součástí jejich vývoje. Často se však jedná o opomíjenou oblast, jíž se věnuje málo času a je prováděna nesystematicky. Možným řešením je automatizace tohoto procesu. Toho lze docílit pomocí nástrojů speciálně navržených pro testování webových aplikací. Jedním z nich je projekt Selenium.

Na FEL ČVUT je vyvíjen nástroj Oxygen pro generování procesních testů aplikací. Jeho výstupem je však pouze seznam testovacích situací a je třeba jednotlivé testovací situace převést do podoby scénáře pro manuální, v optimálním případě pro automatizované testování a testy doplnit vhodnými vstupními daty.

Zároveň byla na KIV ZČU obhájena diplomová práce, v rámci které byla vytvořena knihovna Java Object Populator pro generování testovacích dat pomocí anotací.

Cílem této práce je ověřit možnost propojení těchto dvou nástrojů a vytvořit aplikaci, která bude na základě dat exportovaných z Oxygenu generovat sady testovacích dat pro automatizované testování. V případě automatizovaných testů by se jednalo o skripty pro Selenium.

Výsledkem bude strojově generovaná sada automatizovaných testů, která zajistí mnohem důkladnější „protestování“ aplikace v porovnání s dosud používanými ručními postupy.

2 Testování webových aplikací

Testováním rozumíme kontrolovaný proces s určitými podmínkami, vyhodnocující kvalitu softwaru. Tato kvalita je definována nalezenými výhodami produktu (jeho vlastnostmi) a problémy (jeho chybami). Testování jako takové je velmi důležité a mnohdy i klíčové. Důkladné testovaní software již od počátku jeho vytváření sníží cenu produkce v dlouhodobém horizontu. Chyby totiž mohou být objeveny kdykoliv během výrobního cyklu a ty, které vyvstanou v pozdějších fázích, jsou do značné míry nákladnější co se týče jejich opravení. [1]

2.1 Webové aplikace

Pojmem webové aplikace označuje aplikace odpovídající struktuře klient-server[1]. Jedná se tedy o služby běžící na vzdáleném serveru, které jsou uživatelům přístupné pomocí počítačové sítě, za použití internetového prohlížeče. Takové aplikace mohou nabízet velmi jednoduchou funkcionalitu, ale rovněž mnohdy komplexní a složitou logiku, například v podobě informačních systémů. [7] Jádrem každé webové aplikace je skutečnost, že její funkcionalita je řízena komunikací pomocí HTTP a jejich výsledky jsou obvykle formátovány pomocí HTML. [6]

2.2 Oblasti testování webových aplikací

2.2.1 Funkční testování

Jak název napovídá, tímto způsobem je kontrolována aplikace podle její specifikace a funkčních požadavků. Dochází k ověřování správnosti chování na základě definovaného nebo očekávaného chování.

V rámci funkčního testování jsou zahrnuty následující:

- **hyperlinky** – kontrola správnosti odkazů a eliminace existence nefunkčních odkazů napříč aplikací,
- **formuláře** – správnost výchozích hodnot, jejich správné uložení (např. do databáze) a ověření korektního zobrazení formuláře,

[1] Síťová architektura, která odděluje klienta (často aplikaci s grafickým uživatelským rozhraním) a server, kteří spolu komunikují přes počítačovou síť.

- **cookies** – kontrola chování aplikace při povolených a zakázaných cookies. Otestování jejich zašifrování před zápisem do uživatelova počítače,
- **HTML a CSS** – kontrola chyb v syntaxi a jejich celková validace,
- ***workflow*** – kontrola správnosti scénářů jak pozitivní, tak negativních,
- **databáze** – kontrola zobrazení korektních informací obsažených v databázi

2.2.2 Testování obsahu

Zjišťuje, zda stránka obsahuje všechny potřebné elementy, zda jsou viditelné, jsou správných (očekávaných) rozměrů, obrázky jsou správně nataženy, je použito správné kódování textu a mnoho dalšího.

2.2.3 Kompatibilita s prohlížeči

Daná aplikace by měla být kompatibilní s většinou běžných webových prohlížečů. Tím se rozumí, že v aplikaci by měly být použity jen ty technologie, které jsou prohlížeči podporovány. Ideální stav je, že aplikace při použití různých prohlížečů vypadá ve všech ohledech stejně. Toho je samozřejmě často velmi obtížné dosáhnout a proto je potřeba si určit hranici, jaké odlišnosti lze ještě tolerovat. Podstatné je, že se musí chovat všude stejně.

2.2.4 Testování transakcí

Informace vložená uživatelem se musí správně propsat do databáze a uživatel musí z databáze dostat ta správná data. Významné hlavně pro informační systémy a aplikace v bankovnictví.

2.2.5 Konfigurační testování

Ověřuje funkčnost aplikace v závislosti na operačním systému, typu internetového připojení nebo poskytovateli internetu.

2.2.6 Použitelnost

Zde se hodnotí *user experience*[2]. Aplikace musí být navržená tak, aby bylo pro každého uživatele snadné ji použít a byla intuitivně ovladatelná. Tato

[2] *User Experience* (UX) – vnímání a reakce osoby plynoucí z používání produktu

oblast je poměrně obtížně testovatelná a většinou se zde využívá skupiny konečných uživatelů, kteří aplikaci přímo hodnotí.

2.2.7 Výkon a škálovatelnost

Aplikace by měla co se výkonnosti týká splňovat požadavky zadání v oblasti rychlosti odezvy. To například znamená, za jak dlouho aplikace provede změny po akci uživatele, jak rychlé stahování souboru umožňuje nebo kolik dokáže provést transakcí za jednotku času. Míra škálovatelnosti potom určuje, kolik aplikace zvládne naráz uživatelů nebo transakcí. Zároveň však musí mít i při vysoké zátěži akceptovatelnou míru odezvy.

2.2.8 Bezpečnost

Testování se obvykle skládá ze dvou částí – testování infrastruktury a testování zranitelnosti samotné aplikace, jako je například správné použití cookies a autentizace uživatele. Tato oblast je kritická především pro oblast bankovnictví.

2.3 Manuální a automatizované testování

K další možností rozdělení způsobu testování patří rovněž forma provádění testů. Rozlišuje se testování manuální (prováděné člověkem) a automatizované (prováděné počítačem). Oba způsoby mají své přednosti a nevýhody.

2.3.1 Manuální testování

V případě manuálního testování se tester ocitá v roli koncového uživatele a ověřuje funkčnost aplikace. Typickým konkrétním případem bývá zpravidla ruční ověření správnosti chování aplikace na základě připravených testovacích scénářů.

Jeho největší využití lze nalézt v následujících typech testů.

Průzkumné testování

Průzkumné neboli exploratorní testování není spojené s žádným konkrétním scénářem. Tester zkoumá aplikaci a na tomto základě provádí její testování. Díky tomu potřebuje méně času na přípravu a rovněž nalezne potenciální problém rychleji. Nalezený problém může být vedle funkční chyby i neintuivní uživatelská přívětivost aplikace. I z tohoto důvodu je tento způsob vhodný pro zkušené a kreativní testery. [10]

Ad-hoc testování

Jedná se spíše o doplňující způsob testování, kdy je na rozdíl od průzkumného testování, které je systematičtější a důkladnější, aplikace zcela nesystematicky testována, bez jakéhokoliv plánování. Znovu ale naráží na to, že zkušený tester může nalézt nemalý počet chyb a tímto je tato zdánlivě neefektivní technika užitečná. [10]

2.3.2 Automatizované testování

Automatizovaným testováním se rozumí spouštění testovacích skriptů, které ověřují funkčnost a správnost aplikace. Jedná se tedy o testy, které při svém běhu nevyžadují přítomnost samotného testera.

Automatizované procesní testy

Procesní testy jsou typem testování aplikace, který využívá scénářů. Výhody automatizace procesních testů spočívají především v tom, že jsou systematické a mohou pokrýt množství kombinací vstupních dat a optimálně všechny cesty aplikací. Součástí scénářů jsou vždy i očekávané výsledky.

Poloautomatizované testy

Poloautomatizované testování je kombinací automatizovaného a manuálního testování. Jedná se o takové testování, které je sice automatizované, ale zároveň je nutná přítomnost testera, který se stará o spouštění a správu automatizovaných procesů, případně vyhodnocuje jejich výstupy.

2.3.3 Porovnání automatizovaného a manuálního testování

V tabulce 2.1 je uvedeno srovnání výše uvedených způsobů testování. [9]

Tabulka 2.1: Srovnání manuálního a automatizovaného testování

Manuální	**Automatizované**
Vhodné při provádění testovacího scénáře jednou nebo dvakrát	Vhodné při provádění opakovaného testování
Možnost okamžitého testování	Nutnost prvotního vytvoření testů (naprogramování)
Umožňuje kontrolovat i strojově netestovatelné ukazatele, například *user experience*	Kontroluje jen to, co může software testovat
Riziko chyby lidského faktoru	Eliminace těchto rizik
Časová náročnost pro testera při častém testování	Rychlejší, díky softwarovým nástrojům

3 Oxygen

Oxygen (dříve PCTgen) je aplikace vytvořená v rámci bakalářské práce, [8] dále vyvíjená na FEL ČVUT. Jejím hlavním účelem je vytváření zjednodušených modelů testovaných aplikací a to například pomocí diagramu, v Oxygenu nazývaném „diagram aktivit“, ze kterého je možné vygenerovat sadu testovacích scénářů.

3.1 Použití Oxygenu

3.1.1 Správa projektu

Popisovaná aplikace je v Oxygenu zastoupena v podobě projektu. Ten může být uložen ve formátu XML s příponou `.prj`, jenž byla zvolena pro odlišení projektu Oxygenu od ostatních XML souborů. Aplikace jej tedy umí opět načíst a dovoluje tak pokračovat v rozpracovaném projektu nebo upravit již dokončený. Tento soubor bude vyvíjenou aplikací analyzován.

3.1.2 Grafy

K projektu lze přidat několik orientovaných grafů a několik diagramů aktivit. V obou případech je uživatel vytváří pomocí grafického editoru nebo zadáním uzlů a hran do tabulky. Jak orientovaný graf, tak diagram aktivit mají svůj název, počitadlo celkového množství uzlů a hran a také počtu uzlů a hran s konkrétními prioritami, popis, odkaz do dokumentace a verzi. Ke každému uzlu a každé hraně je pak možnost přiřadit název, prioritu a popis. Grafy je možné vyexportovat ve formátech XML, CSV a JSON, což zajišťuje pohodlnou přenositelnost pro další aplikace.

3.1.3 Validace grafů

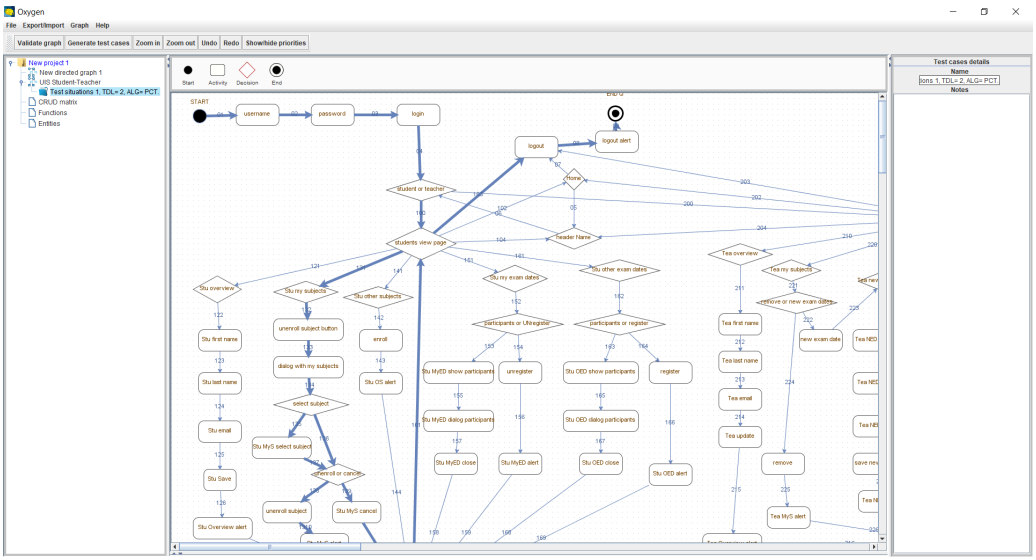
Každý graf musí splňovat určité podmínky, aby bylo možné nad ním provádět požadované operace, například z něj generovat testovací scénáře. Správnost grafu si uživatel může ověřit automatickou validací. Ta kontroluje například tyto náležitosti:

- je přítomný počáteční uzel
- počáteční uzel nemá žádné vstupní hrany

- je přítomný alespoň jeden koncový uzel
- obyčejný uzel má alespoň jednu vstupní hranu
- každý uzel musí být dostupný z počátečního uzlu
- uzly i hrany musí být pojmenovány unikátními jmény
- v diagramu aktivit musí mít uzel *activity node* pouze jednu výstupní hranu
- koncový uzel nesmí mít žádnou výstupní hranu

3.1.4 Testovací scénáře

Po úspěšném zvalidování grafu lze vygenerovat několik testovacích scénářů. Ty jsou vytvořeny např. algoritmem Process Cycle Test (PCT), odtud tedy dřívější název Oxygenu PCTgen. Tato technika je zde pro snazší testování rozšířena o redukci cyklů. Ke každému vygenerovanému testovacímu scénáři aplikace umožňuje zvýraznit průchod grafem.



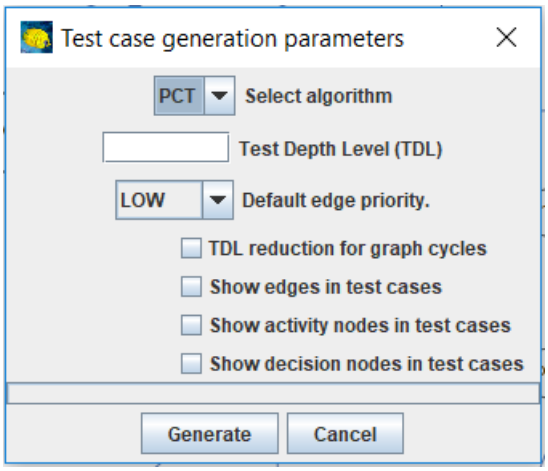
Obrázek 3.1: Vytváření grafu v aplikaci Oxygen

3.2 Algoritmy generování

Pro generování testovacích případů je možné zvolit jeden ze dvou algoritmů generování – Process Cycle Test (dále PCT), nebo Prioritized Process Test

(dále PPT). Dále je nutné těmto algoritmům zadat hloubku testování (Test Depth Level), prioritní úroveň (Priority Level) a výchozí prioritu hran (Default Edge Priority). Před generováním je ještě možné zvolit, zda budou v testovacích případech zobrazeny i hrany, uzly aktivit nebo rozhodovací uzly.[2]

Na obrázku 3.2 je vidět modální okénko, do kterého se tyto hodnoty zadávají. Pro účel vytvářené aplikace je pouze zadán patřičný Test Depth Level, všechny ostatní parametry jsou ponechány ve výchozím stavu.



Obrázek 3.2: Volby pro generování testovacích případů

Konkrétní vygenerovaný testovací případ je reprezentován jako orientovaný graf $G = (N, E)$. [15] Uzly N odpovídají rozhodovacím bodům. Ty reprezentují konkrétní místa aplikace, kde je vyžadována akce uživatele. To, jakým způsobem jsou tyto body definovány, odlišuje dva výše zmíněné algoritmy. U PCT jsou všechny body definovány testerem, na rozdíl od PPT, kde je definován pouze jeden a to výchozí uzel. Hrany E odpovídají průběhu procesu mezi jednotlivými rozhodovacími body. Sekvence těchto po sobě jdoucích hran jsou vygenerované testovací případy.

3.2.1 Test Depth Level

Test Depth Level (TDL) je důležitým faktorem při vytváření testovacích případů. Je definován následovně:
TDL stupně N značí jistotu, že jsou pokryty všechny kombinace N po sobě jdoucích cest. [14]

$TDL = h$, pokud platí, že testovací případy obsahují všechny možné podsekvence h následujících hran grafu. To se dá jinými slovy vyložit tak, že

pro každý uzel (bod) obsahují testovací scénáře všechny hrany (akce) z právě tohoto uzlu.

$TDL = 1$ nastává, pokud pro každou hranu (akci) grafu existuje alespoň jedno využití v testovacím scénáři.

Nicméně konkrétní použití se u dvou výše uvedených algoritmů liší a je nutné je blíže analyzovat.

3.2.2 Process Cycle Test

Prvním algoritmem pro tvorbu procedurálních testů je PCT. Generování testovacích scénářů se skládá ze dvou hlavních částí. V první dochází ke generaci sub-kombinací určených pomocí TDL. Jelikož je délka těchto sub-kombinací limitována hodnotou TDL, nejsou kvůli tomu kompletně pokryté cesty v grafu. Pro generování sub-kombinací je použit algoritmus DFS (grafový algoritmus pro procházení grafů metodou backtrackingu). Ve druhé části dochází ke kompozici testovacích scénářů z identifikovaných sub-kombinací délky rovné TDL.

3.2.3 Prioritized Process Test

Problémem, kterému čelí techniky testování užívající nejdelší cesty grafem[1], je, že vedou k množině testovacích případů, která je příliš rozsáhlá. Naivní redukce může vést k nekontrolované redukci pokrytí a k celkové neefektivnosti. K řešení vede systematická prioritizace a redukce množiny testovacích scénářů. PPT uvažuje rozličné úrovně priorit v testovaném systému. Nejdříve je ovšem nutné rozšířit model SUT, aby obsahoval prioritizaci. Rovněž je zapotřebí definovat tzv. kritérium pokrytí. To bude definováno dále. Nyní už je možné popsat algoritmus PPT.

Graf G bude nyní díky prioritám ohodnocený. Tímto získají hrany E hodnotu: *vysoká, střední a nízká*. Pokud priorita definovaná není, je brána jako *nízká*. Díky tomu vzniká množina hran s vysokou prioritou E_h a množina se střední prioritou E_m. Priority by měly být určeny testovým analytikem. Nyní zbývá už jen určení intenzity testování v konkrétních částech SUT a k tomu jsou použity dvě kritéria pokrytí:

- Prioritized Test Level
- Modifikované TDL algoritmu PCT.

[1] Nejdelší cesta je taková cesta, která by přidáním dalšího uzlu již neodpovídala definici cesty (v rámci cesty se uzly ani hrany nesmí opakovat).

Prioritized Test Level

Toto kritérium může být buďto:

- *vysoké* – pokud platí, že $\forall e \in E_h$, kde se akce e objeví alespoň v jednom testovacím scénáři.
- *střední* – pokud platí, že $\forall e \in E_h \cup E_m$, kde se akce e objeví alespoň v jednom testovacím scénáři.

V tabulce 3.1 je uvedeno modifikované TDL kritérium.

Tabulka 3.1: Definice TDL pro PPT

kritérium pokrytí	**PTL = *vysoké***	**PTL = *střední***
TDL = 1	$P = E_h$	$P = E_h \cup E_m$
TDL = n, n>1	P = množina všech zjištěných cest v grafu G, které začínají jakýmkoliv $e \in E_h$	P = množina všech zjištěných cest v grafu G, které začínají jakýmkoliv $e \in E_h \cup E_m$

Algoritmus produkuje více efektivní testovací scénáře ve smyslu výsledného počtu testovacích kroků, při zachování stejné úrovně pokrytí testovacího systému. [3]

3.3 Vyexportované XML soubory

Ze souborů, které lze pomocí programu Oxygen vyexportovat, jsou pro naši aplikaci nejdůležitější dva. Prvním je soubor popisující celý graf projektu, jeho hrany a uzly, včetně charakteristik ve formátu XML. Druhým souborem je pak seznam testovacích případů. Jak již bylo dříve uvedeno, možnými formáty pro export obou těchto souborů jsou XML, CSV a JSON. Pro potřeby vytvářené aplikace byl vybrán formát XML. Hlavním důvodem takového rozhodnutí je především fakt, že XML soubor lze velmi snadno číst a atributy v něm obsažené analyzovat.

3.3.1 XML s grafem projektu

XML soubor s grafem projektu obsahuje celkem tři typy elementů: `graph`, `node` a `edge`.

Element `graph` je kořenovým elementem a mimo jiné obsahuje atributy `name` s názvem grafu a `type`, který určuje, zda se jedná o diagram aktivit, nebo orientovaný graf. V případě popisu projektu pro vytvářenou aplikaci zde bude typ uvedený vždy jako diagram aktivit (`ACTIVITY_DIAGRAM`).

Hrany jsou zde reprezentovány elementy typu `edge` a uchovávají o sobě informace v podobě atributů `id` (ID komponenty), `name` (jméno), `priority` (priorita, v tomto případě nemusí být při vytváření zadána, není totiž dále zpracovávána), `source` (jméno uzlu, ze kterého hrana vychází) a `target` (jméno uzlu, do kterého hrana směřuje).

Uzly grafu jsou v tomto souboru elementy typu `node`. Atributy, které jsou pro vytvářenou aplikaci důležité, jsou `name`, `style` a `description`. `name` udává jméno uzlu. Pokud se jedná o počáteční nebo koncový uzel, je jeho výchozí hodnota `START`, případně `END`. Hodnota atributu `name` je pak použita v atributech `source` a `target` výše popsaného elementu typu `edge`. Atribut `style` může nabývat čtyř hodnot:

- `STYLE_ACTIVITY_START` – počáteční uzel
- `STYLE_ACTIVITY_NODE` – uzel představující aktivitu
- `STYLE_ACTIVITY_DECISION` – rozhodovací blok
- `STYLE_ACTIVITY_END` – koncový uzel

Velmi důležitým atributem pro budoucí zpracování datasetů je `description`, popsaný dále.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<graph description="" id="2" linkage="" name="New activity diagram
    1" type="ACTIVITY_DIAGRAM" version="">
<node description="" height="25.0" id="2" name="START" priority=""
    style="STYLE_ACTIVITY_START" width="25.0" xpos="430.0"
    ypos="40.0"/>
<edge description="" id="50" name="1" priority="" source="2"
    target="3"/>
<node description="ID=login_nick&#10;type=input&#10;
       TE=TE_dictionary-name&#10;URL=index.php" height="40.0"
           id="3" name="login - nick" priority="(not defined)"
           style="STYLE_ACTIVITY_NODE" width="80.0" xpos="510.0"
           ypos="50.0"/>
<edge description="" id="51" name="62" priority="" source="3"
    target="4"/>
<node description="ID=login_heslo&#10;type=input&#10;
```

```
        TE=TE_passwd&#10;URL=index.php" height="40.0" id="4"
            name="login - heslo" priority="(not defined)"
            style="STYLE_ACTIVITY_NODE" width="80.0" xpos="650.0"
            ypos="100.0"/>
<edge description="" id="52" name="60" priority="" source="4"
    target="49"/>
<node description="" height="35.0" id="13" name="END K"
    priority="" style="STYLE_ACTIVITY_END" width="35.0"
    xpos="390.0" ypos="2230.0"/>
<node
    description="ID=result_text&#10;type=text&#10;URL=web=result"
    height="40.0" id="14" name="result" priority="(not defined)"
    style="STYLE_ACTIVITY_NODE" width="80.0" xpos="370.0"
    ypos="2000.0"/>

</graph>
```

Listing 3.1: Ukázka struktury XML s grafem projektu

3.3.2 XML s testovacími případy

XML soubor s testovacími případy obsahuje pouze dva typy elementů: `test_situations` a `test_situation`.

Element `test_situations` je kořenový a obsahuje dva atributy – `name` a `note`. Ty udávají například zkratku použitého algoritmu nebo hodnotu Test Depth Level, tj. pro generování testovacích datasetů nepodstatné informace.

Důležitějším elementem je `test_situation`. Ten obsahuje jediný atribut `selected_row` značící pořadí testovacího případu, indexovaný od nuly. Obsahem tohoto elementu jsou pak názvy hran oddělené pomlčkou a to v takovém pořadí, v jakém se mají v rámci testovacího případu procházet.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<test_situations name="Test situations 4, TDL= 2, ALG= PCT."
    note="">
<test_situation selected_row="0">1 - 62 - 60 - 61 - 58 - 59 -
    52</test_situation>
<test_situation selected_row="2">1 - 62 - 60 - 57 - 4 - 5 - 6 - 7
    - 8 - 9 - 10 - 100 - 101 - 13 - 1003</test_situation>
</test_situations>
```

Listing 3.2: Ukázka struktury XML s testovacími případy

To, jaké aktivity jsou mezi hranami, lze tedy určit ze souboru projektu. Ve zjednodušené ukázce 3.2 lze vidět jednotlivé vygenerované průchody aplikací. Ty obsahují jednotlivé názvy hran. K určení prováděných akcí je však zapotřebí druhého souboru s grafem celého projektu, který tyto hrany uvádí do souvislostí. Ten lze vidět v ukázce 3.1. Pokud tedy například v souboru s testovými situacemi máme zadáno, že první hrana má název 1, je potřeba hranu s tímto názvem najít i v souboru s grafem projektu, kde jsou elementy hran typu `edge`. Zde je vidět, že tato hrana má atribut `source` roven 2 a atribut `target` 3. Tyto údaje udávají ID počátečního a koncového uzlu této hrany. Elementy uzlů jsou zde typu `node`. Uzel s ID 2 lze tedy jednoznačně určit a podle atributu `style`, který je roven hodnotě `STYLE_ACTIVITY_START` lze poznat, že se jedná o vstupní bod celého grafu. Obdobně lze určit i cílový uzel této hrany, který má ID 3. Protože se jedná o uzel s aktivitou, je už u něj vyplněn atribut `description`, podle kterého již lze zjistit, jaká akce má být testem provedena. Stejným způsobem je postupováno i u dalších hran ze souboru s testovými situacemi.

4 Java Object Populator

Java Object Populator (dále pouze JOP) je nástroj, který vznikl v rámci diplomové práce. [4] Jedná se o knihovnu pro generování testovacích dat za pomoci anotací v jazyce Java. Princip spočívá ve využití již existujících tříd, kterým se dodatečně označkují jejich atributy konkrétními anotacemi, které upřesňují jeho typ a charakteristiku budoucích generovaných dat. To znamená například určit, že se jedná o typ `Integer`, případně pseudonáhodný výběr hodnoty omezit nějakým typem rozdělení, minimální a maximální hodnotou nebo závislostí na jiném atributu.

Druhým způsobem použití JOP je příprava kompletních vlastních tříd, jimž se stejným způsobem anotují atributy již při vytváření třídy. Výhoda tohoto způsobu je, že při generování testovacích dat můžeme využít všech existujících možností JOP. Na druhou stranu nevýhoda tohoto řešení je, že pro generování jednoduchých náhodných dat je tento způsob poněkud komplikovaný.

4.1 Struktura

Knihovna se skládá ze tří hlavních částí – analyzátoru tříd, populátoru objektů a generátorů. Analyzátor slouží ke zpracování tříd a následně k vytvoření jejich zjednodušeného popisu zaměřeného na atributy a jejich anotace. Populátor vytváří kolekce objektů v závislosti na použitých generátorech. Lze pomocí nich například určit třídu nebo minimální a maximální hodnotu atributu vytvářeného objektu.

4.2 Generátory

Generátory slouží ke generování pseudonáhodných hodnot v závislosti na typu konkrétního generátoru a uvedených omezeních. Všechny z nich musejí implementovat rozhraní `ValueGenerator`. To má dva generické parametry. Prvním je typ generované hodnoty a druhý představuje typ anotace. Zároveň toto rozhraní definuje dvě metody – `getValueType`, která vrací typ generovaných dat, a `getValue`, která již vrací vygenerované hodnoty. Parametrem `getValueType` jsou omezení, podle nichž bude generátor vytvářet hodnoty. Pokud během generování dojde k chybě, je navrácena výjimka `ValueGeneratorException`. Některé (základní) generátory jsou již součástí

knihovny, např. generátory čísel podle daných rozdělení, polí hodnot nebo řetězců. Nicméně díky modulárnosti knihovny není problém další doprogramovat.

Podle typu generovaných hodnot lze generátory rozdělit do tří skupin: [4]

- Číselné generátory
  Generování celočíselných hodnot nebo desetinných hodnot.
- Textové generátory
  Vytváření řetězců podle daných pravidel (délka, struktura).
- Ostatní generátory
  Generování logických hodnot, instancí tříd a vlastní generátory.

4.3 Použití

Při práci s JOP není zapotřebí, aby třídy měly přístupné *gettery*, *settery* nebo atributy. Každý atribut může mít několik anotací, ale pouze jednu pro generování hodnot. Pokud není označen žádnou anotací, je ignorován.

4.3.1 Anotace generátorů

Anotace generátorů poskytují hlavní funkčnost JOPu. Mohou být kombinovány s anotacemi populátorů atributů, popsanými níže. Je ovšem potřeba dát pozor na to, aby byly datové typy vygenerovaných hodnot kompatibilní s typy očekávaných populátorem. V opačném případě je navrácena výjimka `PopulatingStrategyException`.

Anotace číselných generátorů

`@BinomialGenerator` je generátor využívající binomické rozdělení. Parametr `trials` udává počet pokusů n a `probability` pravděpodobnost p daného jevu. Výsledkem generátoru je počet úspěšných pokusů, tj. pokusů při nichž nastane daný jev. Pravděpodobnost, že počet úspěšných pokusů bude roven x, je dána funkcí $P(x) = \binom{n}{x} p^x (1-p)^x$.

```
@BinomialGenerator(
    trials = 10,
    probability = 0.6
)
```

Listing 4.1: Ukázka použití `@BinomialGenerator`

`@CategoricalGenerator` slouží ke generování číselných hodnot ze zadané množiny, z nichž každá má uvedenou vlastní pravděpodobnost výskytu. Množina řetězců (tj. generovaných hodnot) je zadávána v podobě pole jako atribut `value`. Pravděpodobnosti těchto číselných hodnot jsou zadávány taktéž v poli (atribut `probabilities`) a to ve stejném pořadí jako číselné hodnoty. Z tohoto vyplývá, že obě pole musejí být stejně dlouhá.

```
@CategoricalGenerator(
    value = {1, 2, 3},
    probabilities = {0.1, 0.6, 0.3}
)
```

Listing 4.2: Ukázka použití `@CategoricalGenerator`

`@ConstantGenerator` je jednoduchý generátor generující konstantně tu samou číselnou hodnotu (`value`).

```
@ConstantGenerator(
     value = 10
)
```

Listing 4.3: Ukázka použití `@ConstantGenerator`

`@DiscreteUniformGenerator` generuje diskrétní hodnoty z daného intervalu (ohraničeného hodnotami `min` a `max`) na základě rovnoměrného rozdělení, které přiřazuje všem hodnotám stejnou pravděpodobnost $\frac{1}{\texttt{max}-\texttt{min}+1}$. Diskrétností zde rozumíme skutečnost, že generuje pouze celá čísla.

```
@DiscreteUniformGenerator(
     min = 100,
     max = 180
)
```

Listing 4.4: Ukázka použití `@DiscreteUniformGenerator`

`@ExponentialGenerator` je generátor využívající k určení pseudonáhodných hodnot pravděpodobnostní rozdělení podobné exponenciálnímu s hustotou pravděpodobnosti $f(x) = \lambda e^{-\lambda x}$. Na rozdíl od klasického exponenciálního rozdělení lze použít i záporné λ a argumenty hustoty pravděpodobnosti jsou pouze nenulové. Parametr λ je zde označen jako `rate`.

```
@ExponentialGenerator(
    rate = 6
)
```

Listing 4.5: Ukázka použití `@ExponentialGenerator`

`@GaussianGenerator` generuje hodnoty dané normálním (neboli Gaussovým) rozdělením s funkcí hustoty $f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\mu^2}}$. Její hustotu je tedy nutno jednoznačně určit pomocí střední hodnoty μ (`mean`) a rozptylu σ^2 (`variance`).

```
@GaussianGenerator(
     mean = 8,
     variance = 2
)
```

Listing 4.6: Ukázka použití `@GaussianGenerator`

`@PoissonGenerator` využívá ke generování hodnot pravděpodobností získaných Poissonovým rozdělením $P(x) = \frac{\lambda^x}{x!} e^{-x}$. Toto rozdělení vyjadřuje počet jevů v určitém intervalu, tudíž generované pseudonáhodné hodnoty jsou pouze celá čísla včetně nuly. Parametr `mean` značí jeho střední hodnotu λ, přičemž $\lambda > 0$.

```
@PoissonGenerator(
    mean = 5
)
```

Listing 4.7: Ukázka použití `@PoissonGenerator`

`@UniformGenerator` je generátor podobný `@DiscreteUniformGenerator`. Také zde se generují hodnoty dané rovnoměrným rozdělením v intervalu ohraničeném parametry `min` a `max`, tedy s hustotou pravděpodobnosti $\frac{1}{\texttt{max}-\texttt{min}}$. Rozdíl je však v tom, že tento generátor není diskrétní. To zde znamená, že generuje reálná čísla.

```
@UniformGenerator(
    min = 126.7,
    max = 130.1
)
```

Listing 4.8: Ukázka použití `@UniformGenerator`

Anotace textových generátorů

`@ConstantString` je generátor podobný `@ConstantValue`, jen slouží ke generování nikoliv číselné, ale textové hodnoty, která je stále stejná. Je určena parametrem `value`.

```
@ConstantString(
    value = "string"
)
```

Listing 4.9: Ukázka použití `@ConstantString`

`@DictionaryGenerator` pseudonáhodně generuje textové řetězce ze zadaného slovníku, který je vytvořen uživatelem. Jedná se o textový soubor, jehož jméno a cesta je uvedena v `path`. V tomto slovníku jsou jednotlivé řetězce na samostatných řádkách. To znamená, že pokud je na jedné řádce vícero slov, je tento řádek brán celý jako jeden řetězec včetně mezer. Je možné též zadat kódování, a to pomocí atributu `encoding`. Výchozí hodnotou je UTF-8.

```
@DictionaryGenerator(
    path = "./dictionary.txt"
)
```

Listing 4.10: Ukázka použití `@DictionaryGenerator`

`@RandomString` je textovou obdobou `@CategoricalGenerator`. Slouží ke generování textových řetězců ze zadané množiny (`value`), z nichž každý má uvedenou vlastní pravděpodobnost výskytu (`probabilities`).

```
@RandomString (
    value = {"string1", "string2", "string3"},
    probabilities = {0.3 , 0.5 , 0.2}
)
```

Listing 4.11: Ukázka použití `@RandomString`

Anotace ostatních generátorů

`@ConstantBoolean` je generátor, jenž funguje na podobném principu jako `@ConstantGenerator` a `@ConstantString`. Generuje stále stejnou booleovskou hodnotu, v anotaci označenou jako `value`.

```
@ConstantBoolean(
    value = true
)
```

Listing 4.12: Ukázka použití `@ConstantBoolean`

`@RandomBoolean` slouží ke generování náhodných booleovských hodnot, tedy hodnot *true*, nebo *false*. Parametr `probability` určuje pravděpodobnost výskytu hodnoty *true*.

```
@RandomBoolean(
    probability = 0.2
)
```

Listing 4.13: Ukázka použití `@RandomBoolean`

`@RandomClass` je anotace podobná výše uvedeným `@CategoricalGenerator` a `@RandomString`. Používá kategorický generátor pro výběr třídy ze zadané množiny (`value`). Pokud jsou uvedeny pravděpodobnosti výskytu, musejí být uvedeny pro všechny třídy z množiny (`probabilities`).

```
@RandomClass(
    value = {First.class, Second.class, Third.class},
    probabilities = {0.2, 0.5, 0.3}
)
```

Listing 4.14: Ukázka použití `@RandomClass`

`@RandomClassForName` je anotace založená na velmi podobném principu jako

`@RandomClass`. Hlavní rozdíl je v tom, že parametr `value` vyžaduje plně kvalifikovaná jména tříd jako textové řetězce datového typu `java.lang.String`. Výhodou předávání plně kvalifikovaného jména spočívá ve skutečnosti, že v tomto případě nemusí být třída v době kompilace přítomná na `classpath`. Parametr `probabilities` je stejný jako v předchozím případě. Dále je možné ještě zvolit, zda bude při generování instance inicializována, a to pomocí parametru `initialize`, jehož výchozí hodnotou je *true*. Posledním parametrem tohoto generátoru je `classLoader`. Ten může nabývat těchto hodnot:

- `CALLER` – implicitní hodnota, získává `ClassLoader` pomocí `ClassLoaderUtils.getCallerClassLoader()`
- `CONTEXT` – `Thread.currentThread().getContextClassLoader()`
- `SYSTEM` – `ClassLoader.getSystemClassLoader()`
- vlastní textový řetězec reprezentující instanci třídy `ClassLoader`, jenž má být použita

```
@RandomClassForName(
    value = {"Instance.First", "Instance.Second"},
    probabilities = {0.2, 0.8}
)
```

Listing 4.15: Ukázka použití `@RandomClassForName`

`@TargetClass` se využívá ke generování instance stále té samé třídy, předané parametrem `value`.

```
@TargetClass(
    value = First.class
)
```

Listing 4.16: Ukázka použití `@TargetClass`

`@TargetClassForName` slouží ke stejnému účelu jako `@TargetClass`. Generovaná třída je zde však předána pomocí parametru `value` plně kvalifikovaným jménem. Zbylé dva nepovinné parametry `initialize` a `classLoader` pak fungují stejně jako u výše popsané `@RandomClassForName`.

```
@TargetClassForName(
    value = "Instance.First"
)
```

Listing 4.17: Ukázka použití `@TargetClassForName`

Vlastní generátory

Vzhledem k modulárnosti aplikace lze jednoduše přidat vlastní generátory. Každý vlastní generátor musí dědit od třídy `AbstractValueGenerator` a implementovat metody `getValueType()` a `getValue()`. Atribut, který pak tento generátor používá, je označen anotací `@CustomValueGenerator`, jež má jediný parametr značící třídu nového generátoru. Pokud má třída s generovaným atributem tento atribut i jako vstupní parametr populátorem používaného konstruktoru, musí i zde být tento vstupní atribut opatřen anotací. Datové typy parametrů je nutné uvést v generickém parametru rozhraní generátoru a jako parametry anotace generovaného atributu.

Pokud tento vlastní generátor nevyžaduje žádné vstupní parametry, je vhodné v generickém parametru použít anotaci `@EmptyParameters`. [4]

4.3.2 Anotace populátorů

Populátory slouží především k osazení hodnoty do atributu a určení datového typu, ale mohou být využity například i k jeho formátování. [4] Každý atribut může být označen libovolným množstvím populátorů, zde je však potřeba dát pozor na to, zda jsou mezi sebou dané populátory kompatibilní, a určit jejich správné pořadí. Způsob určení pořadí populátorů je uveden dále. V závislosti na konkrétním použití může mít populátor vlastní parametry nebo je uveden bez paramentrů. Omezení daná atributy generátoru mají vždy přednost před omezeními danými atributy generátorů.

Číselné hodnoty jsou anotovány populátorem `@NumberValue`. Ten může mít tři parametry – `min`, `max` a `target`. `min` určuje minimální hodnotu, `max` maximální hodnotu. Pokud tyto hodnoty vymezují interval, který nemá s intervalem hodnot vymezených generátorem žádné společné hodnoty, je generována taková hodnota z intervalu populátoru, jenž se nejvíce blíží intervalu generátoru. Pomocí `target` lze zadat požadovaný datový typ, ale v tomto případě pouze takový, jenž dědí od `java.lang.Number`.

```
@NumberValue(target = Double.class, min = 100, max = 250)
```

Listing 4.18: Ukázka použití `@NumberValue`

Pro textové hodnoty je připravena anotace `@StringValue`. Zde jsou možnými parametry `minLength`, `maxLength`, `length`, `fill` a `target`. `minLength` značí minimální délku řetězce, `maxLength` maximální. `length` udává přesnou požadovanou délku řetězce. Pokud jsou vyplněny všechny tyto tři atributy, ale `length` není součástí intervalu ohraničeného `minLength` a `maxLength`, je použit pouze parametr `length`. Pokud je `minLength` nebo `length` delší než vygenerovaná hodnota, je použit jako výplň zbylých míst parametr `fill`. Jeho výchozí hodnotou je znak mezera. Datový typ daný parametrem `target` musí dědit od `java.lang.CharSequence`.

```
@StringValue(length = 10, fill = '0', target = String.class)
```

Listing 4.19: Ukázka použití `@StringValue`

`@ArrayValue` je případ anotace populátoru, který se využívá pro generování polí. Lze k ní připojit čtyři parametry – `minLength` (minimální délka), `maxLength` (maximální délka), `length` (přesná délka) a `target` (datový typ). V případě kombinování `minLength`, `maxLength` a `length` platí totéž, jako v předchozím případě. Parametr `target` pak lze využít pro určení cílového datového typu pole. V tomto případě je ideální atribut anotovat ještě populátorem daného cílového datového typu a použít anotaci `@PropertyPopulatorsOrder`, jež je popsaná dále.

```
@ArrayValue(minLength = 1, maxLength = 10, target = String.class)
@StringValue
```

Listing 4.20: Ukázka použití `@ArrayValue` spolu se `@StringValue`

4.3.3 Ostatní anotace

Při použití více populátorů najednou může dojít, pokud to v onom konkrétním případě implementace dovoluje, k jejich zřetězení. Vzhledem k tomu, že pořadí jejich vyvolání nelze určit z pořadí v jakém jsou uvedeny ve zdrojovém kódu, je k dispozici anotace `@PropertyPopulatorsOrder`, která jej pevně určuje.

```
@PropertyPopulatorsOrder({
      ArrayValue.class,
      NumberValue.class
  })
```

Listing 4.21: Ukázka použití `@PropertyPopulatorsOrder`

U tříd, které mají více konstruktorů nebo tovární metodu, je potřeba označit jeden konstruktor nebo tovární metodu anotací `@Constructor`, aby bylo zabráněno nejednoznačnosti při vytváření objektů.

Pokud pro nějaký atribut není vyžadováno generování testovacích hodnot, lze jej nechat bez anotace a v tom případě bude JOPem ignorován. Další možností, pokud je například vyžadováno mít všechny atributy anotované, je využití `@Ignore`. Tato anotace bude mít stejný účinek, tzn, atribut bude při generování ignorován.

Obdobnou funkcionalitu má `@NullValue`. S tím rozdílem, že sice nebude vygenerována žádná nová hodnota, ale atribut bude nastaven buď na `null`, nebo, v případě primitivních datových typů, na jeho výchozí hodnotu.

Spolu s použitím anotací pro poskytovatele tříd, jako jsou například `@RandomClass` nebo `@TargetClassForName`, je možné pro vytvoření nové instance použít anotaci `@NewInstance`. Ta se velmi podobá anotacím populátorů a jedná se v podstatě o jakýsi „populátor“ pro třídy, implementačně se však liší.

V případě použití této anotace se všechny vygenerované instance ukládají do společné kolekce *session*. S využitím anotace `@SearchInstance` lze tuto *session* vytvořených instancí prohledat a případě do generovaného atributu dosadit vhodnou z nich.

Anotaci `@CustomInstanceMatcher` lze použít najednou spolu s výše uvedenou `@SearchInstance`. Používá se k určení instance třídy, jež vyhovuje zadaným kritériím. Třída, jejíž instance jsou hledány, musí implementovat rozhraní `@InstanceMatcher`.

Pro určení způsobu přístupu k samotným atributům je vytvořena anotace `@Access`. Lze ji použít jak pro celou třídu, tak pro jednotlivé atributy zvlášť. Pomocí jejího parametru lze přístup nastavit buď na `AccessType.PROPERTY`, nebo na `AccessType.FIELD`. Výchozí hodnotou je `AccessType.PROPERTY`, která značí přístup k atributům pomocí *getterů* nebo *setterů*, které však nemusejí být přístupné veřejně. Při použití parametru `AccessType.FIELD` je k atributům přistupováno reflexí. Atribut nemusí být přístupný veřejně a v případě zápisu nesmí být deklarován jako `final`. [4]

4.4 JOP Demo

V rámci studia knihovny Java Object Populator byla vytvořena aplikace demonstrující její možnosti a správný způsob jejího používání. Je členěna do modulů podle použitých populátorů, případě dle ostatních anotací, jako je například `@EmptyParameters`. V každém z těchto modulů je ukázáno jejich

použití s vhodnými generátory, přičemž v rámci projektu jsou zastoupeny všechny generátory knihovny. Každý modul má rovněž třídu, která slouží k populování atributů zbylých tříd z modulu.

Během vytváření této demonstrační aplikace bylo zároveň zjištěno, že v rámci knihovny existuje několik připravených anotací, pro něž není doplněna funkční implementace. To může být značně matoucí, jelikož lze bez problémů přeložit třídy opatřené těmito anotacemi, ale při pokusu o populování je navrácena výjimka `ObjectPopulatorException`. Jedná se například o anotace generátorů `@MarkovChain`, `@RegularExpression` nebo populátoru `@CollectionValue`.

Níže je uvedena ukázka generování textových řetězců za pomoci anotace `@ConstantStringGenerator`. V ukázce kódu 4.22 je třída s atributem, jenž je opatřen anotací populátoru `@StringValue` a anotací generátoru `@ConstantString` s parametrem `value`.

Objekty této třídy jsou populovány v `SV_Main`, kterou lze vidět v ukázce kódu 4.23. Děje se tak pomocí instance populátoru `ObjectPopulator`. Ta přímo využívá svoji metodu `populate`, jenž vrací kolekci požadovaných objektů a dané délky.

```
import cz.zcu.kiv.jop.annotation.Access;
import cz.zcu.kiv.jop.annotation.AccessType;
import cz.zcu.kiv.jop.annotation.generator.string.ConstantString;
import cz.zcu.kiv.jop.annotation.populator.StringValue;

@Access(AccessType.FIELD)
public class SV_ConstantStringGenerator {

  @StringValue
  @ConstantString(
      value = "string"
  )
  private String attribute;

  @Override
  public String toString() {
    return "" + attribute;
  }
}
```

Listing 4.22: Třída využívající `@ConstantStringGenerator`

```
import cz.zcu.kiv.jop.ObjectPopulator;
import cz.zcu.kiv.jop.ObjectPopulatorProvider;
import java.util.List;

public class SV_Main {

    public static final int MAX = 50;

    public static void main(String[] args) throws Exception {
        ObjectPopulator populator =
            ObjectPopulatorProvider.getObjectPopulator();

        List<SV_ConstantStringGenerator> constantStringValues =
            populator.populate(SV_ConstantStringGenerator.class,
            MAX);
        System.out.println(constantStringValues.get(0)
                                .getClass().getName());
        for (SV_ConstantStringGenerator cc : constantStringValues) {
            System.out.print(cc + ", ");
        }
    }
}
```

Listing 4.23: Třída s populátorem

Další ukázky použití JOP jsou součástí přiloženého CD.

4.5 Možnosti využití pro testování webových aplikací

Pro testování webových aplikací je mimo jiné potřeba zajistit data, která budou do webové aplikace zadávána. Myšlenka využití JOP pro testování webových aplikací tedy vychází z předpokladu, že takováto data poskytuje a zároveň se jedná o již hotovou snadno použitelnou knihovnu, tudíž odpadne nutnost vytvářet vlastní generátory pseudonáhodných hodnot. Před vytvářením aplikace pro generování testovacích datasetů pro webové aplikace je však důležité tyto předpoklady analyzovat a ověřit.

Jak již bylo ukázáno v předchozích kapitolách, JOP obsahuje mnoho nástrojů pro pseudonáhodné generování různých sad hodnot. Je však typické, že tyto hodnoty jsou často typově stejné. I přes velké množství generátorů lze generovat převážně pouze textové, číselné, booleovské datové typy nebo

instance javovských tříd. Liší se povětšinou pouze pravděpodobnostním rozdělením.

Pro testování webových aplikací je potřeba vygenerovat hodnoty představující čísla, textové řetězce. Naopak například generátory instancí objektů by v tomto případě jistě zůstaly nevyužity. Přesný formát hodnot pro webové aplikace je rovněž většinou určen minimálními a maximálními mezemi nebo slovníkem obsahujícím výčet povolených řetězců. Z tohoto vyplývá, že při použití JOPu pro testování webových aplikací by stejně většina již implementovaných generátorů zůstala nevyužita a bylo by navíc potřeba stávající knihovnu doplnit dalšími generátory, například datovým.

Pro praktické použití je zároveň nutné, aby tato omezení mohl zadávat sám uživatel bez nutnosti měnit zdrojový kód. Ideální řešení je tedy využití externích tříd ekvivalence, jenž by mohl upravovat i přidávat sám uživatel. Data z těchto tříd ekvivalence by pak byla načtena jako parametry generátorů a na základě nich vygenerovány nové hodnoty. To znamená, že by tyto hodnoty byly dosazovány až za běhu programu.

Knihovna JOP využívá pro označení generovaných atributů anotace a omezení generovaných hodnot jsou zadávána jako parametry těchto anotací. Anotace v jazyce Java jsou však navržené tak, že musejí být včetně hodnot parametrů jednoznačné již při překladu kódu. To znamená, že v programu obsahujícím atributy anotované generátory nelze za běhu měnit parametry těchto generátorů a tím pádem ani upravovat omezení pro generované hodnoty.

Jediným možným řešením, jak lze využít této knihovny pro generování testovacích datasetů pro webové aplikace, je tedy v rámci vytvářené aplikace generování celých tříd s již s pevně stanovenými hodnotami parametrů anotací a následný překlad a spuštění tohoto kódu. Jeho výstupem by byly pouze vygenerované hodnoty, které by byly dále použity aplikací jako součást testovacích datasetů.

Toto řešení je však značně komplikované. Vzhledem k tomu, že by nebyl ani dostatečně využit potenciál většiny generátorů, bylo po četných pokusech rozhodnuto, že tato knihovna nebude v rámci projektu použita.

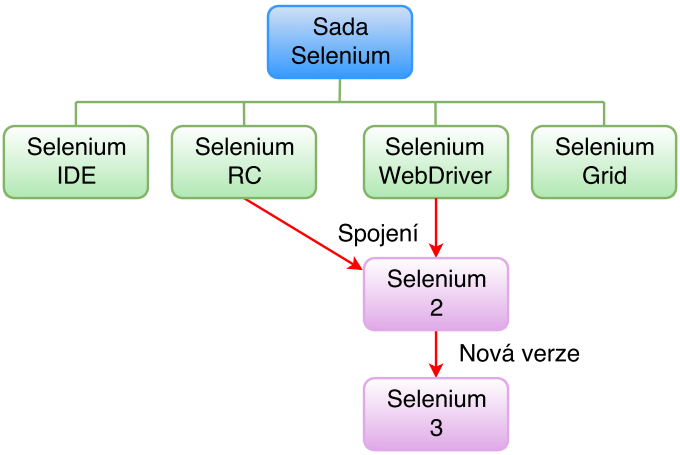
5 Selenium

Selenium je framework pro automatizované testování webových aplikací běžících na rozdílných prohlížečích a platformách. Jedná se o volně dostupnou sadu, která je složena z několika rozličných nástrojů. Každý přistupuje k testování odlišným způsobem a představuje tak široké spektrum specificky zaměřené pro testování webových aplikací rozličných druhů. Jedná se konkrétně o čtyři komponenty:

- **Selenium WebDriver**
- **Selenium IDE**
- **Selenium Remote Control (RC)**
- **Selenium Grid**

Nicméně verzí Selenia 2.0 došlo ke sloučení nástrojů RC a WebDriver. Tímto se Selenium 2 (nyní již verze 3) stalo komplexnějším nástrojem s větší funkčností. Verze Selenium 1 (Selenium RC) je označena za zastaralou, avšak vývojáři stále udržovanou. [11]

Na obrázku 5.1 je toto přehledně ukázáno.



Obrázek 5.1: Sada Selenium

5.1 Selenium IDE

Selenium IDE je nástroj k vytváření vlastních seleniových testovacích scénářů, díky kterému je možné je vytvářet i bez hlubší znalosti jejich zápisu.

Jedná se plugin pouze pro prohlížeč Mozilla Firefox, ostatní nejsou podporovány. Obsahuje kontextovou nabídku, za jejíž pomoci lze vybrat konkrétní element stránky právě zobrazené v prohlížeči, a přidat k ní patřičný příkaz s parametry. Navíc rovněž obsahuje funkci pro „nahrávání akcí“. Takto lze zaznamenat posloupnost akcí tak, jak jsou testerem prováděny, a následně je možné je vyexportovat jako znovupoužitelný kód v programovacích jazycích Java, C#, Python a Ruby. Tímto způsobem je vytváření testů časově snazší a také představuje vhodný úvod pro seznámení se se syntaxí seleniových skriptů. [12]

5.2 Selenium Remote Control

Tento nástroj umožňuje testerovi napsat automatizované testy v mnoha programovacích jazycích pro jakoukoli HTTP stránku zobrazenou prohlížečem podporující JavaScript. Skládá se ze dvou částí:

1. **Selenium RC Server**, který automaticky zapíná/vypíná jednotlivé instance prohlížečů a funguje jako HTTP proxy[1] pro webové žádosti
2. **Selenium Core**, poskytující klientské knihovny určené pro konkrétní programovací jazyk

RC tedy funguje na principu propojení klientské knihovny a seleniového serveru. Ten nastartuje nebo použije běžící prohlížeč s konkrétní URL adresou, která je pro spojení nastavena. Klientské knihovny posílají serveru seleniové příkazy. Ty jsou následně serverem interpretovány a spuštěny. [5]

5.3 Selenium Grid

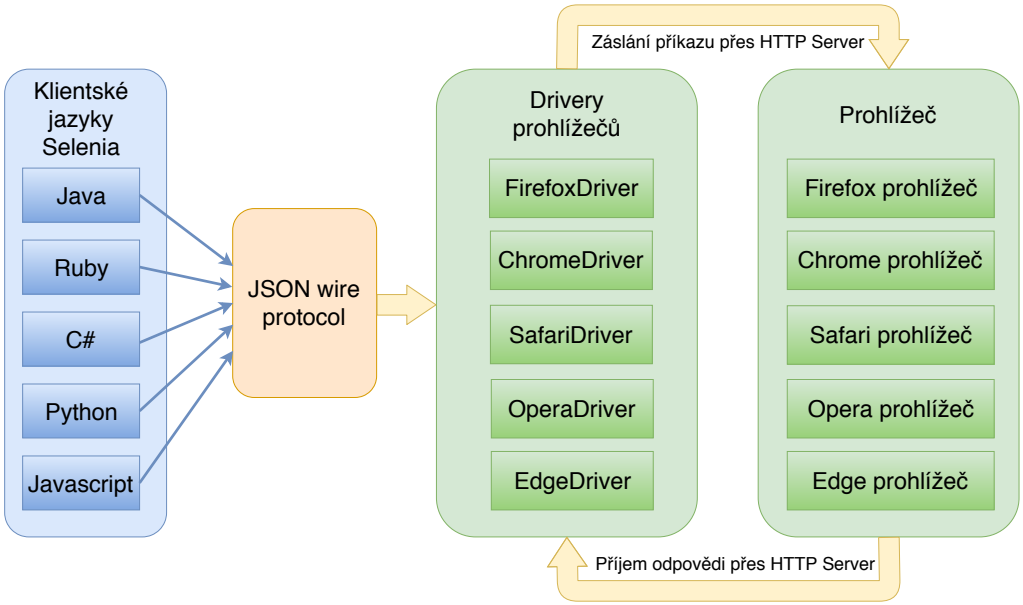
Selenium Grid umožňuje sadě RC a Seleniu 2 škálování pro rozsáhlejší testovací scénáře. Ty mohou probíhat jak na různých operačních systémech, tak i na rozdílných verzích webového prohlížeče a to v paralelním běhu. To znamená, že odlišné testy mohou běžet ve stejný čas na různých fyzických či virtuálních strojích. Jeho hlavní výhodou je tedy škálování testovacích scénářů a úspora značného množství času. [12]

[1] Proxy server – Prostředník mezi klientem a serverem. Pro klienta se tváří jako server.

5.4 Selenium WebDriver

Selenium WebDriver je navržen tak, aby kromě řešení některých omezení v rozhraní API Selenium RC poskytoval jednodušší a stručné programovací rozhraní. Tento nástroj byl rovněž vyvinut pro lepší podporu dynamických webových stránek, kde se prvky stránky mohou měnit, aniž by byla stránka znovu načtena. Cílem aplikace WebDriver je poskytnout dobře navržené objektově orientované rozhraní API, které poskytuje lepší podporu moderním, pokročilým problémům s testováním webových aplikací.

Hlavní změnou oproti Seleniu RC je možnost přímého volání prohlížeče pomocí přirozené podpory každého prohlížeče [12]. To je schematicky zobrazeno na obrázku 5.2.



Obrázek 5.2: Architektura Selenium WebDriver

Je možné si povšimnout čtyř komponent, které jsou ve WebDriveru obsaženy. Selenium podporuje mnoho programovacích jazyků díky již vyvinutým jazykovým vazbám ke konkrétním jazykům. Pod pojmem JSON Wire Protocol se skrývá REST rozhraní, které přeposílá informace uvnitř HTTP serveru. Každý *driver* prohlížeče má svůj vlastní HTTP server a komunikuje jen s příslušným prohlížečem. Když *driver* obdrží jakýkoliv příkaz, bude tento příkaz spuštěn v příslušném prohlížeči a odpověď se vrátí zpět ve formě odpovědi HTTP [13].

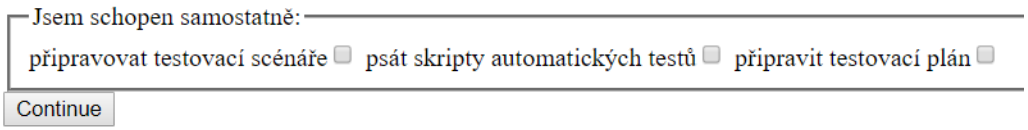
6 Testovací aplikace

K otestování výsledné aplikace generující testovací datasety bylo nutno použít webovou aplikaci, která je relativně jednoduchá a zároveň je zde možnost ji důkladně zanalyzovat.

Prvním problémem byl tedy samotný výběr aplikace. Nejdostupnější možnost, běžné webové stránky, jsou pro takový účel nevhodné. Už jen z toho důvodu, že pro potřeby vytvářené aplikace je zapotřebí mít u všech testovaných elementů vlastní ID, což drtivá většina z nich nesplňuje.

Řešením je tedy použít nějakou „uměle“ vytvořenou aplikaci, která bude splňovat dané podmínky. Vzhledem k tomu, že je v současné době na ZČU v rámci diplomové práce vyvíjena aplikace určená přímo pro testování testovacích metod, bylo zprvu rozhodnuto, že testování bude probíhat právě na ní. Jedná se o aplikaci napodobující reálný informační systém univerzity University Information System (UIS). Bylo tedy nutné tuto aplikaci analyzovat a případně navrhnout některé úpravy. Při její analýze a následném znázornění jako graf v aplikaci Oxygen bylo zřejmé, že pro průzkumné účely je tato aplikace příliš složitá. Testovacích případů je zde velmi mnoho a příliš dlouhé. Dalším problémem je zde velká míra akcí, kde je zapotřebí pouze klikání na tlačítka, případně výběry z comboboxů. To by znamenalo degradaci průzkumné práce téměř pouze na dva typy ovládání, tedy klikání a výběry z comboboxů.

Z těchto důvodů byla nad rámec zadání bakalářské práce vytvořena v jazyce PHP vlastní webová aplikace, která je jednoduchá, s limitovaným počtem stavů a přechodů a se všemi běžnými ovládacími prvky. Je záměrně graficky velmi jednoduchá, bez jakéhokoliv stylování, protože jejím účelem je jen připravit dobře testovatelnou fungující aplikaci. Ukázka jedné její stránky je na obrázku 6.1.



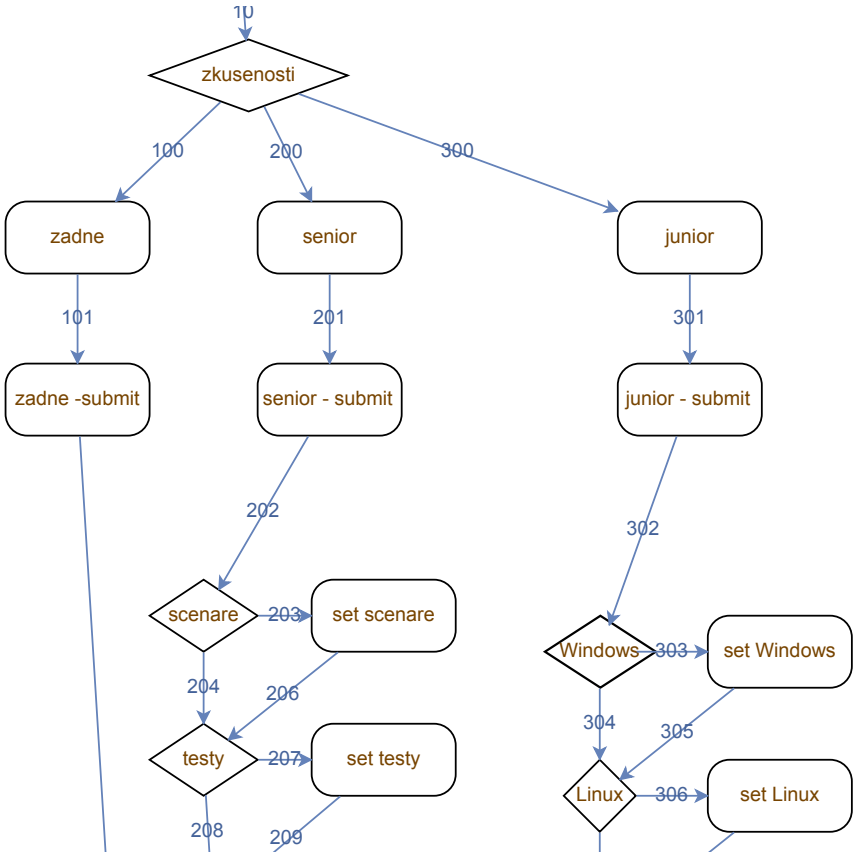
Obrázek 6.1: Ukázka testovací aplikace

Obsahuje celkem osm samostatných stránek, 32 stavů a 44 přechodů. Ukázka stavů a přechodů zakreslených v aplikaci Oxygen je na obrázku 6.2. Zde je vidět několik kosočtvercových rozhodovacích uzlů a obdélníkových

uzlů s aktivitami.

Například rozhodovací uzel s názvem *zkusenosti* představuje náhodný výběr mezi úrovněmi zkušeností *zadne*, *senior* a *junior*. Tyto úrovně zkušeností jsou zde zobrazeny jako uzly aktivit pro vybrání příslušného radioboxu v aplikaci. V případě radioboxů se tedy volí pouze jedna z nabízených možností.

Problém však nastává v případě checkboxů, kdy lze zvolit libovolný počet položek. To bylo nakonec vyřešeno použitím rozhodovacího uzlu pro každý checkbox. Z něj vedou dvě hrany. Jedna na uzel představující zaškrtnutí checkboxu a druhá na další akci. Na obrázku 6.2 lze tuto situaci vidět například u dvojice uzlů *scenare* a *set scenare*.



Obrázek 6.2: Část diagramu popisující testovací aplikaci

7 Rozšíření formátu atributu `description`

Atribut `desciption` obsahuje v případě uzlu představujícího aktivitu speciálně strukturovanou informaci o povaze popisovaného objektu. Tu při tvorbě grafu v aplikaci Oxygen zadává do textového pole a formátuje sám uživatel. To je sice poměrně uživatelsky nepohodlné a je zde velká pravděpodobnost lidské chyby, ale na druhou stranu současná implementace Oxygenu jinou možnost předání těchto informací nedovoluje. Formát tohoto atributu bylo zapotřebí pevně určit, jelikož bude v rámci vytvářené aplikace parsován na jednotlivé informace, podle nichž se budou datasety generovat. Jedním z cílů práce bylo zjištění, jaké typy údajů je zde nutné uvést. Toto bylo zjištěno množstvím experimentů i na původní testovací aplikaci představující univerzitní informační systém.

V budoucnu by bylo vhodné, aby byla možnost tyto atributy zadávat pomocí např. comboboxů, jelikož by už při zadávání mohly být validovány a omezila by se tak možnost lidské chyby. Takto určené atributy by byly součástí exportovaného XML v podobě jeho dalších XML atributů. V současné době však Oxygen tuto možnost nenabízí a je nutné se spolehnout na předávání informací pomocí `description`.

Prvním úkolem bylo stanovení atributů, jež budou k testování nezbytné, nebo alespoň užitečné. Jedním z těch nezbytných je ID webového elementu, se kterým má být operováno, jelikož se jedná o jeho jediný jednoznačný identifikátor. V rámci `description` bude mít název `ID`. Tím ale zároveň vzniká potřeba mít v testované webové aplikaci všechny potřebné elementy opatřeny právě tímto atributem, což mnoho webových aplikací nesplňuje. To lze však vyřešit jejich doplněním do HTML.

Další nepostradatelnou informací je typ elementu. K tlačítkům, vstupním polím nebo zaškrtávacím okénkům bude přistupováno rozdílně a jen pro některé typy budou generována náhodná testovací data. Pro ostatní bude pouze vygenerována akce s nimi se vážící, jako je například pro tlačítko kliknutí. Tato informace bude v `description` reprezentována jako `type`.

S generováním náhodných testovacích dat se dostáváme k otázce, jaká data přesně generovat. Je rozdíl, zda se jedná o data představující například e-mail, datum, uživatelské jméno, věk nebo heslo, ačkoliv pro zadání všech z nich může být použitý pouze jeden typ elementu, textové pole. Typ elementu tak stačí ke stanovení typu akce, která se nad ním má provést, ni-

koliv však k určení formátu generovaných náhodných dat. Vzhledem k tomu, že stejný formát dat lze vyžadovat u více elementů, je vhodné tuto společnou definici vyčlenit mimo atribut `description` a popsat ji pomocí třídy ekvivalence odkazované z tohoto atributu. Ty budou tedy podle daných konvencí vytvářeny samotným uživatelem a dále zpracovávány vytvářeným programem. Z důvodu jednoduchosti vytváření a čtení byl zvolen formát souboru `.txt`. Přesný formát těchto tříd ekvivalence bude popsán později. Do `description` pak bude uváděn pouze název této třídy ekvivalence neboli název odpovídajícího souboru a to pod názvem `TE`.

Ne vždy je ale vyžadováno, aby byly pro dvě vstupní pole generovány dvě rozdílné hodnoty. Klasickým příkladem mohou být například pole pro heslo a pro ověření hesla, kde se očekává, že uživatel v obou případech zadá stejnou hodnotu. Z tohoto důvodu bylo potřeba přidat nový parametr `sameAs`, který by v případě takovéhoto požadavku nahrazoval parametr s názvem třídy ekvivalence, protože ta v tomto případě není potřebná, a odkazoval by na prvek, se kterým má hodnotu sdílet. Tento odkaz musí být jednoznačný a proto je na místě zde použít rovněž ID elementu, které toto splňuje.

Formát atributu `description` byl určen tak, aby jej bylo zároveň snadné zadávat i následně parsovat. Názvy atributů jsou vždy na začátku řádky a jejich hodnoty jsou od nich odděleny symbolem `=`. Je důležité, aby uživatel tuto syntaxi dodržoval, jinak nebude tento atribut správně naparsován a datasety se buď nevygenerují nebo se vygenerují chybné.

```
ID=login_nick
type=input
TE=TE_name
```

Listing 7.1: Ukázka atributu `description`

8 Implementace generátoru testovacích sad

Aplikace se skládá z modulu hlavní aplikace a modulu `selenium-tests`. Lze ji však rozdělit do těchto funkčních celků:

- třída `XMLParser` – Zpracovává XML soubory a extrahuje z nich potřebné informace.
- třída `DescriptionParser` – Představuje instance elementů se všemi jejich náležitostmi, případně volá patřičné generátory.
- generátory – Slouží ke generování hodnot podle zadaných omezení.
- třídy `TestsMaker` a `TestSuiteMaker` – Generují javovské třídy – `TestsMaker` s jednotlivými testy pro Selenium a `TestSuiteMaker` s test suite pro spouštení všech testů najednou.
- modul `selenium-tests` – Obsahuje rodičovské třídy pro seleniovské testy, test suite a konfigurační soubor. Zároveň jsou do tohoto adresáře tyto testy generovány.

8.1 Zpracování XML souborů

Aplikace vyžaduje dva vstupní XML soubory poskytované Oxygenem, soubor se strukturou projektu a soubor s vygenerovanými testovacími případy. Bylo proto potřeba zvolit vhodnou technologii jejich zpracování.

Dvěma základními technologiemi, které Java nabízí pro práci s XML soubory, jsou následující:

- **DOM Parser/Builder** – Načte do paměti celý XML dokument a vytvoří jeho objektovou reprezentaci. O parsování souboru se stará metoda `parse()` třídy `DocumentBuilder`. Jejím výstupem je instance třídy `Document`, jenž má stromovou strukturu. Každý uzel implementuje rozhraní `Node` a může být různého charakteru, například reprezentující elementy XML dokumentu s atributy nebo textový obsah ohraničený startovací a ukončovací značkou elementu.
- **SAX Parser** – Na rozdíl od předchozího případu jej lze využít pouze pro čtení dokumentu, nikoliv pro zápis. Zároveň nenačítá do paměti

celou strukturu dokumentu, ale pouze jeho potřebnou část pomocí *callbacků*, jež jsou volány třídou `SaxParser`. Tento způsob přístupu se nazývá *streaming XML*. Třída `ContentHandler` je pro vývojáře rovněž důležitá. Poskytuje metody pro práci se začátkem a koncem zpracování dokumentu (`startDocumnent()` a `endDocument()`) i jednotlivými elementy (`startElement()`, `endElement()` a `characters()`).

Z důvodu snadného použití a manipulace s XML elementy byl vybrán DOM Parser/Builder. K extrakci elementů podle daných kritérií se zde využívá především jazyka **XPath** a knihovny s ním spojené. Jeho základní součástí je tzv. *path expression*, což je obdoba regulárního výrazu obecně popisujícího hledané uzly.

V případě vytvářené aplikace bylo potřeba nejprve zjistit posloupnost hran v grafu pro každý testovací případ, tudíž extrahovat obsah mezi startovacími a ukončovacími značkami všech elementů se jménem `test_situation`. Takovouto jednoduchou operaci lze provést pouze s využitím možností DOM Parseru. Po získání potřebných uzlů nad nimi už jen stačí zavolat metodu `getTextContent()`, jenž vrací obsah uvnitř ohraničujících značek.

```
document.getElementsByTagName("test_situation");
```

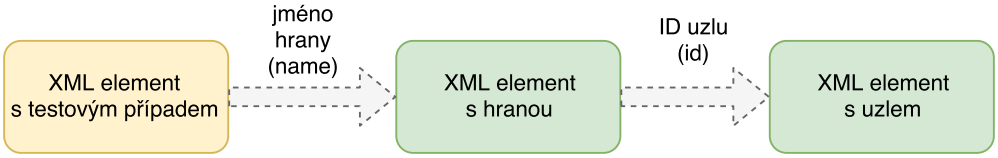
Listing 8.1: Získání všech uzlů se jménem `test_situation`

Takto extrahovaná data obsahují názvy hran, jimiž cesta prochází. Pro získání informací o těchto hranách z druhého XML souboru, představujícího graf testované webové aplikace, již však nestačí pouze metody poskytované DOM Parserem, ale je nutné využít například již zmiňovaný jazyk XPath.

```
XPathExpression edgeExpr =
                        xpath.compile("//edge[@name=" + token + "]");
edges =
(NodeList) edgeExpr.evaluate(document, XPathConstants.NODESET);
Node edge = edges.item(0);
```

Listing 8.2: Použití XPath pro extrakci hrany podle jména

Z objektu hrany lze pak opět pouze s pomocí DOM Parseru zjistit `id` cílového uzlu. Dalším dotazem s využitím knihovny XPath již lze získat samotný element s uzlem. Jak již bylo dříve popsáno, ten a zejména jeho atribut `description` obsahuje informace o dané akci. Na základě nich je vytvořen objekt `DescriptionParser`. Tento postup je přehledně vyjádřen na obrázku 8.1.



Obrázek 8.1: Postup získávání XML elementů

8.2 Třída `DescriptionParser`

Třída `DescriptionParser` představuje akci uživatele. Formálním parametrem jejího konstruktoru je atribut `description` daného uzlu a seznam již vytvořených instancí. V konstruktoru dojde ke zpracování tohoto popisu. Pokud se jedná o akci vyžadující vygenerování nových hodnot na základě externě popsané třídy ekvivalence, je taková akce zavolána. List již vytvořených instancí je přítomný z toho důvodu, že lze požadovat, aby hodnota nebyla nově generována, ale byla stejná jako v některé z předcházejících akcí.

8.3 Generátory

Aplikace ve výchozím stavu může využívat celkem čtyř generátorů. Těm je vždy nutné ještě přiřadit parametry, které určují přesný formát vytvářených dat. Ty se získávají z tříd ekvivalence.

- `DateGenerator`
  Generuje datum na základě parametrů určujících minimální a maximální hodnotu zadanou ve formátu `d.M.yyyy` a formát vygenerovaného data. Formát `d.M.yyyy` značí, že den i měsíc je možné zadat pouze jednou číslicí, aniž by bylo nutné před ni psát ještě nulu. Příkladem takovéhoto formátu je například `4.7.1979`. Je však možné tyto jednočíselné dny a měsíce zadávat i s nulou na začátku, tedy `04.07.1979`.

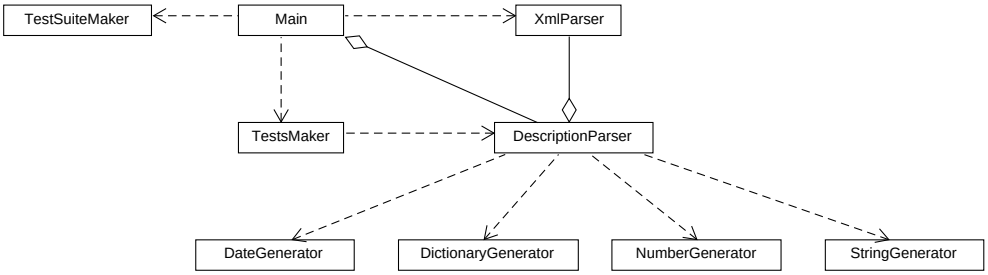
- `DictionaryGenerator`
  Generuje náhodnou položku z uživatelem připraveného slovníku. Cesta k němu je také jediným parametrem. V tomto slovníku musí být každá položka na samostatném řádku.

- `NumberGenerator`
  Generuje číselné hodnoty na základě zadané spodní a horní hranice a informace, zda se má jednat pouze o celočíselné hodnoty.

- `StringGenerator`
  Generuje textové řetězce. Parametry udávají jejich minimální a maximální délku a informaci, zda mohou obsahovat i čísla, nebo pouze písmena.

Vlastní generátory lze přidat jednoduše do adresáře `src/generators`. Dále je potřeba přidat volání tohoto generátoru do třídy `DescriptionParser` a dosadit správné parametry.

Vztah generátorů k ostatním třídám lze vidět na obrázku 8.2, kde je vykreslen UML diagram tříd hlavního modulu aplikace.



Obrázek 8.2: UML diagram tříd hlavní aplikace

8.4 Třídy ekvivalence

Omezení generovaných testovacích hodnot jsou dány pomocí tříd ekvivalence. Tento postup využívá metody rozdělení všech hodnot do skupin pro testování ekvivalentních případů. Z těchto skupin tester vybere tu, která je potřeba pro dané *workflow*. Pro ní sám uživatel (tester) vytváří soubor, jež ji popisuje. Konvencí pro název tohoto souboru je `TE_<název_třídy>.txt` a musí být obsažen v adresáři `TEs`. Je potřeba popsat, jaký generátor se má při jejím vytváření použít a definovat jeho vstupní parametry. Pro název generátoru je vyčleněn parametr `type`. Jelikož jsou názvy tříd generátorů ve formátu `<typ>Generator`, je vhodné zde udávat pouze první část tohoto názvu, tedy typ požadovaného generátoru, a to malými písmeny. Je ovšem i možnost zde uvést plný tvar názvu generátoru. Atributy příslušející jeho parametrům mají vždy daný výstižný název, jenž je uvedený v aplikaci, kde se podle něj tyto parametry dosazují do příslušného generátoru. Příklad třídy ekvivalence využívající generátoru `NumberGenerator` s udanými minimálními a maximálními hodnoty je uveden v ukázce kódu 8.3.

```
type=number
min=15000
max=100000
integer=true
```

Listing 8.3: Příklad třídy ekvivalence využívající `NumberGenerator`

8.5 Kritérium úspěchu

Při testování webové aplikace je nutné nastavit, podle jakých pravidel bude testována a na základě čeho budou testy probíhat. Tím se určí kritéria, na základě kterých bude test označen, že prošel. Tato kritéria lze odvodit z dat, která tester poskytl při vytváření modelu testované aplikace pomocí Oxygenu. Těmito údaji jsou pro každý uzel grafu jeho ID, typ a URL. Z tohoto důvodu bude správnost aplikace ověřována pomocí ověření současné URL adresy, přítomnosti elementů na základě jejich ID a možnost jejich použití v závislosti na typu.

V případě URL adresy je však důležité vzít v potaz, že ne vždy je možné, aby tester ještě před započetím testování určil její přesnou podobu. Jedná se především o ty případy, kdy jsou do URL přidávány různé proměnné. Z tohoto důvodu bylo rozhodnuto, že URL adresa bude testována pouze na to, zda obsahuje daný řetězec.

8.6 Modul `selenium-tests`

Modul `selenium-tests` slouží jako výstupní adresář pro generování testů pro Selenium. Vždy obsahuje třídy `Configurations`, `TestBase`, `TestSuite`, `TestRun` a konfigurační soubor `config.txt`.

Všechny vygenerované testy dědí od společného rodiče `TestBase`, jež je také součástí tohoto modulu. `TestBase` obsahuje potřebné metody s anotacemi `@BeforeClass` a `@AfterClass`, které mají na starosti především práci s `WebDriverem`, a zbylé metody obsahující společný kód pro všechny testy.

Tyto vygenerované testy jsou parametrizované, tudíž každý z nich obsahuje sady testovacích dat pro několikanásobné spuštění. Počet těchto sad zadává uživatel jako parametr při spouštění této aplikace. Příklad takovéhoto testu se třemi testovacími sadami je v ukázce kódu 8.4.

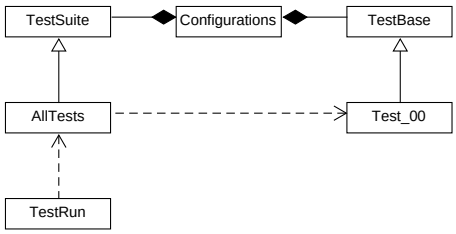
Aby bylo spouštění testů co nejjednodušší, je spolu s nimi vygenerována i test suite, jenž se vždy jmenuje `AllTests`. Aby bylo množství takto gene-

rovaného kódu co nejmenší, dědí tato třída od `TestSuite` obsahující metody `before()` a `after()`.

Pro co nejuniverzálnější použití vytvářené aplikace je potřeba, aby si sám uživatel mohl volit, jaký webový prohlížeč pro testování použije (Google Chrome, Mozilla Firefox, Opera), kde bude mít uložený *driver* nebo jaká je vstupní URL adresa, na níž se testovaná webová aplikace nachází.

K tomu slouží konfigurační soubor `config.txt`. Pomocí něj se dá nastavit výchozí URL adresa testované aplikace, typ prohlížeče a cesty k *driverům* pro jednotlivé prohlížeče.

UML diagram se znázorněnými vazbami mezi třídami je na obrázku 8.3. Vzhledem k tomu, že počet vygenerovaných tříd se seleniovými skripty není omezen a zároveň mají všechny stejné vazby s ostatními třídami, jsou na tomto diagramu zastoupeny pouze první z nich – `Test_00`.



Obrázek 8.3: UML diagram tříd modulu `selenium-tests`.

```
import org.junit.Test;
import java.util.Collection;
import java.util.Arrays;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameter;
import org.junit.runners.Parameterized.Parameters;

@RunWith(Parameterized.class)
public class Test_01 extends TestBase {

   @Parameters
   public static Collection<Object[]> getData() {
     return Arrays.asList(new Object[][] {
          { "xIhtDP", "SvgPhIXmJN", "3IhGgODOg", },
          { "yYRKNlbpsJG", "pQrqbjgKh", "ojSwrD", },
          { "kYrSCG", "RoYGCR", "7BOUKG", },
     });
```

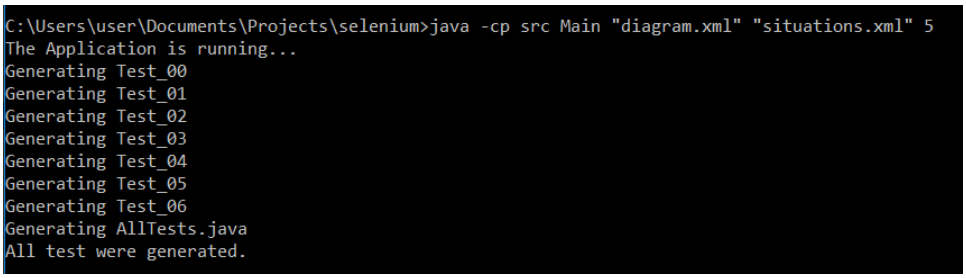
```
  }

  @Parameter(value = 0)
  public String login_nick;
  @Parameter(value = 1)
  public String login_heslo;
  @Parameter(value = 2)
  public String login_heslo_znovu;

  @Test
  public void test_01() {
    checkActualURL("index.php");
    writeToInput("login_nick", login_nick);
    checkActualURL("index.php");
    writeToInput("login_heslo", login_heslo);
    checkActualURL("index.php");
    writeToInput("login_heslo_znovu", login_heslo_znovu);
    checkActualURL("index.php");
    clickAndWait("login_submit", "result_neuspesne");
    checkActualURL("web=neuspesne");
  }
}
```

Listing 8.4: Ukázka vygenerovaného testu

8.7 Použití aplikace

Aplikace jako taková se spouští s kořenového adresáře aplikace následujícím příkazem:
`java -cp src Main "<diagram>.xml" "<test_situations>.xml" <počet_datasetů>`

Parametr `<diagram>` zde udává název XML souboru s diagramem projektu a `<test_situations>` název XML souboru s testovými situacemi. Oba tyto soubory jsou generované aplikací Oxygen. Pomocí `<počet_datasetů>` se volí počet datasetů pro parametrizování výsledných seleniových skriptů. Příklad spuštění aplikace z příkazové řádky lze vidět na obrázku 8.4.

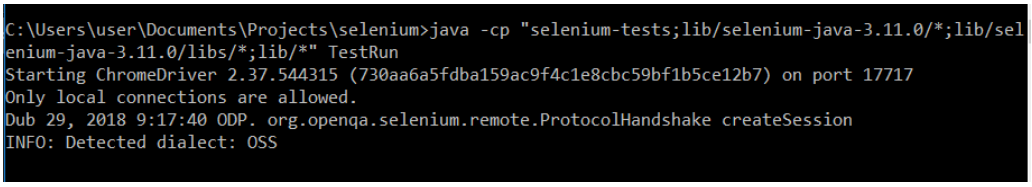


Obrázek 8.4: Spuštění výsledné aplikace z příkazové řádky

Test suite všech vytvořených testů lze spustit pomocí třídy `TestRun` a to rovněž z kořenového adresáře projektu, ve kterém je umístěna, následujícím příkazem:

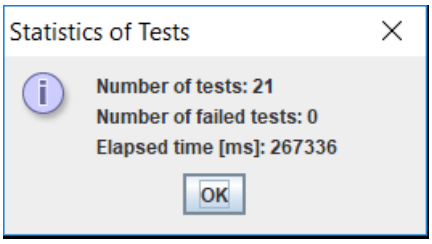
```
java -cp "selenium-tests;lib/selenium-java-3.11.0/*;
lib/selenium-java-3.11.0/libs/*;lib/*" TestRun
```

Příklad spuštění všech testů z příkazové řádky pomocí třídy `TestRun` s nakonfigurovaným webovým prohlížečem Google Chrome lze vidět na obrázku 8.5.



Obrázek 8.5: Spuštění všech testů pomocí třídy `TestRun` z příkazové řádky

Po vykonání všech testů se zobrazí `JPanel` se stručnou statistikou, jehož příklad je na obrázku 8.6.



Obrázek 8.6: `JPanel` se statistikou spuštěných testů

9 Závěr

Cílem této práce bylo ověřit možnost vzájemného použití nástrojů Oxygen a Java Object Populator a vytvořit aplikaci, jenž bude na základě dat vyexportovaných aplikací Oxygen generovat testovací datasety buď pro manuální, nebo automatizované testování.

V rámci této bakalářské práce byly analyzovány obě tyto aplikace. V případě Oxygenu byly popsány i výstupní soubory, jenž jsou dále zapotřebí pro generování testovacích datasetů, a byly navrhnuty úpravy jednoho z nich. Knihovna Java Object Populator byla taktéž popsána a byla k ní vytvořena aplikace demonstrující její použití. Po důsledném zvážení však bylo rozhodnuto, že tato knihovna není vhodná pro generování testovacích dat pro webové aplikace a proto není ve výsledné aplikaci použita.

Pro experimenty byla nad rámec zadání vytvořena speciální webová aplikace v jazyce PHP.

Aplikace připravená v rámci této práce na základě XML souborů z Oxygenu generuje funkční testovací skripty pro Selenium a ke generování testovacích dat využívá generátory vlastní. Ty existující lze snadno podle potřeby doplnit dalšími.

V práci se podařilo ukázat, že při důkladném popisu testované aplikace v nástroji Oxygen je možné automaticky vygenerovat sadu procesních testů. Ty dokáží ověřit správnost (průchodnost) všech existujících cest v testovaném programu a to s použitím vždy několika variant vstupních dat.

Seznam zkratek

API	Application Programming Interface – rozhraní pro programování aplikací, obsahuje definice podprogramů, protokoly a další nástroje pro vývoj aplikací
CSS	Cascading Style Sheets – jazyk popisující styl zobrazení elementů daných značkovacím jazykem
CSV	Comma-Separated Values – způsob zápisu do souboru využívající k oddělování hodnot znak čárka
DFS	Depth First Search – grafový algoritmus pro procházení grafů metodou backtrackingu
DOM	Document Object Model – způsob reprezentace např. HTML nebo XML jako stromové struktury
HTML	Hypertext Markup Language – značkovací jazyk využívaný k tvorbě webových stránek obsahující hypertextové odkazy
HTTP	Hypertext Transfer Protocol – aplikační protokol sloužící k přenosu hypertextových dokumentů ve formátu HTML
IDE	Integrated Development Environment – softwarová sada, nabízející vývojáři prostředky pro vytváření softwaru
JOP	Java Object Populator – knihovna pro generování testovacích dat pomocí anotací
JSON	JavaScript Object Notation – javascriptový zápis objektu a také forma pro přenos dat ve formátu klíč-hodnota
PCT	Process Cycle Test – algoritmus procházení využívající algoritmus DFS
PPT	Prioritized Process Test – algoritmus procházení uvažující rozličné úrovně priorit
PTL	Prioritized Test Level – kritérium pokrytí využívající úrovně *vysoká* a *střední*
RC	Remote Control – nástroj Selenia fungující na principu propojení klientské knihovny a seleniového serveru

REST Representational State Transfer – architektura, jenž využívá pro přístup k datům vzdálené volání procedur

SAX Simple API for XML – typ přístupu k XML, načítá pouze potřebnou část dokumentu pomocí *callbacků*

TDL Test Depth Level – kritérium pokrytí využívající úrovně od 1 do N

UML Unified Modeling Language – grafický jazyk sloužící k vyjádření návrhu aplikace

URL Uniform Resource Locator – jednoznačné určení lokace a způsobu získání webového zdroje

XML Extensible Markup Language – metajazyk umožňující vytváření značkovacích jazyků

Seznam obrázků

Seznam tabulek

Seznam ukázek kódu

Literatura

[1] Ash, L. *The Web Testing Companion: The Insider's Guide to Efficient and Effective Tests*. Wiley; 1 edition (May 2, 2003), 2003. ISBN 978-0471430216.

[2] Bures, M. PCTgen: Automated Generation of Test Cases for Application Workflows. In Rocha, A. et al. (Ed.) *New Contributions in Information Systems and Technologies*, s. 789–794, Cham, 2015. Springer International Publishing. ISBN 978-3-319-16486-1.

[3] Bures, M. – Cerny, T. – Klima, M. Prioritized Process Test: More Efficiency in Testing of Business Processes and Workflows. In Kim, K. – Joukov, N. (Ed.) *Information Science and Applications 2017*, s. 585–593, Singapore, 2017. Springer Singapore. ISBN 978-981-10-4154-9.

[4] Dékány, M. *Generování testovacích dat z anotací*. Západočeská univerzita v Plzni, 2016.

[5] Garg, N. *Test Automation using Selenium WebDriver with Java: Step by Step Guide*. Test Automation Using Selenium with Java; 1.0 edition (December 11, 2014), 2014. ISBN 9780992293512.

[6] Hope, P. – Walter, B. *Web Security Testing Cookbook: Systematic Techniques to Find Problems Fast*. O'Reilly Media; 1 edition (October 27, 2008), 2008. ISBN 978-0596514839.

[7] *What Exactly Is a Web Application?* [online]. 2018. [cit. 2018/03/03]. Dostupné z: `https://www.lifewire.com/what-is-a-web-application-3486637`.

[8] Löwinger, L. *Aplikace pro generování testovacích situací pro techniku Process Cycle Test*. České vysoké učení technické v Praze, 2014. Dostupné z: `https://www.fel.cvut.cz/cz/education/prace/00003.pdf`.

[9] *What is Manual Testing?* [online]. 2018. [cit. 2018/04/01]. Dostupné z: `http://www.softwaretestingclass.com/what-is-manual-testing/`.

[10] Roudenský, P. – Havlíčková, A. *Řízení kvality softwaru, Průvodce testováním*. Computer press; 1. vydání (2013), 2013. ISBN 978-80-251-4519-7.

[11] *Introduction to Selenium* [online]. 2018. [cit. 2018/04/02]. Dostupné z: `https://www.guru99.com/introduction-to-selenium.html`.

[12] *Selenium Documentation* [online]. 2018. [cit. 2018/04/02]. Dostupné z: https://www.seleniumhq.org/docs/.

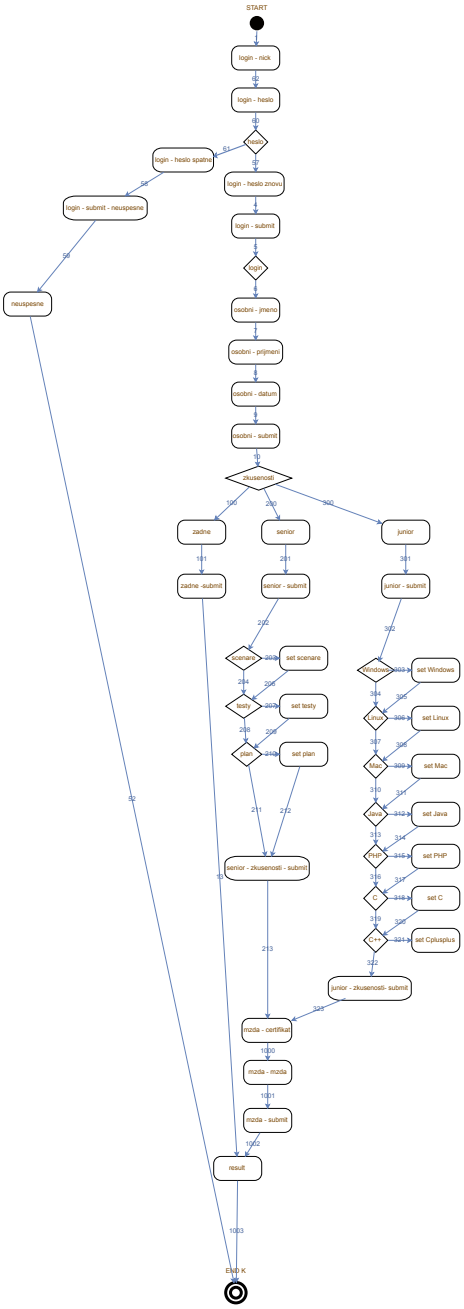
[13] SM, R. *Selenium WebDriver Architecture | Software Testing Material* [online]. 2018. [cit. 2018/04/08]. Dostupné z: https://www.softwaretestingmaterial.com/selenium-webdriver-architecture/.

[14] *Paths* [online]. 2018. [cit. 2018/03/06]. Dostupné z: http://www.tmap.net/wiki/paths.

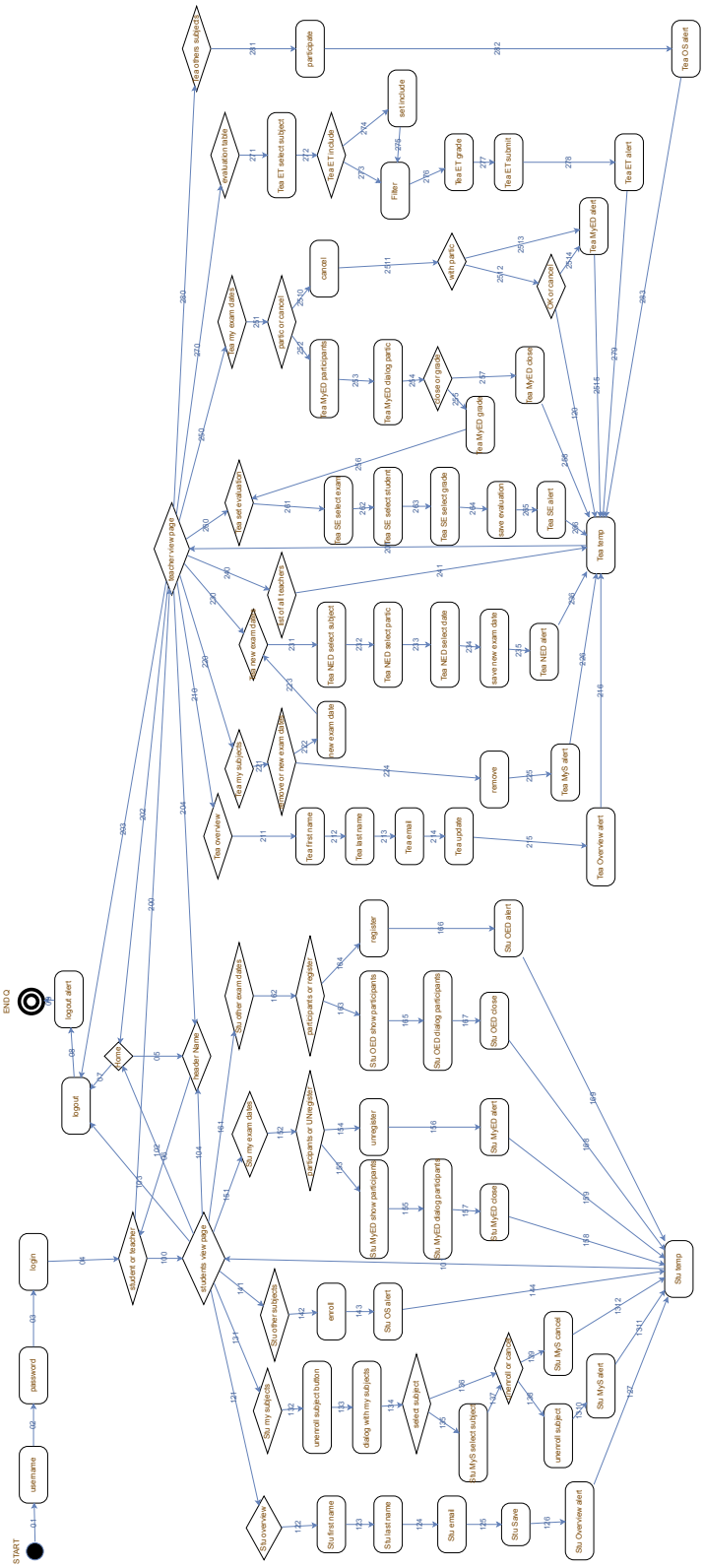
[15] *SUT* [online]. 2018. [cit. 2018/03/06]. Dostupné z: http://xunitpatterns.com/SUT.html.

Diagramy testovacích aplikací

Všechny diagramy z této kapitoly jsou rovněž v elektronické podobě na přiloženém CD.



Obrázek 1: Diagram výsledné testovací aplikace



Obrázek 2: Diagram původní testovací aplikace UIS