Univerzita Pardubice
Fakulta elektrotechniky a informatiky

Automatizované testování informačních systémů využívajících webové služby

Bc. Josef Jetmar

Diplomová práce
2019

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: **2017/2018**

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Josef Jetmar**
Osobní číslo:	**I16221**
Studijní program:	**N2646 Informační technologie**
Studijní obor:	**Informační technologie**
Název tématu:	**Automatizované testování informačních systémů využívajících webové služby**
Zadávající katedra:	**Katedra softwarových technologií**

Zásady pro vypracování:

V teoretické části práce se student zaměří na problematiku testování webových služeb. Důraz bude kladen na přípravu a realizaci softwarových testů, především na tvorbu vstupních datových souborů, inicializačních SQL příkazů, provádění dotazů a vyhodnocování výsledků těchto testů. Dále student provede rešerši v oblasti softwarů pro automatické testování a srovná automatické a manuální testování webových služeb.
Pro praktickou práci bude vybrán jeden testovací nástroj (platforma / software), který je využitelný pro tvorbu testů. Cílem práce je provést takové rozšíření software, aby ho bylo možné využívat k provádění automatických testů a jejich vyhodnocování. Klíčovou akceptační podmínkou je možnost testovat rozhraní REST i SOAP.

Rozsah grafických prací: **10**

Rozsah pracovní zprávy: **60**

Forma zpracování diplomové práce: **tištěná**

Seznam odborné literatury:

ERL, Thomas. SOA: servisně orientovaná architektura : kompletní průvodce. Brno: Computer Press, 2009. Programování (Computer Press). ISBN 978-80-251-1886-3.
BUREŠ, Miroslav, Miroslav RENDA, Michal DOLEŽEL, Peter SVOBODA, Zdeněk GRÖSSL, Martin KOMÁREK, Ondřej MACEK a Radoslav MLYNÁŘ. Efektivní testování softwaru: klíčové otázky pro efektivitu testovacího procesu. Praha: Grada, 2016. Profesionál. ISBN 978-80-247-5594-6.
ISO/IEC/IEEE 29119-1. Software testing. New York: ISO copyright office IEC Central Office Institute of Electrical and Electronics Engineers, 2014.

Vedoucí diplomové práce: **Ing. Josef Brožek**
Katedra informačních technologií

Datum zadání diplomové práce: **30. října 2017**

Termín odevzdání diplomové práce: **18. května 2018**

Ing. Zdeněk Němec, Ph.D.
děkan

L.S.

prof. Ing. Antonín Kavička, Ph.D.
vedoucí katedry

V Pardubicích dne 15. listopadu 2017

Prohlašuji:

Tuto práci jsem vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorský zákon, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Beru na vědomí, že v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů, a směrnicí Univerzity Pardubice č. 9/2012, bude práce zveřejněna v Univerzitní knihovně a prostřednictvím Digitální knihovny Univerzity Pardubice.

V Pardubicích dne 23. 08. 2019

Josef Jetmar

PODĚKOVÁNÍ

Děkuji vedoucímu diplomové práce Ing. Josefu Brožkovi za odborné vedení, dobré rady, podněty a připomínky, které mi poskytoval při zpracovávání diplomové práce.

ANOTACE

Diplomová práce popisuje vybrané přístupy k testování softwaru. V teoretické části je zkoumána obecná problematika testování softwaru. Důraz je kladen zejména na způsob testování webových aplikací a služeb pomocí vývojářských a uživatelských nástrojů. Poznatky o softwarovém testování jsou dále rozšířeny o možnost automatizace a způsoby jakými může být realizována. V praktické části je popsán vývoj vlastní softwarové aplikace sloužící pro otestování koncových bodů rozhraní webových služeb SOAP a aplikačního rozhraní architektury REST.

KLÍČOVÁ SLOVA

automatizované testování, testování softwaru, informační systémy, webové služby, REST, SOAP

TITLE

The Automated Testing of the Information Systems Based on Web Services

ANNOTATION

This thesis describes selected approaches to software testing. The theoretical part examines general problems of software testing. The emphasis is placed on testing web applications and web services using user-oriented or development tools. Knowledge about software testing is further extended by the possibility of automation and the ways in which it can be realized. The practical part describes the development of own software application for testing of SOAP web services and the REST application interface endpoints.

KEYWORDS

automation testing, software testing, information systems, web services, REST, SOAP

OBSAH

Seznam obrázků **10**

Seznam tabulek **11**

Seznam zkratek **12**

Terminologie **13**

Úvod **14**

1 Testování softwaru **15**

1.1 Vztah testování software a pojmů verifikace validace 15

1.2 Testovací dokumentace 17

1.2.1 Testovací případ 17

1.2.2 Testovací scénář/skript 18

1.3 Metody přístupu k testování softwaru 18

1.3.1 Manuální testování 18

1.3.2 Testování s využitím softwaru 19

1.3.3 Automatizované testování 19

2 Softwarové testy **20**

2.1 Jednotkové testy 21

2.2 Integrační testy 21

2.3 Systémové testy 22

2.4 Smoke testy 22

2.5 Další úrovně testování 23

3 Životní cyklus softwarového testu **24**

3.1 Zdroje testovacích dat 25

3.1.1 Zadávání dat uživatelem 25

3.1.2 Zdroje dynamických dat 26

3.1.3 Vnější zdroje dat 27

3.2 Popis implementace testu 27

3.3 Spuštění a provedení testu 28

3.4 Vyhodnocení a prezentace výsledků 29

4 Problematika testování softwaru 31

4.1 Souběžné testování v rámci jednoho prostředí 31

4.2 Změna požadavku na systém 32

4.3 Jednotný repozitář testovacích dat 32

4.4 Nedeterministické výsledky testů 32

5 Význam automatizace testů 34

5.1 Z pohledu vývoje 34

5.1.1 Continuous Integration 35

5.1.2 Code coverage 35

5.1.3 Concolic testing 35

5.2 Z pohledu řízení projektu 37

6 Softwarová řešení pro automatizaci testů 38

6.1 SoapUI 39

6.2 Travis CI 41

6.3 Selenium 42

7 Architektura webových služeb 44

7.1 Simple Object Access Protocol (SOAP) 44

7.2 Representational State Transfer (REST) 44

7.3 GraphQL 46

7.4 Způsob popisu rozhraní 47

8 Analýza a návrh aplikace 48

8.1 Postup analýzy 48

8.2 Slovník pojmů 49

8.3 Funkční požadavky 50

8.4 Nefunkční požadavky 51

8.5 Případy užití a hranice systému 52

8.6 Analytický model tříd 54

8.7 Návrh uživatelského rozhraní 54

9 Aplikace AutoTest 56

9.1 Vývojové prostředky 56

9.1.1 Node.js a Koa 56

9.1.2 Ant Design Pro 57

9.1.3 JSON Schema Faker 57

9.1.4 MongoDB 58

9.1.5 Docker 58

9.1.6 Vývojové prostředí 58

9.2 Struktura řešení praktické části práce 59

9.2.1 Webové rozhraní (client) 59

9.2.2 Serverová část aplikace AutoTest (server) 60

9.2.3 Testované rozhraní (tested-server) 60

9.2.4 Veřejné rozhraní datových zdrojů 60

9.3 Tvorba vstupních dat pro testování systému 61

9.4 Generování testovacích běhu 61

10 Diskuze – aplikace Autotest 63

10.1 Testované rozhraní 63

10.2 Cíle 64

10.3 Webové uživatelské rozhraní 64

10.4 Výsledky 65

10.5 Prostor pro rozšíření aplikace 65

Závěr 66

Použitá literatura 67

Přílohy 70

SEZNAM OBRÁZKŮ

Obrázek 1: Schéma vztahu mezi testování softwaru a procesy validace/verifikace 16
Obrázek 2: Graf závislosti ceny a úsilí pro realizaci testu v čase (vlastní tvorba).....20
Obrázek 3: Blokové schéma popisující fáze životního cyklu testu.....24
Obrázek 4: Blokové schéma uspořádání softwarového testu pro nástroje JUnit a Jest. 28
Obrázek 5: Forma vyhodnocení výsledku testovacích sad v rámci NetBeans IDE.....30
Obrázek 6: Znázorněný čtyř konečných stavů programu dle hodnoty vstupních dat.....36
Obrázek 7: Grafické uživatelské prostředí nástroje SoapUI.....40
Obrázek 8: Průběh a výsledek testu v rámci webového rozhraní nástroje Travis CI 41
Obrázek 9: Vyhodnocení testů pomocí Selenium RC (vlastní tvorba).....42
Obrázek 10: Grafické uživatelské rozhraní nástroje Selenium IDE 43
Obrázek 11: Obecné případy užití aplikace AutoTest 52
Obrázek 12: Analytický model tříd aplikace AutoTest.....54
Obrázek 13: Wireframe – Přehled entitních schémat 55
Obrázek 14: Wireframe – Tvorba nového entitního schématu 55
Obrázek 15: Realizace webového uživatelského rozhraní – tvorba entitního schématu 64

SEZNAM TABULEK

Tabulka 1: Skupiny funkčních požadavků aplikace AutoTest.....50
Tabulka 2: Popis skupiny požadavků REQ002 – Struktura entitního schématu 50
Tabulka 3: Popis skupiny požadavků REQ003 – Generování datových entit 51
Tabulka 4: Nefunkční požadavky aplikace AutoTest 52
Tabulka 5: Závislost počtu generovaných běhů na složitosti entitního schématu 62

SEZNAM ZKRATEK

CRUD	Create, Read, Update, Delete
DTD	Document Type Definition
GUI	Graphic user interface
HTTP	Hypertext Transfer Protocol
IEEE	Institute of Electrical and Electronics Engineers
IDE	Integrated development environment
ISO	International Organization for Standardization
JSON	JavaScript Object Notation
OSI	Open Systems Interconnection
REST	Representational State Transfer
RMI	Remote Method Invocation
RPC	Remote Procedure Call
SMTP	Simple Mail Transfer Protocol
SOAP	Simple Object Access Protocol
SVN	Apache Subversion
SQL	Structured Query Language
TDD	Test Driven Development
UAT	User Acceptance Testing
URI	Uniform Resource Identifier
UX	User Experience
W3C	World Wide Web Consortium
WSDL	Web Services Description Language
XML	Extensible Markup Language
XSD	XML Schema Definition
XSLT	Extensible Stylesheet Language Transformations
YAML	YAML Ain't Markup Language

TERMINOLOGIE

black box testing	Testování systému pouze pomocí jeho vnějšího rozhraní, bez znalosti vnitřního uspořádání nebo zdrojového kódu.
deployment	Proces zahrnující všechny úkony spojené se spuštěním systému na konkrétním prostředí.
flaky tests	Softwarové testy jejichž průběh nebo vyhodnocení se neřídí pravidly deterministických algoritmů.
mock	Nástroj umožňující modifikaci implementace systémové komponenty, za účelem dosažení očekávaného průběhu softwarového testu.
proxy	Prostředník v rámci dvoubodové komunikace. Přijímá zprávy od klienta, které transformuje a zasílá na server. Následnou odpověď ze serveru opět transformuje a vrací zpět na klienta. Klient ani server o existenci proxy nemusí vůbec vědět.
softwarový test	Popis konkrétního testovacího scénáře pomocí programu nebo speciálního softwarového.
stack trace	Posloupnost naposledy volaných funkcí v rámci spuštěného programu. V případě, neočekávané chyby v podobě výjimky pomáhá odhalit místo kde výjimka vznikla.
teardown	Fáze testu, jenž má za úkol vrátit použité datové a systémové prostředky zpět do stavu před testem a tím připravit prostředí pro spuštění následujícího testu.
testovací prostředí	Běžící instance testovacího nástroje.
testované prostředí	Testované prostředí se skládá z hardwaru a testovaného systému.
testování softwaru	Proces ověřování funkčních a nefunkčních požadavků kladených na vyvíjený software.
white box testing	Testování se znalostí vnitřního uspořádání systému a jeho zdrojového kódu.
test suite	Testovací sada – jedná se o logicky pojmenované seskupení testovacích případů.
artefakt	V rámci soft. inženýrství označuje entity, které jsou využívány pro modelování softwaru. Např. zdrojový kód, dokumenty, modely atd.

ÚVOD

Zvyšující se výkon a dostupnost výpočetních zařízení má za následek stoupající poptávku po stále kvalitnějším a komplexnějším aplikačním softwaru. Na trhu se zpravidla objevuje množství produktů, jenž si vzájemně do jisté míry konkurují. Schopnost včasně reagovat na měnící se potřeby zákazníků může představovat významnou konkurenční výhodu. Avšak množství nových požadavků (a s tím spojený následný zásah do stávající implementace aplikace) s sebou nese riziko v podobě narušení dosavadní funkčnosti nebo stability vyvíjeného produktu.

Z potřeby průběžného vyhodnocování stavu projektu vyvstává požadavek vnést do procesu vývoje softwaru metriku, která by byla schopna vyčíslit aktuální funkčnost, či naopak chybovost aplikace a tento výsledek porovnat vzhledem k předchozím stavům. Takový nástroj může poskytovat podporu pro vývojáře při zanášení změn do dosavadní implementace aplikace.

Situace se komplikuje v okamžiku, kdy jsou do projektu začleňování vývojáři v různých časových fázích projektu. V takovém případě není v silách nově příchozího jednotlivce vytvořit si v krátké době dostatečný přehled nad vnitřní integritou vyvíjeného produktu natolik, aby bylo možné zcela vyloučit nechtěné narušení jeho dosavadní stability.

Ve chvíli, kdy je potřeba ověřit, zda vyvíjený produkt splňuje požadavky, které jsou od něj očekávány, přichází na řadu proces testování. Realizací testů jsou získány výsledky, které mohou být kritické pro další projektové řízení. Management, tímto způsobem dostává informace o tom, jak velká (procentuální) část produktu je dokončena, což usnadňuje lépe určit čas kdy, bude možné přejít do další plánované fáze projektu.

Z důvodu udržení průběžné informovanosti projektového vedení o stavu projektu, je nutné provádět vyhodnocování testů opakovaně. V případě, že je taková činnost svěřena lidem, pak se vzrůstající složitostí produktu nelze vyloučit komplikace zapříčiněné lidským faktorem. To je způsobeno například tím, že ke stávajícím testovacím scénářům jsou doplňovány nové, čímž narůstá režie spojená s jejich sestavením, aktualizací a vyhodnocením.

Kritické části aplikace je výhodné mít podchycené pomocí programových testů. Tento přístup omezuje riziko lidské chyby, neboť jednou zapsaný test pomocí programu je vždy vyhodnocován stejným způsobem. I v tomto případě je však potřeba aby takový test někdo vytvořil, spustil, vyhodnotil a na základě výsledku učinil patřičná rozhodnutí. Dalším logickým krokem je tedy zvýšit samočinnost procesu vývoje produktu, a to prostřednictvím zavedení automatizace v procesu testování.

1 TESTOVÁNÍ SOFTWARU

Testování softwaru lze definovat jako cílenou činnost během níž probíhá kontrola naplnění požadavků, které jsou kladeny na vyvíjený systém. Hlavním smyslem je nalezení chyb v softwaru, tak aby bylo možné jejich výskyt eliminovat ještě předtím, než se vyvíjený produkt dostane ke koncovému uživateli. Testování je součástí procesu vývoje softwaru už od samého počátku. První pokusy o ladění chyb v softwaru lze datovat do pozdních šedesátých let dvacátého století (O'Broin 2013, s. 14). Tehdy se však jednalo o aktivitu, která byla plně v kompetenci vývojářů a probíhala pouze během vývoje softwaru (zpočátku formou debuggování).

V důsledku zvyšující se poptávky po informačních systémech v rozličných oblastech průmyslu jsou však na nově vyvíjené systémy kladeny stále větší požadavky, které vychází z individuálních podnikových pravidel (business requirements). V odvětvích jako je zdravotnictví, bankovnictví nebo letecká doprava může chyba v softwaru zapříčinit škodu ve formě finanční ztráty daleko převyšující pořizovací cenu vytvořeného softwaru. V nejhorších scénářích pak i ztrátu na lidských životech.

S nárustem složitosti programu zákonitě roste i pravděpodobnost, že bude obsahovat chyby. Z tohoto důvodu není vhodné spoléhat se pouze na vývojáře, ale zodpovědnost za otestování produktu distribuovat i mimo skupinu lidí zodpovědných za programovou implementaci výsledného řešení.

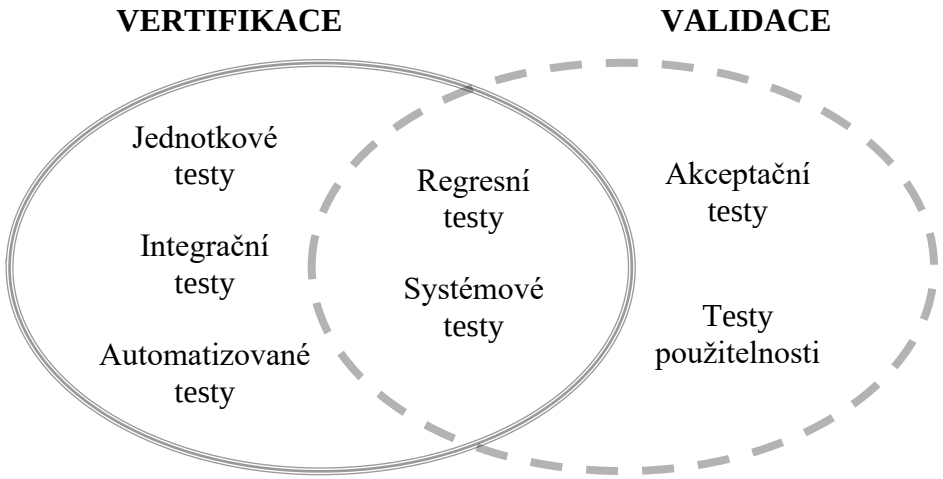
V současnosti jsou pro tyto účely vyčleňovány samostatné analytické a testovací týmy, využívající nástroje a techniky vyvinuté přímo pro popis a testování softwaru. Není tak potřeba, aby člen takového týmu disponovat znalostmi z oboru programování nebo aby nějakým způsobem musel zasahoval do programového kódu. Testeři zpracovávají testovací dokumentaci, tvoří testy, provádějí testovaní a dále reportují výsledky managementu nebo ohlašují chyby zpět do vývoje. Odpovědnost za testování systému na úrovní zdrojového kódu nadále zůstává v režii vývojového týmu.

1.1 Vztah testování software a pojmů verifikace validace

Procesu vývoje a testování softwaru předchází fáze sběru a analýzy požadavků na vyvíjený produkt. Poté co jsou požadavky zanalyzovány, je vytvořena tzv. specifikace požadavků, jenž slouží jako součást dohody mezi zadavatel a vývojovým týmem. Funkční specifikace je dokument, který popisuje vnější funkční rozhraní vyvíjeného produktu. Vymezuje hranice systému a mimo jiné formou datového modelu popisuje entity využívané v rámci systému.

Nezmiňuje však použití konkrétních nástrojů a technologií. Konkrétní technické řešení včetně použitých technologií je pak předmětem technické specifikace, která popisuje konkrétní postupy a realizace pomocí konkrétních technologií.

Následná implementace produktu je realizována na základě technické specifikace. To, do jaké míry odpovídá implementace návrhu systému je třeba ověřit pomocí testů nižší úrovně (jednotkové a integrační testy). Tento proces se označuje jako **verifikace**. V pozdější fázi, kdy se systém dostane k zadavateli je třeba ověřit, zda systém splňuje jeho očekávání, tedy zda skutečně slouží účelu, pro který byl zamýšlen (proces **validace**) (Lewis 1992).



Obrázek 1: Schéma vztahu mezi testování softwaru a procesy validace/verifikace[1]

Standard IEEE 610.12 (1990, volně přeloženo) vysvětluje pojmy verifikace a validace následovně: *„Verifikace je proces vyhodnocování softwaru za účelem rozhodnutí, zda produkty v rámci jedné vývojové fáze splňují podmínky, které na ně byly na začátku dané fáze kladeny.“* Jedná se o postup jenž, má za úkol ověřit, zda je systém vyvíjen odpovídajícím způsobem. U pojmu validace pak lze nalézt formulaci: *„Validace je způsob vyhodnocování softwaru nebo komponenty v průběhu či na konci vývojového procesu, jehož náplní je rozhodnout, zda výsledný software splňuje specifikované požadavky.“*

[1] Zjednodušené schéma převzato z https://www.easterbrook.ca/steve/2010/11/the-difference-between-verification-and-validation/.

1.2 Testovací dokumentace

Smyslem testovací dokumentace je poskytnout jednotné rámcové řešení pro podporu úkonů spojených s testováním během jednotlivých etap vývoje produktu. Jedná se o sadu dokumentů, které mají za úkol dostatečným a jednoznačným způsobem popsat v jaké fázi projektu budou probíhat konkrétní testy. Obsah a struktura testovací dokumentace je obvykle přizpůsobena konkrétnímu produktu nebo společnosti, která software vyvíjí. Existují však mezinárodní standardy, jako například norma IEEE 829, respektive novější IEEE 29119-3, které specifikují formu a základní rozdělení jednotlivých testovacích dokumentů, dále definují také terminologii, využívanou při testování softwaru.

1.2.1 Testovací případ

Testovací případ (test case) obsahuje množinu akcí/kroků, jejichž provedením nad konkrétní množinou vstupních dat by mělo dojít k dosažení předem očekávaného výsledku. Jedná se zpravidla o nízko úrovňový pohled soustředěný na ověření funkčnosti konkrétní systémové komponenty. Vstup testovacího případu je definován vstupními daty a výčtem předpokladů (preconditions) určujících, pro jaké vstupní hodnoty má daný test smysl.

Vstupní testovací data se obvykle znázorňují v jednoduše čitelné formě. Může jít například o výčet proměnných s hodnotami. Další možností je, že jsou data obsaženy v přiložených textových souborech, a to z důvodu zjednodušení potenciálního zásahu do testu (např. z důvodu změny požadavků na implementaci). Výstup z testovacího případu je popsán ve formě očekávaného a skutečného výsledku. V případě, kdy jsou si všechny tyto hodnoty rovny, je test označen za úspěšný.

Kroky v rámci testovacího scénáře obvykle ověřují funkční požadavek na základě pozitivního scénáře – tzn. předpokládají, že vstupní data jsou vždy uvedena v platné podobě vzhledem k funkčnosti testované komponenty. V případě, že je zdrojem vstupních dat uživatelský vstup nebo externí systém, nemusí být podmínka validního vstupu vždy splněna. Je tedy vhodné pro danou systémovou komponentu vytvořit další testovací případ, který bude vést k alternativnímu závěru (v rámci vyhodnocení bude ověřeno, že pro daný vstup skončil test konkrétní chybou).

1.2.2 Testovací scénář/skript

Jako testovací scénář označujeme posloupnost testovacích případů uspořádaných do jednoho logického celku. Z důvodu lepší přehlednosti je vhodné testovací případy shlukovat do větších kategorizovaných celků – testovacích scénářů. Nevýhodou pro menší projekty je riziko vyšší režie při správě testovacích dokumentů.

V ideálním případě by testovací scénáře měly pokrývat veškeré události, které mohou v testovaném systému reálně nastat, a to včetně nepříznivých situací, jako je nevalidní vstup od uživatele, nebo nečekaný výpadek kritické části systému. Prioritní je však, aby byly pokryty požadavky uvedené v rámci specifikace.

1.3 Metody přístupu k testování softwaru

Existují tři hlavní způsoby, kterými lze přistupovat k testovaní softwaru:

1. manuální testování,
2. testování s využitím softwaru,
3. automatizované testování.

Jednotlivé metody se liší především způsobem, jakým dochází k provádění a vyhodnocování testů. Dále pak časovou náročností a technickou způsobilostí potřebnou pro jejich vytvoření. Důraz na testovací metody je podmíněn záměrem, rozsahem a časovým plánem projektu. Důležitou roli může při výběru sehrát také množství kvalifikovaných pracovníků, kteří jsou k dispozici pro potřeby testovaní a vývoje .

1.3.1 Manuální testování

Nejjednodušší metodou využitelnou pro prvotní zkoumání funkčnosti systému je pomocí manuální testování. V tomto případě není třeba využívat nějakého specializovaného softwaru. Pokud je to možné, lze využít přímo uživatelské rozhraní vyvíjeného softwaru. Tester postupuje přesně podle bodů popsaných v testovacím případu a následně vyhodnotí výsledek (typicky vyplněním speciálního formuláře). Jedná se o metodu, pro kterou není třeba znalost programování a během níž lze pružně reagovat na chování systému, což usnadňuje popis potenciální chyby.

Mezi nevýhody manuálního testování patří značné vytěžování lidských zdrojů, a to především v případě opakovaného provádění testů. V důsledku toho je vhodné tuto metodu

využívat převážně pro prvotní průzkum dané funkčnosti systému. V případě, že je vše v pořádku, je výhodné podchytit scénář pomocí softwarového testu.

1.3.2 Testování s využitím softwaru

Testy s využitím softwaru řeší problematiku potřeby aktivní účasti testera při opakovaném prověřování a vyhodnocování testovacího případu. Testy jsou v tomto případě sestaveny buď na základě programu, anebo pomocí specializovaných nástrojů. Hlavní výhoda spočívá v tom, že takto jednou připravený test je možné spouštět opakovaně, a to mnohonásobně rychleji v porovnání s manuálním přístupem k testování (odpadá čekání na manuální zadání vstupu od testera, které tvoří většinu času stráveného nad testem).

Nevýhodou takového řešení může být nízká variabilita scénářů během testování. Test je obvykle vytvořen pro jednu konkrétní sadu vstupních dat a není schopen vhodně reagovat na neočekávané stavy systému, které mohou mít za následek vyhodnocení s falešně negativním nebo falešně pozitivním výsledkem.

1.3.3 Automatizované testování

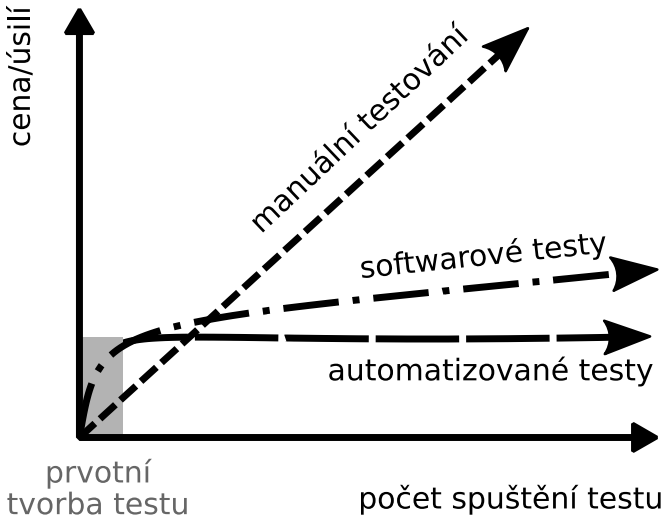
Automatizace v rámci testování spočívá ve snaze minimalizovat vynaložené lidské úsilí při vytváření, spouštění a vyhodnocování softwarových testů. S tím, jak softwarový produkt postupně narůstá je potřeba věnovat stále více pozornosti nově implementovaným částem systému. Automatizované testy, tak mohou sloužit jako aktivita, nahrazující potřebu alokovat lidské zdroje na testování již otestované funkcionality systému.

Nejčastěji je automatizace spojena se spouštěním již vytvořených softwarových testů na základě předem definovaných podmíněných událostí. Takovou událostí může být například naplánované spuštění v daný časový okamžik. Typicky je pro tyto účely zvolena část dne mimo pracovní dobu, tak aby byla minimalizována pravděpodobnost, že na testovaném systému budou probíhat další úkony, které by mohly ovlivnit průběh takto spuštěných testů.

V poslední době je pojem automatizované testování často zmiňován jako součást novodobých přístupů k vývoji softwaru. Toto platí především pro ty moderní přístupy, které zahrnují tzv. postupnou integraci změn (Continuous Integration) respektive na to navazující postupné testování (Continuous Testing). Automatizované spouštění testů je realizováno k ověření skutečnosti, zda se do produkční verze kódu nedostala změna, jenž by mohla negativně ovlivnit dosavadní funkčnost systému. Obvykle je tento způsob automatizovaného testování využit v kombinaci se systémem pro verzování zdrojového kódu (například GIT nebo SVN).

2 SOFTWAROVÉ TESTY

Softwarové testy spadají do kategorie tzv. dynamického testování – realizace testu je podmíněna existencí alespoň základní implementace testovaného systému. Provádění testu je svěřeno testovacímu nástroji, který jej vykonává na základě programu. Takový test je vytvořen buď přímo pomocí programovacího jazyka, nebo pomocí speciálního nástroje. Hlavní výhoda softwarových testů spočívá v úspoře lidských zdrojů při opakovaném spouštění a vyhodnocování testů v průběhu životního cyklu projektu (viz Obrázek 2).



Obrázek 2: Graf závislosti ceny a úsilí pro realizaci testu v čase (vlastní tvorba)

Testování lze na základě obecných kritérií rozdělit následovně:

- Dle úrovně, ve které se nacházejí testované části systému:
  - jednotkové/unit testy,
  - integrační testy,
  - funkční testy.
- Dle znalosti vnitřní struktury a funkčnosti systému:
  - **white box** – test má přístup k vnitřním částem systému,
  - **black box** – test přistupuje k systému pouze pomocí veřejného rozhraní.
- Dle testovaného nefunkčního požadavku:
  - bezpečnostní testy (security tests),
  - testy použitelnosti (usability tests),
  - výkonnostní testy (performance tests).

V závislosti na rozsahu a typu projektu se lze setkat s dalšími typy a označeními. Některá literatura ve spojitosti s kategorizací testu popisují přes 100 druhů testů (Rana, 2019). Následující kapitoly se zaměřují pouze na ty typy testů, které jsou pro automatizované testování významné.

2.1 Jednotkové testy

Nejnižší úroveň softwarových testů představují jednotkové testy (unit test). Jedná se o testy tvořené vývojáři během fáze vývoje. V závislosti na typu řízení projektu mohou být testy realizované ještě před funkční implementací (např. projekt řízený metodou TDD). Jednotkové testy mají za úkol otestovat jednotku programové kódu (typicky funkce nebo metoda objektu). Zápis takových testů by měl být dostatečně abstrahovaný od částí, které nesouvisí s testovanou funkcionalitou. V rámci spuštěného testu by nemělo docházet k interakci s vnějšími aktéry(zejména ke komunikaci skrze počítačovou síť, t využívání vzdáleného databázového systému, autorizačních nebo jiných aplikačních serverů).

Výhody jednotkových testů se projeví především při častých změnách zdrojového kódu. Správně napsané jednotkové testy dokáží už během fáze vývoje zachytit defekty, které byly způsobeny například zásahem do implementace. Lze tak velmi efektivně předcházet situacím, kdy oprava jedné chyby má negativní dopad na jiné, dříve funkční, části systému. V případě, že dojde ke změnám na úrovni specifikace je potřeba počítat nejen se zásahem do implementace dané funkčnosti, ale také s přizpůsobením souvisejících testů.

2.2 Integrační testy

Při zahájení testování pomocí integračních testů se předpokládá, že jednotlivé moduly systému již byly ověřeny prostřednictvím jednotkových testů. V této úrovni testování je kladen důraz na ověření správného přenosu dat mezi jednotlivými částmi systému a jejich funkčnosti v rámci jednoho celku. Důvodů pro tvorbu integračních testů může být několik. Jednak se na vývoji zpravidla podílí větší množství vývojářů, kteří jednotlivé části vyvíjejí odděleně, tím vyvstává potřeba ujistit se, že takto vyvinuté části vzájemně kooperují. Další důvody pro tvorbu integračních testů může představovat technologická různorodost systému nebo existence vazby na řešení třetích stran. V takových případech je výhodné při návrhu systému věnovat dostatečnou pozornost softwarovým rozhraním, které bude systém využívat. Možná vázanost na vnější prostředí systému může být spjata s potřebou vlastnit licenci, certifikáty nebo

přístupové údaje, které by v rámci prvotní fáze vývoje mohly přinést nemalé finanční výdaje. Z tohoto důvodu jsou obvykle integrační testy vytvářeny a spouštěny až v pozdějších fázi vývoje.

2.3 Systémové testy

Obsahuje-li vyvíjený produkt potřebné komponenty, které byly úspěšně otestovány na úrovni integračních testů, je možné soustředit se na testování systému jako celku. Systémové testy představují poslední fázi testování předtím, než je produkt předán k otestování zadavateli (akceptační testy). V této fázi je možné skutečně otestovat splnění funkčních a nefunkčních požadavků, které byly kladeny na systém v rámci jeho analýzy.

Protože v předchozích fázích bylo testování zaměřeno buď na jednu komponentu, či její nejbližší okolí v podobě menšího počtu integrovaných komponent, nebylo možné otestovat aspekty spojené s fungováním systému jako celku. V této fázi se proto lze lépe soustředit na veličiny jako například časovou odezvu, vytížení databáze nebo hardwarové nároky systému. Dále je možné ověřit, zda části systému, které byly v analýze označeny, za systémově náročné skutečně odpovídají původním předpokladům z pohledu odezvy systému.

2.4 Smoke testy

Smoke testy ověřují základní stabilitu a obecnou funkcionalitu systému. Jedná se o skupinu testovacích případů, které mají za úkol okamžitě otestovat kritické funkce jako je bezproblémové spuštění systému, připojení k databázi, přihlášení uživatele a jeho následná navigace napříč systémem. Na základě jejich výsledku se nadále rozhoduje o tom, zda má smysl pokračovat v další testovací fázi.

Spuštění je obvykle podmíněno nějakou předchozí událostí jako například vydání nové vývojové verze, nasazení systému na nové prostředí nebo posunutí projektu do další fáze vývoje. Jedná se tedy o proces, který je výhodné automatizovat. Vhodným pravidlem pro určení, které části systému je výhodné mít pokryto kouřovými testy je za využití obecného popisu regresních testů – tedy testů které vznikli na základě již dříve nahlášeného defektu při testování funkčnosti systému (Craig 2008, s. 129).

2.5 Další úrovně testování

Běžně lze narazit i na další kategorie testů, které už ale z pohledu automatizace nemají takový význam. Typickým příkladem jsou akceptační testy, které zkoumají funkční požadavky na systém z pohledu zadavatele. Jedná se o poslední fázi testování před uvedením do produkce. Nejčastěji je realizováno lidmi, kteří mají později skutečně systém používat.

Další úroveň testování představují UX testy, které vyhodnocují to, do jaké míry je práce se systémem intuitivní z uživatelského hlediska. Pro testování jsou obvykle vybírání jednotlivci, mají znalosti týkající se podnikové problematiky, kterou se systém zabývá, avšak neprošli specializovaným školením pro ovládání systému.

Kategorii testů, které není možné zcela automatizovat představují testy bezpečnosti. Lze předpokládat, že vývoj softwaru probíhá s úmyslem, záměrně nevytvářet v systému slabá místa z pohledu bezpečnosti. Implementace bezpečnostních technik je primárně v rukách vývojářů a je tedy potřeba, aby byl vývojový tým s touto problematikou dostatečně obeznámen a byl v této oblasti i řádně proškolen.

Velké množství vyvíjeného softwaru je v dnešní době závislé na programových knihovnách třetích stran. Zdrojový kód těchto knihoven nemusí být vždy dostupný a obsahuje-li nějakou bezpečností díru, obvykle ani není v možnostech vývojáře takové chování předem předpokládat. Proto je možné se při vývoji software setkat se zastánci vývoje bez použití nejnovějších verzí knihoven. Využití starších verzí knihoven zvyšuje pravděpodobnost vyhnutí se tzv. zero-day útoku, tedy softwarovým útokům za využití bezpečnostních děr, které ještě nejsou ve všeobecném povědomí veřejnosti nebo na ně ještě nebyla vydaná bezpečností záplata.

3 ŽIVOTNÍ CYKLUS SOFTWAROVÉHO TESTU

Životní cyklus softwarového testu popisuje jednotlivé fáze od návrhu testu až po jeho vyhodnocení. Existují tři pohledy, jakými lze nahlížet na realizaci softwarového testu:

- obecný popis životního cyklu testu,
- Arrange Act Assert pattern (vysvětleno dále v kapitole),
- popis životního cyklu z pohledu implementace pomocí testovacího frameworku.

První zmíněný popis životního cyklu testování softwaru, popisuje posloupnost fází obecného procesu testování. Netýká se tedy pouze softwarových testů, ale je aplikovatelný i na testy manuální. V případě popisu softwarových testů se však odlišuje způsobem realizace jednotlivých fází. Z pohledu životního cyklu softwarového testu je důležitá především fáze implementace. V rámci implementace je navržený testovací případ popsán pomocí vhodného softwarového nástroje. Tím je zápis testu z podoby textového dokumentu převeden do spustitelné formy. Po této fázi končí technicky náročnější část procesu testování za využití softwarového testu. Následuje fáze spuštění a fáze vyhodnocení výsledků, jenž jsou popsány v následujících podkapitolách. Poslední dvě zmíněné fáze je možné realizovat bez hlubší znalostí softwarového vývoje (Garousi a Elberzhager 2017).

Obrázek 3: Blokové schéma popisující fáze životního cyklu testu[2].

Pro tvorbu testu je nutné nejprve zvolit části systému, které budou zkoumány, a určit za jakých podmínek bude jejich testování probíhat. Na základě této úvahy je proveden výběr odpovídající úrovně testu a je vhodně zvolen zdroj vstupních dat.

Pro udržení přehlednosti se jednotlivé testy spojují do kolekcí, které jsou součástí, tzv. testovací sady (test suite). Jedná se o rozčlenění testů do logických celků tvořených na základě vazby se stejnou částí či funkcionalitou testovaného softwaru. Testy obsažené uvnitř stejné testovací sady mohou sdílet společný datový prostor a provádět interakci se stejnou instancí systému.

[2] Zjednodušené schéma převzato z článku (Garousi a Elberzhager 2017, s. 92).

V případě zápisu testu pomocí programu je vhodné (především na úrovní jednotkových testů) dodržovat logické členění zdrojového kódu pomocí vzoru Arrange Act Assert. Hlavním myšlenkou takového přístupu k implementaci testů je dodržování předem stanovené formy zápisu do třech logických částí:

- **Arrange** – Přípravná fáze testu, kdy dochází k přípravě vstupních dat pro testovanou komponentu.
- **Act** – Činná fáze testu, v rámci které jsou data z předchozí fáze využita jako vstup do testované části systému. Data jsou následně systémem vyhodnocena a výsledek je formulován v podobě návratové hodnoty nebo změnou vnitřního stavu systému.
- **Assert** – Část testu, v rámci které dochází k vyhodnocování výsledku předchozí fáze. Vyhodnocování probíhá na základě porovnávání výsledků s očekávanými hodnotami.

Zápis testu za pomocí testovacího frameworku je blíže popsán v kapitole 3.2.

3.1 Zdroje testovacích dat

Aby bylo možné funkci systému otestovat je nejprve nutné definovat vhodná vstupní data. Zdroje takových dat mohou být deterministické – hodnoty jsou obsažené v proměnných, jež jsou definovány přímo uvnitř implementace testu nebo pomocí přiložených datových souborů. Tímto způsobem je kontrolováno chování systému v rámci jednoho konkrétního případu. Při využití pevné podoby vstupních dat, zpravidla očekáváme při opakovaném provedení dosažení stejného výsledku, proto nemá příliš velký význam provádět tyto testy opakovaně (pokud nedošlo ke změně implementace testovaného softwaru).

Výjimkou může být situace, kdy je potřeba otestovat vzájemnou nezávislost softwarových testů, například při práci s perzistentními daty. V takovém případě může do průběhu testování vstupovat i pozůstatek z dříve provedeného testu. V důsledku toho se při jiné konstelaci testovacích scénářů může být ten samý test vyhodnocen jiným způsobem, než pokud byl proveden samostatně.

Další možností je využít stochastické zdroje dat, jako například generátory, nebo interaktivní uživatelské vstupy. Vybrané stochastické zdroje dat jsou popsány dále.

3.1.1 Zadávání dat uživatelem

Přímé zadávání dat uživatelem je zprostředkováno pomocí speciálně vytvořeného uživatelského rozhraní, které může funkčně připomínat rozhraní výsledné aplikace, nebo se

může jednat o práci se systémem v tzv. „testovacím módu“. Tento přístup ovšem vyžaduje vyšší režii, kdy je při implementaci systému třeba brát ohledy i na potřeby testovacího rozhraní. V důsledku toho se může funkčnost v testovacím režimu lišit – při interakci s rozhraním, tak nemusí přímo docházet k testování vlastností a funkcí skutečného systému.

Výhodou metody je, že testerovi je připraveno speciální prostředí, do kterého stačí pouze vkládat vstupní data a samotný průběh testu se pak sám řídí dle kroků popsaných v rámci předpřipravených testovacích procedur. Nevýhodou je stálá potřeba operátora (testera), který musí test obsluhovat. Tato metoda také není využitelná pro plně automatizované testování.

3.1.2 Zdroje dynamických dat

Dalším možným zdrojem dat pro testování mohou být stochastické generátory vstupních dat. Generátorem je zde zpravidla myšleno softwarové řešení, které je, na základě vstupní konfigurace, schopné vygenerovat odpovídající sady testovacích dat. Konfiguraci pro takové generátory představuje zpravidla sada pravidel popisujících strukturu a omezení výsledné podoby generovaných dat.

Hodnoty jsou generovány náhodně, respektive pseudonáhodně. Využití generátoru umožňuje v rámci jediného testovacího případu provést větší množství testů s různými hodnotami vstupních dat, a to bez potřeby lidského operátora. Je výhodné, aby byl generátor schopen tato data generovat na základě uživatelsky nastavitelné hodnoty od které se odvíjí generování stochastických hodnot (random seed). V takovém případě je možné reprodukovat generovaná data a průběh již vyhodnoceného testu bez potřeby ukládat veškerá testovací data během každého provedení testu.

Generátory náhodných dat poskytují jednoduchý a efektivní způsob jakým otestovat rozsáhlé variace vstupních dat v rámci jedné testované komponenty. Problém nastává v případě, pokud se vstup skládá z více zdrojů. Hodnota vygenerovaných dat je tak ovlivněna nejen konfigurací vlastního generátoru, ale také hodnotou dat generovaných z jiného zdroje. Může tak dojít k situaci, kdy bude generováno množství (z hlediska podnikových pravidel) duplicitních testů, které pro účely ověření chování systému nebudou mít žádnou další vypovídající hodnotu.

3.1.3 Vnější zdroje dat

V rámci testu mohou být pro získání testovacích dat využity externí zdroje, tedy data, která nejsou přímou součástí nebo přílohou implementované části testu. Výhoda spočívá v získávání dat z centralizovaného zdroje, čímž je zjednodušen zápis samotného testu. Také je zajištěno využívání stále aktuálních testovacích dat, a to bez potřeby přímo zasahovat do implementace testu. Nevýhodu představuje závislost testu na aktuální dostupnosti externích zdrojů, kdy problém se síťovou konektivitou znemožňuje provedení testu.

Typickým představitelem externího zdroje je databázový systém. Ten je pro potřeby testů nutné nejdříve naplnit daty. V případě SQL databází mohou být data nejprve připravena v uživatelsky snadno upravitelné podobě (tabulkový procesor) a následně vyexportována do podoby SQL souboru, který zpravidla bývá přiložen jako součást testu. Pomocí tohoto souboru je posléze inicializován stav systému a následným voláním systémových funkcí jsou prováděny operace nad daty. Problém v takovém případě představuje perzistence, neboť pokud jsou v průběhu testu data modifikována, zůstanou v pozměněném stavu i po skončení testu, což vytváří odlišné prostředí pro spuštění následujícího testu. Tento přístup je využitelný pouze v případě postupného spuštění testů. Pokud existuje požadavek provádět v jeden moment více testů nad stejnými vstupními daty, pak je před nahráním dat do systému vždy třeba nejprve všechna data (pozůstatky z předchozích testů) vrátit do původního stavu.

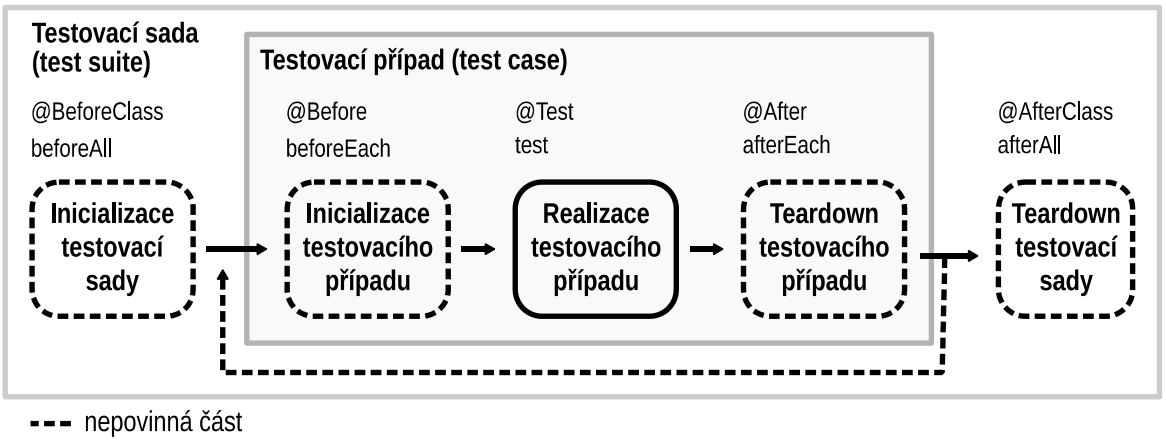
3.2 Popis implementace testu

Způsob, jakým jsou jednotlivé kroky testu zapsány se odvíjí od programovacího jazyka, testovacího frameworku nebo použitého testovacího nástroje. Pro popis softwarových testů se používá jedna ze dvou základní forem, kterými jsou:

- **Program**; tedy posloupností jednotlivých instrukcí, které přímo nebo nepřímo volají funkce testovaného systému. Typickým představitelem je programový zápis jednotkového testu pomocí testovacího frameworku (například JUnit nebo Jest).
- **Datová struktura**, která obsahuje potřebná testovací data, jež jsou následně pomocí dalšího nástroje do podoby programu přeložena a interpretována (například uložení projektu v nástroji SoapUI ve formátu XML). Posloupnost akcí je volena tak aby odpovídala krokům uvedeným v testovacím případu.

Softwarové testy začínají inicializační částí, během které se připraví vstupní data a inicializuje se testovaná komponenta. V případě integračních testů, kdy dochází k ověřování funkčnosti testované komponenty v závislosti na jiné části systému, která však ještě není implementována, nebo se nachází úplně mimo systém, lze za předpokladu znalosti jejího rozhraní vytvořit tzv. mock. Mock je objekt sdílející stejné rozhraní jako komponenta, jejíž chování se snažíme přizpůsobit potřebám daného testu. Tomuto objektu je možno pomoci programového kódu při zachování stejného rozhraní vynutit odlišnou funkčnost oproti původně zamýšlenému chování.

V rámci samotného vyhodnocování testu mohou být jednotlivé kroky průběžně doplňovány o kontrolu aktuálního vnitřního stavu testované komponenty nebo její výstupní hodnot. Neodpovídá-li hodnota očekávanému stavu je test okamžitě ukončen a vyhodnocen jako neúspěšný.



Obrázek 4: Blokové schéma uspořádání softwarového testu pro nástroje JUnit a Jest.[3]

Po dokončení testu může následovat fáze „anulace změň“ (teardown), během níž je testované prostředí na úrovni testovacího případu uvedeno zpět do stavu před testem. Typicky dojde ke smazání perzistentních dat, která by mohla negativně ovlivnit průběh následujícího testu.

3.3 Spuštění a provedení testu

Spouštění softwarových testů může z pohledu granulity probíhat na úrovni jednotlivých testovacích případů nebo výběru konkrétních testových sad. Z důvodu časové náročnosti

[3] Vlastní vizualizace popisu životního cyklu unit testu z https://junit.org/junit5/docs/current/user-guide/

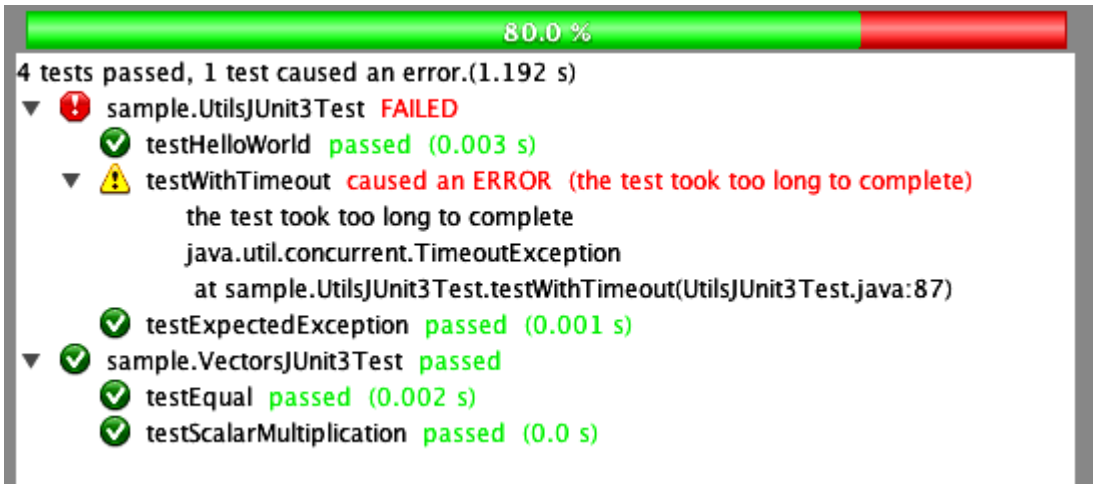
v podobě spuštění velkého množství testů je vývojáři a testery upřednostňováno spouštění testů pouze na úrovní právě testovaných komponent.

Samotné spuštění testů může být z pohledu vyhodnocování realizováno buď v sekvenci (sériově), kdy v jeden čas běží pouze jeden test nebo paralelně, čímž se výrazně snižuje doba provádění sady testů. Paralelní spouštění testů vyžaduje speciální přístup k implementaci testů, zvláště v případě, kdy jsou v průběhu testu modifikovány sdílené systémové prostředky nebo datové zdroje. Pokud při implementaci testů nebude brán zřetel na konkurentní přístup může dojít k tomu, že si testy budou vzájemně přepisovat data a pravděpodobně skončí s negativním nebo pozitivně negativním výsledkem, ačkoliv neobsahují přímo logickou chybu nebo odklon od scénáře popsaného v testovacím případu (Naik a Tripathy 2008, s. 394).

Nejčastějším využívaným principem pro souběžnost testů je spouštěním jednotlivých testů v samostatných vláknech. Aby nedošlo k zahlcení systému, na kterém běží testování nástroj, je nejprve potřeba nakonfigurovat maximální počet paralelně běžících vláken. Ve výchozím stavu bývá obvykle hodnota nastavena na hodnotu odpovídající počtu procesorů, kterými disponuje systémový hardware. Následně je spuštěná první dávka testů. V případě, že je test dokončen je vlákno uvolněno pro běh dalšího testu.

3.4 Vyhodnocení a prezentace výsledků

Vyhodnocení softwarových testů probíhá jak na úrovní jednotlivých testovacích případů, tak i na úrovní testovacích sad. Nejobecnější forma zápisu výsledku obsahuje název testu (testovací sady) a informaci o výsledku testu. Název by měl být dostatečně konkrétní, aby bylo možné daný test snadno dohledat a v případě potřeby jej spustit samostatně. Formulace výsledku na úrovni samotného testu nejčastěji obsahuje informaci o tom, zda daný test proběhl úspěšně nebo neúspěšně a zda byly v rámci testu naplněny všechny jeho předpoklady. Sofistikovanější nástroje jsou schopny měřit dobu běhu jednotlivých testovacích případů a v případě neúspěchu testu vypsat chybovou zprávu včetně stack trace.



Obrázek 5: Forma vyhodnocení výsledku testovacích sad v rámci NetBeans IDE[4]

Schopnost vyšetřovat nově způsobené defekty v rámci testu z dlouhodobého hlediska vyžaduje, aby výsledky byly průběžně ukládány. V případě, že jsou k dispozici data obsahující výsledky z předchozího spuštění testů, může být v závislosti na použitém na nástroji provedeno jejich porovnání.

[4] Zdroj obrázku: Writing JUnit Tests in NetBeans IDE https://netbeans.org/kb/docs/java/junit-intro.html

4 PROBLEMATIKA TESTOVÁNÍ SOFTWARU

Nejčastěji skloňovaným problémem při testování softwaru je bezesporu nedostatek konstruktivní komunikace, mezi členy týmu. Zvláště pak v případech, kdy jednotliví členové týmu působí v odlišných lokacích (Cruzes et al. 2016). Problematika se týká nejen členů testovacího týmu, ale také mezitýmové komunikace mezi analytiky, testery a vývojáři.

Další problém může zapříčinit nevhodná volba testovacího nástroje, respektive z něj plynoucích omezení. Některé nástroje obsahují grafické uživatelské rozhraní (GUI), jiné jsou implementovány v podobě frameworku, tedy rámcového řešení určeného pro vývojáře. Testy, které jsou popsány programem jsou mnohem lépe přizpůsobitelné, neboť jejich omezení často plynou pouze z omezení daného programovacího jazyka. Oproti tomu GUI nástroje jsou jednodušší na ovládání a mohou poskytovat dobré výstupy v podobě grafických nebo datových výsledků ve více formátech bez potřeby tyto schopnost vůbec implementovat.

Testovací nástroj je vhodné volit na základě, znalosti rozsahu projektu, použitých technologií a případů užití. Změna testovacího nástroje v pozdější fázi projektu může představovat zátěž v podobě potřeby transformace již vytvořených softwarových testů do nové podoby, což v závislosti na počtu a realizovaných testů představuje zdlouhavý a nákladný proces.

4.1 Souběžné testování v rámci jednoho prostředí

Rozsáhlé informační systémy se skládají z většího množství jednotlivých subsystémů a aplikací, v důsledku čehož je zapotřebí disponovat dostatečně výkonnými hardwarovými prostředky. V situacích, kdy není možné takové systémy spustil v rámci stejného zařízení, ze kterého probíhá testovaní, je třeba tuto situaci řešit vytvořením vzdáleného testovaného prostředí na samostatném zařízení disponující požadovaným HW. K takovému prostředí se následně každý člen vývojového týmu připojuje pomocí svých přístupových údajů.

Jedno prostředí, na kterém probíhá více testovaní je náchylné k problémům způsobených konkurentním přístupem popsaným v kapitole 3.3. V tomto případě je však systém dostupný i jiným uživatelům a situaci je tak třeba obvykle řešit ústní dohodou. Nabízí se možnost vytvoření dalšího testovaného prostředí, ale tento úkon bývá z finančního i provozního hlediska náročný a nelze jej opakovat do nekonečna.

4.2 Změna požadavku na systém

V případě implementace nových požadavků existuje unifikovaný proces jejich zavedení do systému. Požadavek je zanalyzován, následně naimplementován, otestován a zaveden do produkční verze systému. Ve skutečnost se však situace může zkomplikovat v případě, že dojde k upřesnění zadání ve specifikaci nebo ke změně již naimplementovaného a otestovaného funkčnosti systémové komponenty. Pokud se jedná o změnu funkcionality, která je v systému již delší dobu zvyšuje se riziko, že na implementaci dané systémové komponenty je závislá jiná část systému. V takovém případě může zásah způsobit „řetězovou reakci“ v podobě neúspěšného vyhodnocení některých stávajících testů (Schneidewind 1987, s. 304-307).

V takovém případě je potřeba dbát na to, aby se změnou specifikace byla znovu provedena i řádná analýza souvisejících částí systému. V opačném případě může dojít k posunutí termínu dodání systémových částí, které jsou závislé na nově zavedené funkčnosti dané komponenty, neboť je potřeba se nejprve vyřešit stabilitu stávající funkčnosti systému.

4.3 Jednotný repozitář testovacích dat

Pokud jde o sdílení zdrojového kódu systému v rámci vývoje existují pokročilé a hojně využívané nástroje pro jeho verzování a správu (například SVN nebo GIT). V rámci testovacích dat už je situace složitější. Testovací data se obvykle skládají z obrovského množství souborů, kde se řada z nich objevuje ve více variacích. To samo zvyšuje režii při jejich realizaci. Výjimkou pak není ani duplicita dat v rámci různých testovacích případů.

V praxi je běžným scénářem, že se tato data k testerům dostávají různým způsobem. Některá jsou zasílána v příloze emailu, další přicházejí skrze interní systémy jiné si mohou testeři vytvářet sami. Nejenom, že je potřeba udržovat stále aktuální verzi testovacích dat, ale také sdílet nově vytvořená data mezi členy tetovacího týmu. Další problematiku může představovat případ, kdy testování probíhá na více verzích vyvíjeného systému, které se liší podobou testovacích dat. Vzniká tedy potřeba tato data opatřit příznakem verze pro kterou jsou určena (Naik a Tripathy 2008, s. 398).

4.4 Nedeterministické výsledky testů

V případě, že jsou na projektu využívány softwarové testy, lze se setkat s úkazem, kdy je část testů v rámci opakovaného testování bez zjevné příčiny vyhodnocena neúspěšně, aniž by došlo k jakémukoliv zásahu do kódu systému. Pro tyto testy se vžil termín „flaky tests“. Opakovaným

spouštěním takových testů je dosaženo odlišných výsledků za dodržení zdánlivě stejných vstupních podmínek. S největší pravděpodobností se jedná o chybu uvnitř testu nebo v implementaci systému, která se projevuje pouze v některých případech, které ale nelze s úspěchem předpovědět.

Závažnější problém tyto testy způsobují, pokud se vyskytují ve větším množství. Tím se totiž zvyšuje pravděpodobnost, že v rámci testovací sady skončí alespoň jeden takový test neúspěchem, čímž bude negativně ovlivněno i celkové hodnocení výsledku testování. Možný je i opačný případ, kdy je test navzdory špatné implementaci vyhodnocen pozitivně.

Nejčastějšími příčinami nedeterministických výsledků je špatná práce s vlákny nebo asynchronními požadavky, nedostupnost nebo selhání aplikací třetích stran, závislost na pořadí, v jakém jsou testy spuštěny, využívání náhodných čísel, pozůstatek perzistentních dat z některého přechozího testu, odlišná konfigurace testovaného oproti lokálnímu prostředí atp. Některé z těchto příčin, lze detekovat pomocí debuggeru, jiné pomocí speciálních nástrojů (např. detekce neukončených vláken na konci testu) (Luo et al. 2014).

5 VÝZNAM AUTOMATIZACE TESTŮ

Hlavní důvodem pro zavedení automatizace je zvyšování produktivity dosaženého nahrazením časově náročné nebo opakující se lidské činnosti sofistikovanějším systémovým řešením. V důsledku toho je možné ušetřené prostředky (v podobě času a lidských zdrojů) vyčlenit na řešení problematiky, jež si žádá více tvůrčí přístup – jako je návrh, implementace a testování nových funkcí systému.

Důvody pro zavedení automatizace v rámci testování jsou následující:

- rychlejší provádění testů,
- snížení doby mezi testovacími cykly,
- zvýšení přesnosti testování,
- minimalizace úsilí při regresním testování.

5.1 Z pohledu vývoje

Pro vývojáře poskytují automatizované testy především výhodu ve formě zpětné vazby při zanášení změn do systému. Vytvářejí tak pomyslnou kontrolní vrstvu mezi programátorem a zdrojovým kódem, která by jinak chyběla a bylo by tím umožněno modifikovat části systému bez ohledu na celkovou funkčnost vyvíjeného produktu. V případě použití běžných programových testů je zodpovědnost za stabilitu systému podmíněna tím, že spuštění testů po jakémkoliv zásahu do funkčnosti systému nebude opomenuto.

Existuje několik momentů, které jsou pro spuštění testů kritické. Pro vývojáře je to v okamžik, kdy dokončil implementaci a podává požadavek na provedení změn v systému. V jednu chvíli může na zanesení změn do systému čekat více požadavků, které byly úspěšně otestovány oproti stejné vývojové verzi systému. Po zanesení změn dochází k ovlivnění funkčnosti systému, a tím do jisté míry i k zneplatnění dříve vyhodnocených testů. Proto dalším milníkem, pro spuštění testů je po zanesení změn do vývojové verze systému.

Po dokončení vývoje a testování, následuje fáze umístění systému do produkčního prostředí (deployment). Umístění do nového prostředí je doprovázeno množstvím úkonů, které je potřeba vykonat, tak aby bylo umožněno zavedení systému. Jde o instalaci potřebného softwaru a knihoven, konfiguraci a sestavení (build) projektu a jeho následné spuštění. Jedná se o relativně náročný proces, náchylný k chybám. Proto se i pro tuto činnost využívá automatizace ve formě spuštění sekvence předpřipravených skriptů, které provedou patřičná

nastavení a spuštění. V rámci této sekvence může být spuštěno testování pro ověření funkce nasazeného systému (Humble a Farley 2010).

5.1.1 Continuous Integration

Continuous Integration je proces automatizace sestavení software (build) do spustitelné podoby a následné provedení předem definované obslužné rutiny, jenž typicky zahrnuje i softwarové testy. K tomu dochází automaticky na základě provedení změn ve zdrojovém kódu a následným vytvořením tzv. feature větve v rámci použitého verzovacího systému (například Git nebo SVN), respektive repozitáře, který je pro takové chování nakonfigurován. Výhodou této metody je automatizovaný proces, který nijak nenarušuje běžnou činnost vývojáře (Duvall, 2007).

Testování probíhá asynchronně na vzdáleném serveru. Poté co je to dokončeno, je vývojáři zaslána notifikace s výsledky. Pokud testy proběhly úspěšně je umožněno pokračovat vytvořením tzv. pull-requestu, tedy požadavku v rámci verzovacího nástroje na zařazení změn do hlavní vývojové větve. V opačném případě je potřeba, aby byl zdrojový kód přepracován, jinak nebude umožněno sloučení s hlavní vývojovou větví. Tímto způsobem je zajištěno že vývojová větev, která obsahuje úpravy určené do produkce neobsahuje chyby.

5.1.2 Code coverage

Kvalitu vyvíjeného softwaru lze prověřovat nejen z hlediska plnění funkčních požadavků kladených na systém. Další metoda, která vede ke zvýšení stability výsledného softwarového produktu se nazývá code coverage. Tento způsob testování slouží k procentuálnímu vyčíslení pokrytí zdrojového kódu pomocí již vytvořených softwarových testů. Metoda spočívá ve spuštění testů, během nichž se sleduje průchod v rámci řídících struktur programu.

Na základě znalosti vnitřní struktury systému a částí programu, které byly během vykonávání testu provedeny, lze identifikovat části zdrojového kódu, které nejsou pokryty pomocí softwarových testů. Tyto části pak představují potencionální slabá místa systému. Obvykle dosažitelnou hodnotou je rozmezí mezi 40 a 50 procenty. Pro dosažená více než 80 % je potřeba vynaložit obrovské množství úsilí. (Desikan et al. 2008, s. 62)

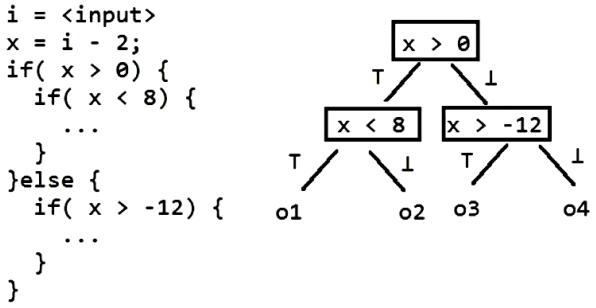
5.1.3 Concolic testing

Poněkud unikátní přístup ve smyslu automatizace v rámci testování přináší technika concolic testing (Concrete and Symbolic Testing). Zatímco všechny předchozí zmíněné metody v rámci této kapitoly se zabývaly tím, kdy a jakým způsobem spouštět již vytvořené softwarové testy, concolic testing řeší problematiku automatizace tvorby testovacích scénářů. Oproti běžnému

přístupu, kdy se testy vytvářejí na základě testovací dokumentace, je zde využito znalosti zdrojového kódu vyvíjeného softwaru – white box testing.

Technika vychází z principu obecné metody náhodného testování, která spočívá v generování náhodných hodnot na vstupu testované části programu. Zcela náhodně generované hodnoty by však měly za následek duplicitní průchody v rámci struktury programu. Při nízkém počtu replikací by tak s dostatečnou jistotou nebylo možné pokrýt všechny koncové stavy programu (návratové hodnoty nebo výjimky).

Jedním z nástrojů využívajících techniku concolic testing představuje Directed automated random testing (zkráceně DART – neplést s programovacím jazykem Dart od společnosti Google). DART provádí řízené generování náhodných vstupních hodnot, takovým způsobem, aby v rámci testovaného programu bylo dosaženo každého stavu právě jednou. Využívá k tomu princip tzv. symbolické exekuce, kdy každý výsledek rozhodovávání v rámci programu, lze zobrazit jako uzel uvnitř stromové struktury (viz Obrázek 6), kde kořen představuje stav se vstupními hodnotami a listy konečné stavy programu. Pomocí procesu zpětné symbolické exekuce lze ke konečným stavům odvodit odpovídající vstupní data (Godefroid et al. 2005).



Obrázek 6: Znázorněný čtyř konečných stavů programu dle hodnoty vstupních dat[5]

Pomocí tohoto přístupu lze tedy do jisté míry automatizovat tvorbu (například jednotkových) testů a zároveň poměrně jednoduchým způsobem navýšit code coverage zdrojového kódu systému. Nevýhoda této metody spočívá v tom, že se v konečném důsledku snaží o dosažení 100% hodnoty code coverage, což znamená provedení testu pro každé větvení programu, a tedy i enormní množství času stráveného testováním. V rámci programu mohou být prováděny operace, pro které nelze jednoduchým způsobem jejich vstupní data odvodit (například jednosměrné hashovací funkce nebo práce s ukazateli do paměti) v takovém případě může

[5] Odvození exekučního stromu pomocí metody symbolické exekuce (Filipovic, 2016).

odvozování vstupní hodnoty představovat proces s mnohonásobně vyšší třídou složitosti, jenž není možné v konečném čase realizovat.

Tento způsob testování nejčastěji objeví chyby spojené s neočekávaným formátem vstupu, které vedou do neočekávaného stavu programu (obvykle zakončený výjimkou – např. *NullPointerException*). Metoda také není primárně zaměřena na ověřováním chyb na úrovni podnikových pravidel vyvíjeného systému, ty je tedy stále potřeba mít pokryté jiným způsobem.

5.2 Z pohledu řízení projektu

V důsledku postupného rozšiřování produktu, jsou na členy vývojového týmu kladeny stále větší nároky na to udržet v povědomí o veškeré dosavadní implementované funkcionalitě systému. Po čase může nastat situace, kdy projekt naroste do takových rozměrů, že pod tíhou stále se opakujících procedur, které musí vývojový tým vykonávat, začne docházet k postupnému úbytku časového prostoru pro implementaci nových požadavků, čímž začne v rámci dané fáze vývoje klesat výsledná produktivita.

Z krátkodobého hlediska může být výhodné rozšířit tým o nové členy. V takovém případě je potřeba počítat s nárustem nákladů vynaložených na další vývoj, čímž ve výsledku dojde k pouhému oddálení stejného problému. Tím, jak se projekt posouvá dopředu je tedy potřeba neustále řešit otázku dostatku finančních a lidských zdrojů (Karhu et al. 2009, s. 207).

Jistou roli může sehrát také psychologie, kdy stojí za to zamyslet se nad tím, zda stále stejná a opakující se činnost nemůže u členů týmu vést k tomu, že po čase začnou brát jednotlivé úkony jako samozřejmost a nebudou jim věnovat dostatečnou pozornost. Zvláště v případě, kdy jednotlivec má alespoň povědomí o automatizačních nástrojích a, jen v rámci daného projektu nebylo rozhodnuto o jejich využití. V oboru jako jsou informační technologie, který se posouvá velmi rychle dopředu může jednotlivec nabýt dojmu, že taková práce není prospěšná ani projektu, ani jemu samotnému. Jedním z následných scénářů může být, že si člen týmu začne hledat nové místo, což zvláště pokročilé fázi projektu může pro vedení znamenat nepříjemnou situaci v podobě hledání dostatečně kvalifikované náhrady (Beecham et al. 2008, s. 21).

6 SOFTWAROVÁ ŘEŠENÍ PRO AUTOMATIZACI TESTŮ

Automatizované testování nemůže být realizováno bez vhodných nástrojů. V květnu roku 2018 provedla společnost Katalon průzkum[6], kterého se účastnilo více než 2 000 respondentů z oblasti testování softwaru. Mezi pět největších problému, které respondenti uvedli v souvislosti s testovacími nástroji, patří následující:

1. Příliš vysoká cena komerčních nástrojů – souhlasilo 71 % respondentů.
2. Častá změna v důsledku nové verze použitého nástroje vede k chybě v rámci již vytvořeného testovacího skriptu – souhlasilo 62 % respondentů.
3. Používání nástroje vyžaduje pokročilé znalosti programování – souhlasilo 54 % respondentů.
4. Nástroj vyžaduje příliš odborných znalostí a zkušeností – souhlasilo 52 % respondentů.
5. Chybí podpora pro testování nefunkčních požadavků – souhlasilo 49 % respondentů.

Naik a Tripathy (2008, s. 392-395) uvádějí následující seznam kritérií, na jejichž základě lze provést výběr vhodného testovacího nástroje:

1. **Vývoj** – Tvorba testu v rámci nástroje by měla probíhat uživatelsky přístupnou formou, popřípadě za využití snadno použitelného skriptovacího jazyku (neproprietárním způsobem – s využitím testovací nadstavby/frameworku).
2. **Údržba** – Nástroj disponuje funkcemi pro správu testovacích sad: procházení, verzování a kategorizaci testovacích případů.
3. **Spuštění** – Spouštění testů je umožněno v předem definovaných sekvencích, buď na úrovní jednotlivých testovacích případů nebo testovací sad.
4. **Vyhodnocení** – V průběh testu probíhá podrobné protokolování (logování) jednotlivých kroků testovacích případů. Uživatel je schopen tyto logy procházet, filtrovat a extrahovat výsledky v textové nebo grafické podobě. Jednotlivé záznamy v protokolu jsou opatřeny časovým údajem, tak aby je bylo možné využít jako podporu pro výkonnostní testování.

[6] Výsledky průzkumu nejvýraznějších problémů v rámci automatizovaného testování https://www.katalon.com/resources-center/blog/select-test-automation-tools-criteria/

5. **Správa** – Nástroj umožňuje uložení a nahrání testovací případů, ty je pak možné sdílet s ostatnímu členy tmu, kteří k nim přistupují na základě přidělené úrovně oprávnění.
6. **Testování GUI** – Při testování uživatelského prostředí systému nástroj umožňuje sledovat, ukládat a přehrávat uživatelem provedené akce (kliknutí myší, vstup z klávesnice) napříč různými testovanými prostředími.
7. **Dodavatel** – V rámci výběru dodavatele testovacího nástroje je třeba posoudit jeho finanční stabilitu, dobu působení na trhu, a také rozsah podpory vyvíjeného nástroje.

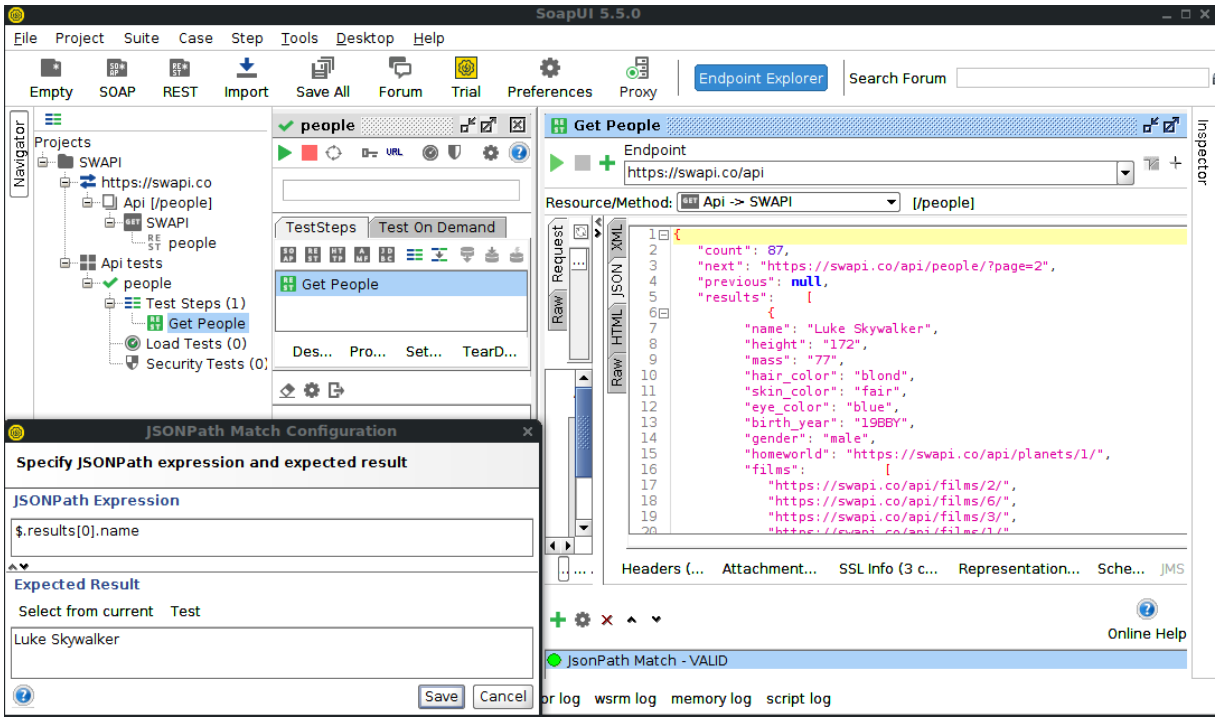
Na základě těchto informací jsou v následujících podkapitolách popsány nástroje SoapUI, Travis CI a Selenium. Jedná se o nástroje, které představují jednu z prvních voleb při výběru řešení pro daný typ testování. V základní licenci jsou tyto nástroje dostupná zdarma a práce s nimi není z uživatelského hlediska příliš náročná. Prezentované nástroje lze použít pro realizaci automatizovaného testování, každý však řeší problematiku testování jiným způsobem.

6.1 SoapUI

SoapUI je nástroj, který byl vyvinut v roce 2005 společností SmartBear Software. Zaměřuje se pouze na testování webových služeb založených na architekturách SOAP a REST. Mezi jeho přednosti patří snadná ovladatelnost, rozšířená komunita uživatelů a podpora integrace s řadou vývojářských nástrojů. Pro tvorbu základní podoby testů nejsou vyžadované znalosti programování, avšak v případě potřeby definice testovacího případu lze využít i skriptování pomocí jazyka Groovy.

Nástroj je k dispozici ve dvou licencích – open-source a komerční verzi Pro, která obsahuje pokročile funkce jako generování testů na základě popisu webové služby, nativní integrace do dalších nástrojů (například GIT, JIRA, Jenkins nebo TeamCity). Jako zdroje dat moou být využívány soubory ve formátu Excel nebo CSV a také databázové zdroje založené na ODBC.

SoapUI je napsán v jazyce Java a je dostupný pro operační systémy Windows, Linux a macOS. Grafické uživatelské rozhraní je realizováno pomocí knihovny Swing, díky čemuž trpí vzhled celé aplikace. Aplikace může na první pohled působit chaoticky (viz Obrázek 7), ale přes mnoho ovládacích prvků, které jsou hned viditelné, je práce s ním poměrně intuitivní.



Obrázek 7: Grafické uživatelské prostředí nástroje SoapUI

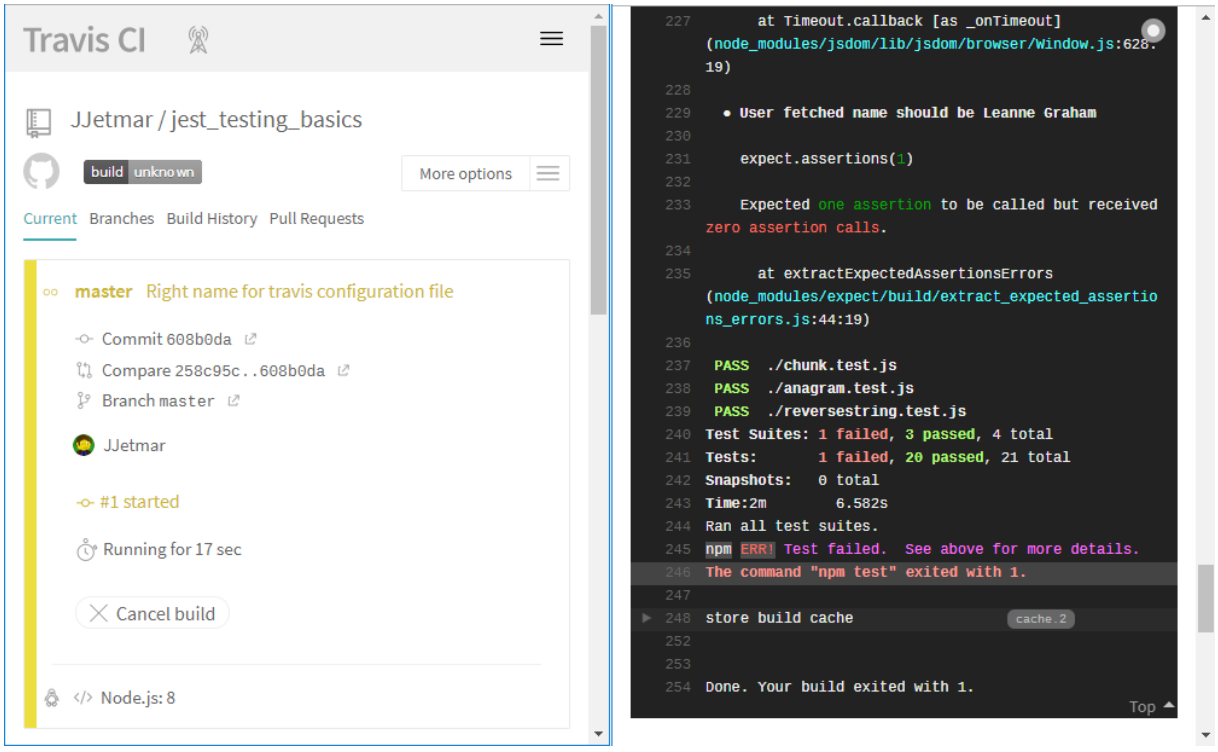
Pro tvorbu jednoduchého testu je třeba nejprve založit projekt a do něho přidat konfiguraci daného rozhraní, která se skládá z výčtu datových zdrojů. Také lze využít rozhraní webové služby, a to buď pomocí souboru WSDL nebo WADL. Poté je třeba vytvořit datovou sadu a do ní už je možné vkládat jednotlivé testovací případy obsahující jednotlivé kroky. V rámci kroku lze definovat volání datového zdroje webového rozhraní, následnou transformací nebo vyhodnocení (viz Obrázek 7).

Nástroj není přímo určen pro automatizované vyhodnocování testů. Nicméně díky možnosti definovat testy uživatelsky přívětivým způsobem se může jednat o dobrou volbu především v situaci, kdy tester neovládá tvorbu softwarových testů pomocí programovacího jazyka. V případě nutnosti lze automatizované spuštění testů realizovat pomocí příkazové řádky[7].

[7] Automate Functional API Tests https://www.soapui.org/test-automation/running-functional-tests.html

6.2 Travis CI

Travis CI je služba dostupná v rámci GitHub Marketplace[8] určená pro sestavení a testování projektů hostovaných v rámci platformy GitHub. Služba slouží jako nástroj pro Continuous Integration. Pracuje tedy pouze na základě změn ve zdrojovém kódu aplikace. Podmínkou je, že repozitář, v kterém je prováděna automatizace, musí být se službou spárován. Nastavení buildu probíhá pomocí souboru *.travis.yml*, který je umístěn v kořenovém adresáři repozitáře. Uvnitř souboru se nachází popis konfigurace, obsahující jednak použitý programovací jazyk, jeho verzi a dále seznam skriptů, které se mají vykonat.



Obrázek 8: Průběh a výsledek testu v rámci webového rozhraní nástroje Travis CI

Spuštění skriptů nastane při provedení během nahrání změn do repozitáře, nebo při vytvoření tzv. pull-requestu (požadavku na zařazení nových změn do dané větve).

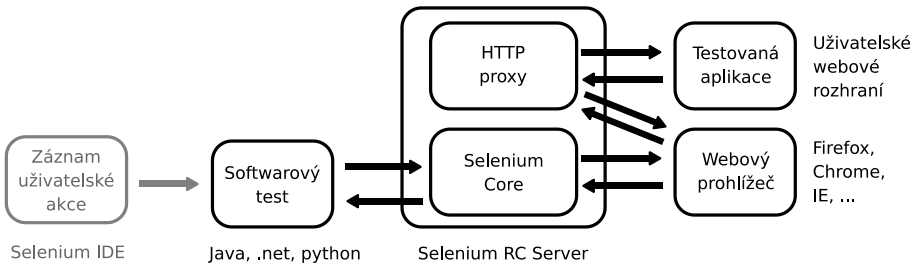
[8] Travis CI na GitHub MarketPlace https://github.com/marketplace/travis-ci

6.3 Selenium

Selenium je sada nástrojů určená pro automatizaci úkonů uvnitř webového prohlížeče.

Skládá se ze čtyř hlavních komponent:

- **Selenium IDE** – Rozšíření do webového prohlížeče (Mozilla Firefox nebo Google Chrome) pomocí něhož lze zaznamenávat aktivitu uživatele a ukládat ji a následně reprodukovat. Aktivita je zaznamenána pomocí Selenese – sady příkazů, která může být následně interpretována pomocí Selenium IDE. Výstup lze také exportovat do podoby programového testu (podporovány jsou testovací frameworky Java JUnit, Python pytest a JavaScript Mocha – viz Příloha A).
- **Selenium WebDriver** – Programové, objektově orientované aplikační rozhraní umožňující spouštět testy uvnitř prohlížeče mnohem efektivnějším způsobem než v případě Selenium IDE. Od verze 2.0 se jedná o součást nástroje Selenium. Implementaci rozhraní pro konkrétní webový prohlížeč je potřeba stáhnout samostatně. Dostupné jsou ovladače pro všechny běžně používané prohlížeče[9] v rámci operačních systému Linux, Windows a macOS.
- **Selenium RC (Remote Control)** – Komponenta umožňuje vzdálené spouštění testů. Tímto způsobem je možné centralizovat testovací prostředí pro spouštění testovacích skriptů (viz Obrázek 9). Díky tomu není potřeba, aby každý tester měl lokálně nainstalovány samotné webové prohlížeče a k nim potřebné ovladače.



Obrázek 9: Vyhodnocení testů pomocí Selenium RC (vlastní tvorba)

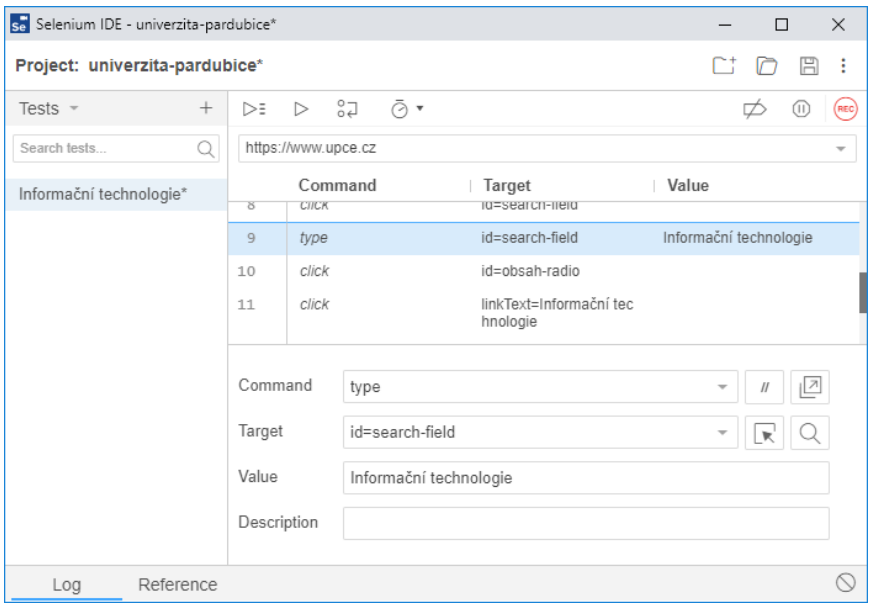
Zápis testu zůstává stejný jako v případě využití WebDriveru, s tím rozdílem, že místo ovladače pro webový prohlížeč je použit ovladač pro vzdálenou komunikaci

[9] Seznam dostupných Selenium WebDrivers - https://www.seleniumhq.org/download//#thirdPartyDrivers

s parametry pro připojení ke vzdálenému serveru, který potřebné ovladače obsahuje a zároveň obsahuje spuštěnou instanci Selenium RC Serveru.

- **Selenium-Grid** – Využití této komponenty umožňuje distribuované spouštění testů v rámci různých webových prohlížečů, operačních systémů anebo celých zařízení.

Pomocí nástroje Selenium IDE je tedy možné na základě běžného uživatelského používání systému odvodit testovací skript, který lze následně převést do podoby programového testu. Tím pádem lze výsledky testu vyhodnocovat stejným způsobem jako jiné softwarové testy. Problém je, že takto odvozené tesy probíhají bez čekání mezi jednotlivými akcemi (tedy jinak, než tomu bylo v případě uživatele).



Obrázek 10: Grafické uživatelské rozhraní nástroje Selenium IDE

Webová uživatelská rozhraní zpravidla využívají asynchronního zasílání a vyhodnocení požadavků. To znamená, že požadavek webového klienta je odeslán na server, a to bez toho, aniž by do té doby bylo používání webového prohlížeče nějakým způsobem blokováno. Simulace uživatelského čekání, musí být tedy do skriptu doplněna. V opačném případě může například nastat situace, kdy se nástroj snaží simulovat stisknutí tlačítka uvnitř modálního okna, kterému ale předchází animace, před jejímž dokončením ještě není dané tlačítko zobrazeno. Takový test skončí neúspěšně, tedy, jinak, než tomu bylo při spouštění v rámci nástroje Selenium IDE.

7 ARCHITEKTURA WEBOVÝCH SLUŽEB

Rozhraní v oblasti informačních technologií představuje zařízení nebo software umožňující komunikaci mezi dvěma prostředími. Rozhraní používaná v rámci webových služeb jsou v modelu ISO/OSI realizována na úrovni aplikační vrstvy nejčastěji za využití protokolu HTTP. Webové služby jsou obdobou technologií používaných pro vzdálené volání funkcí (jako například RMI). V prostředí internetu postupně nahrazují předchozí přístupy, a to díky jednotné formě výměně zpráv bez závislosti na technologii pomocí níž je systém implementován. Datová komunikace bezesporu tvoří nejdůležitější aktivitu v rámci servisně orientovaných informačních systémů.

7.1 Simple Object Access Protocol (SOAP)

Protokol pro zasílání zpráv SOAP byl vyvinut ve spolupráci společností Microsoft, Developmentor, IBM, Lotus a UserLand (Curbera et al. 2002). Protokol využívá principu takzvaného vzdáleného volání procedur (RPC) pro zasílání zpráv ve formátu XML. V rámci webových služeb využívajících SOAP se nejčastěji pro komunikaci používá protokol HTTP, respektive pak jeho metoda POST, která umožňuje odeslání zprávy v rámci těla požadavku.

Zpráva se skládá z následujících částí:

- **Envelope** – Označuje začátek a konec zprávy.
- **Header** – Část obsahuje volitelné atributy, slouží k modulárnímu rozšíření zprávy.
- **Body** – Povinný údaj obsahující data ve formátu XML určené pro příjemce.
- **Fault** – Záznam o chybě, případě že došlo během zpracování zprávy.

Hlavní výhodou protokolu SOAP je jeho standardizace a rozšířená podpora v rámci programových nástrojů. V případě dostupnosti souboru WSDL i snadná implementace při využití moderních IDE nástrojů, které jsou schopny vygenerovat všechny potřebné třídy a datové entity. Zpráva je přenášena ve formátu XML, který disponuje nástroji pro parsování, validaci (DTD, XSD) a transformaci (XLST).

7.2 Representational State Transfer (REST)

Architektura REST byla představena v roce 2000 v rámci disertační práce, jejíž autorem je Roy Fielding. Architektura je orientována na data, respektive datové zdroje a udává způsob jakým se k datům přistupuje. Každý takový zdroj (resource) je reprezentován URI adresou. Práce

s ním probíhá na základě typu použité HTTP metody a parametrů obsažených buď v těle zprávy nebo jako součást URI adresy. Používaný HTTP protokol obsahuje metody pro zasílání požadavku:

- **GET** – Slouží pro získání dat ze zdroje.
- **POST** – Používá se při tvorbě nových datových entit v rámci zdroje.
- **PUT** – Umožňuje modifikovat existující datovou entitu uvnitř zdroje.
- **PATCH** – Slouží pro částečnou modifikaci datové entity.
- **DELETE** – Odstraňuje danou datovou entitu ze zdroje.
- **HEAD** – Slouží k ověření existence datové entity rámci zdroje – Vrací odpověď se statusem 2xx/4xx[10] a prázdným tělem. Využitelné například v situaci, kdy chce klientská aplikace pouze ověřit, zda daný zdroj existuje, ale z důvodu velkého objemu dat není vyžadováno jeho celé načítání.

V rámci své práce Fielding (2000) doporučuje dodržet pří návrhu systému s využitím REST architektury následující omezení:

- **Jednotné rozhraní** – Je potřeba rozhodnout, které zdroje dat zůstanou uvnitř sytému a které budou přístupné skrze rozhraní. Dodržení jednotné konvence v pojmenování zdrojů.
- **Komunikace klient-server** – V rámci komunikace mezi dvěma systémy je každému exkluzivně přidělena role klienta anebo serveru. Klient začíná komunikaci, tím že odešle požadavek na server a následně čeká na odpověď. Server zaregistruje požadavek od klienta, zpracuje ho a vrátí klientovy odpověď. Tímto je komunikace ukončena.
- **Bezstavová komunikace** – Během komunikace nejsou v paměti serveru uchovávány žádná uživatelská data vztahující se ke aktuálnímu relaci. Uživatelský požadavek musí vždy obsahovat všechny potřebné údaje pro autentizaci.
- **Cachování** – Do komunikace mezi klienta a sever lze zapojit i zprostředkovatele s mezipamětí (cache). V případě, že je odpověď ze serveru označena jako „cacheable“, může tak dojít k znovupoužití daného zdroje bez potřeby se znovu dotazovat na server.

[10] Hodnoty HTTP status dle RFC 2616.10 – https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html

- **Úrovňový systém** – V rámci vyhodnocení klientského požadavku může server vystupovat v roli tzv. proxy a data zprostředkovat zaslání požadavku na jiný systém. Tímto způsobem se zvýší latence odpovědi od prvně volaného serveru. Také ale bude snížena potřeba vazby klienta na jiné systémy a data budou poskytnuta skrze jednotné rozhraní.

Co se týče nevýhod architektury REST jedná se o neekonomickou práci se zdroji. V případě, kdy je potřeba načíst data z více zdrojů je potřeba provést více klientských požadavků. Pokud jsou data z jednoho zdroje závislé na datech z jiného zdroje, je potřeba tyto požadavky vyřizovat synchronně, což znamená prodloužení uživatelské odezvy systému. Dalším problémem je, že neexistuje jednotný popis rozhraní.

Pro využití REST rozhraní, je potřeba znát URI každého koncového bodu (zdroje), který systém využívá a také způsob práce s ním. V praxi se tento nedostatek řeší externím popisem v rámci technické dokumentace nebo agregací koncových bodů a následným řešením v rámci serverové logiky, která ale ve výsledku navyšuje počet koncových bodů, čímž zvyšuje nepřehlednost datových zdrojů. Asi největším nedostatkem architektury REST je, že neexistuje žádný jednotný standard. Popis architektury je proveden formou „best practices“ a doporučení.

7.3 GraphQL

Technologie GraphQL byla v roce 2012 vyvinuta společností Facebook a následně v roce 2015 došlo ke změně licence na open-source. Samotné označení GraphQL má dva významy:

- **Jazyk GraphQL** – dotazovací jazyk používaný v klientské aplikaci pomocí něhož aplikace deklarativním způsobem popisuje definuje typ a výslednou podobu výstupních dat.
- **GraphQL runtime** – jedná se o rozšíření aplikační logiky serveru, která umožnuje porozumět a následně interpretovat klientské požadavky zapsané v pomocí jazyku GraphQL.

GraphQL byl navržen jako alternativa k architektuře REST, jenž řeší problémy zmíněné v kapitole 7.2. Klient pro implementaci, potřebuje znát pouze URI jediného koncové bodu, skrze který následné probíhá veškerá komunikace. To, z jakého datového zdroje jsou data vybírána je zmíněno v rámci dotazu jazyka GraphQL. Na úrovni dotazovacího jazyka lze také omezit zobrazení konkrétních parametrů v rámci datových objektů a tím v případě načítání většího množství objektů výrazně zredukovat velikost přenesených dat ke klientovi (Bruna 2016).

Práce s jazykem GraphQL, tedy připomíná práci s jazykem SQL z prostředí relačních databází s tím rozdílem, že GraphQL umožňuje implementaci klienta i v rámci webového prohlížeče a jedná se tak o vhodný nástroj využívaný v rámci tvorby webových rozhraní.

7.4 Způsob popisu rozhraní

V případě protokolu SOAP je veškerý popis rozhraní definován v pomocí souboru WSDL v rámci, kterého jsou uvedeny komplexní datové typy, volatelné procedury včetně jejich deklarace. V případě realizace rozhraní pomocí REST, už je situace poněkud složitější. Existují sice technologie, které nabízejí ekvivalent souboru WSDL pro popis rozhraní pomocí REST jako například WADL, dodnes však buď nebyly standardizovány a nejsou ve větší míře využívány. Existují také způsoby umožňující popis REST rozhraní pomocí WSDL (Mandel, 2008).

Další možností, jak popsat aplikační rozhraní typu REST je pomocí OpenAPI[11]. Jedná se o jednu z nejpropracovanějších specifikací definující způsob popisu REST API. Popis je realizován buď ve formátu JSON nebo YAML, díky čemuž je snadno strojově čitelný. Pomocí speciálních nástrojů mohou z takového popisu být generovány technické dokumentace.

Automatizaci v rámci popisu aplikačního rozhraní mohou poskytovat i projektové knihovny[12]. V tom případě se popis zpravidla řídí zdrojovým kódem. Využívá se buď speciálních dokumentačních komentářů nebo vlastnosti použitého aplikačního frameworku (například Java Spring Framework), který je schopen jednotlivé přístupové body ze zdrojového kódu odvodit (např. za pomocí anotací).

[11] Oficiální specifikace OpenAPI – https://swagger.io/specification/
[12] Např. knihovna Spring REST Docs – https://docs.spring.io/spring-restdocs/docs/current/reference/html5/

8 ANALÝZA A NÁVRH APLIKACE

Pro potřeby samotné automatizace existuje množství nástrojů, jež jsou schopny realizovat všechny běžné požadavky, které mohou být na automatizaci softwarových testů kladeny (viz kapitola 6). Metoda DART popsána v kapitole 5.1.3 nabízí zajímavý způsob, jakým lze automatizovat téměř všechny fáze testování – jednak jsou generována vstupní data a následně také testovací scénáře, které jsou později provedeny a vyhodnoceny. V tomto případě testování využívá znalosti vnitřní struktury systému. V rámci testování informačních systémů založených na webových službách je však velmi obtížné nalézt obdobný nástroj, který by dokázal to samé s pouhým popisem veřejného rozhraní, aniž by využíval znalosti zdrojového kódu systému (black box testování).

Pro účely testování rozhraní informačního systému založeného na webových službách je obvykle k dispozici dokumentace veřejného aplikačního rozhraní. V rámci dokumentace jsou pak vhodným způsobem popsány vstupní podmínky, pro které má použití daného koncového bodu rozhraní smysl (například pokud je nejprve potřeba v systému založit uživatele). V dokumentaci je také popsána podoba vstupních dat, avšak na základě pouhého popisu vstupu do systému, bez znalosti jeho vnitřní implementace, není možné předem odhadnout takovou podobu vstupních dat, která by měla za následek chybu nebo nečekané chování testovaného systému. Aplikace AutoTest tvoří prototyp nástroje, který je schopen provádět testování webových rozhraní pouze na základě obecného popisu vstupních dat.

8.1 Postup analýzy

V následujících kapitolách je popsán způsob zpracování analýzy aplikace AutoTest – vlastního řešení určeného pro potřebu automatizované tvorby testů pro koncové body webového aplikačního rozhraní. Z důvodu rozsahu analýzy vyvíjené aplikace, která by v plném znění mohla překračovat hlavní téma této kvalifikační práce – problematiku automatizovaného testování, je na analytický popis vyvíjené aplikace nahlíženo obecným pohledem. Detailnější popis je uveden pouze u vybraných artefaktů, které jsou významem důležité pro popis návrhu a implementace v dalších kapitolách.

Prvním krokem je definice klíčových slov a slovních spojení pomocí slovníku pojmů (kapitola 8.2), kde jsou popsány a definovány názvy a slovní spojení, které jsou používány napříč následujícími kapitolami. Následuje výčet funkčních požadavků (u vybraných z nich je uveden detailnější popis – kapitola 8.3) a nefunkčních požadavků (kapitola 8.4). Na základě požadavků jsou připraveny a popsány obecné skupiny případů užití. Další kapitoly se zaobírají

popisem analytického schématu tříd, popisujícího kandidáty na entity vyplívající z popisu podnikové domény.

8.2 Slovník pojmů

Při návrhu je potřeba vymezit a ustálit pojmy, které mají v rámci aplikace specifický význam, tak aby nedocházelo k zaměňování s jejich obecnější nebo oborově odlišnou formou významu.

- **Datová entita** – Struktura obsahující parametry s konkrétními hodnotami. Vzniká vygenerováním na základě Entitního schématu. Obsahuje stromovou strukturu parametrů (logické umístění parametrů).
    - Validní/nevalidní datová entita – Vygenerovaná datová entita, jejíž podoba odpovídá/neodpovídá popisu na základě entitního schématu.
- **Entitní rozhraní** – Automaticky vygenerované rozhraní architektury REST založené na entitních schématech. Lze využít jako zdroj vstupních dat pro softwarové testy.
- **Entitní schéma** – Uživatelem definovaná datová struktura popisující pravidla pro tvorbu datových entit. Je definována názvem a seznamem parametrů (reálné umístění).
- **JSON Schema** – Formát pro popis a validaci datových objektů ve formátu JSON.[13]
- **Parametr entitního schématu** – Parametr je definován názvem, datovým typem a validačním omezením.
- **Test** –Test definován pomocí URI datového zdroje REST rozhraní testovaného sytému a entitním schématem. Princip testování je založen na postupném generování datových entit, jejichž hodnoty parametrů jsou náhodně generovány na základě kombinatoricky odvozených stavů platnosti vycházejících z omezení, jež jsou dány entitním schématem.
- **Testovací běh** – Proces testování pracující s konkrétní datovou entitou a koncovým bodem rozhraní. Odesílá požadavek na testovaný systém a vyhodnocuje odpověď.

[13] Oficiální stránky JSON Schema – https://json-schema.org/

8.3 Funkční požadavky

Seznam funkčních požadavků vychází z podmínek pro potřebu realizace způsobu testování, který je popsán v úvodu kapitoly 9. Obecnou formou popisuje, jaké podmínky na datové vstupy a funkční stránku vyvíjeného systému. Funkční požadavky jsou kategorizovány do šesti skupin viz Tabulka 1.

Tabulka 1: Skupiny funkčních požadavků aplikace AutoTest

Označení	**Název skupiny**	**Popis**
REQ001	Správa entitních schémat	Aplikace umožní vytvořit, upravit a odstranit entitní schémata.
REQ002	Struktura entitního schématu	Pravidla pro definici entitního schématu.
REQ003	Generování datových entit	Aplikace umožní generovat datové objekty na základě entitních schémat.
REQ004	Správa testů	Aplikace umožní vytvoření, spuštění a odstranění testu.
REQ005	Entitní rozhraní	Aplikace zprostředkovává rozhraní architektury REST, pro načtení náhodně generovaných datových entit. Takové rozhraní pak bude možné využívat zejména prostřednictvím softwarových testů pomocí volání webových služeb.
REQ006	Architektura klient-server	Aplikace bude rozdělena na klientskou a serverovou část s veřejným aplikačním rozhraním, které bude v budoucnu možné využít pro potřebu integrace s dalšími nástroji.

V tabulkách 2 a 3 jsou pro potřeby ilustrace uvedeny detailní popisy požadavků pro skupiny požadavků REQ002 a REQ004.

Tabulka 2: Popis skupiny požadavků REQ002 – Struktura entitního schématu

Skupina požadavků: **Struktura entitního schématu**		Priorita	Komplexita	Verze
REQ002-1	**Entitní schéma**	Vysoká	Vysoká	1.0
Entitní schéma je definováno unikátním názvem a výčtem parametrů. Název odpovídá regulárnímu výrazu: ^[a-zA-Z](-?[0-9a-zA-Z]+)*$ Parametr je definován v požadavku REQ002-2.				

Skupina požadavků: **Struktura entitního schématu**		Priorita	Komplexita	Verze
REQ002-2	**Entitní schéma – parametr**	Střední	Střední	1.0
Parametr je reálně umístěn v rámci výčtu parametrů uvnitř entitního schématu. V rámci logického umístění se jedná buď o přímého potomka entitního schématu nebo jiného parametru. Parametr je definován: • **Názvem** – název parametru odpovídá regulárnímu výrazu ^[a-zA-Z](-?[0-9a-zA-Z]+)*$. • **Umístěním** – unikátním spojením názvu a logického umístění v rámci struktury entitního schématu. **Datovým typem** – viz požadavek REQ002-3.				
REQ002-3	**Entitní schéma – datový typ parametru**	Střední	Střední	1.0
Datový typ parametru může nabývat hodnot: *Object*, *String*, *Integer*, *Number* a *Boolean*. Typu *Object* nabývá parametr pouze tehdy, pokud se v rámci logické struktury jedná o předka jiného parametru. Z hodnoty datového typu plynou možná nastavení omezení hodnot pro generované hodnoty parametru (viz REQ002-4).				
REQ002-4	**Entitní schéma – omezení datového typu**	Střední	Nízká	1.0
Na základě datového typu parametru je možné pro generované hodnoty parametru definovat následující omezení: • Pro všechny datové typy: příznak, zda je parametr povinný. • Pro *String*: minimální délka, maximální délka a omezení hodnot pomocí regulárního výrazu. • Pro *Integer* a *Number*: minimální a maximální číselná hodnota.				

Tabulka 3: Popis skupiny požadavků REQ003 – Generování datových entit

Skupina požadavků: **Generování datových entit**		Priorita	Komplexita	Verze
REQ003-1	**Datová entita**	Střední	Střední	1.0
Datová struktura vygenerované entity odpovídá logické struktuře parametrů daného entitního schématu. Hodnoty jednotlivých parametrů jsou generovány na základě jim přidělených datových typů a omezení.				
REQ003-2	**Datová entita – generovaní na základě seedu**	Nízká	Nízká	1.0
Generování pseudonáhodných hodnot parametrů datových entit je umožněno na základě seedu ve formě textového řetězce.				
REQ003-3	**Datová entita – validní/nevalidní podoba**	Střední	Střední	1.0
Generování datových entit je umožněno jak validním způsobem (splňující logickou strukturu parametrů, datové typy a omezení definované v rámci entitního schématu), tak i nevalidním způsobem (pomocí záměrného generování nevalidních hodnot parametrů).				

8.4 Nefunkční požadavky

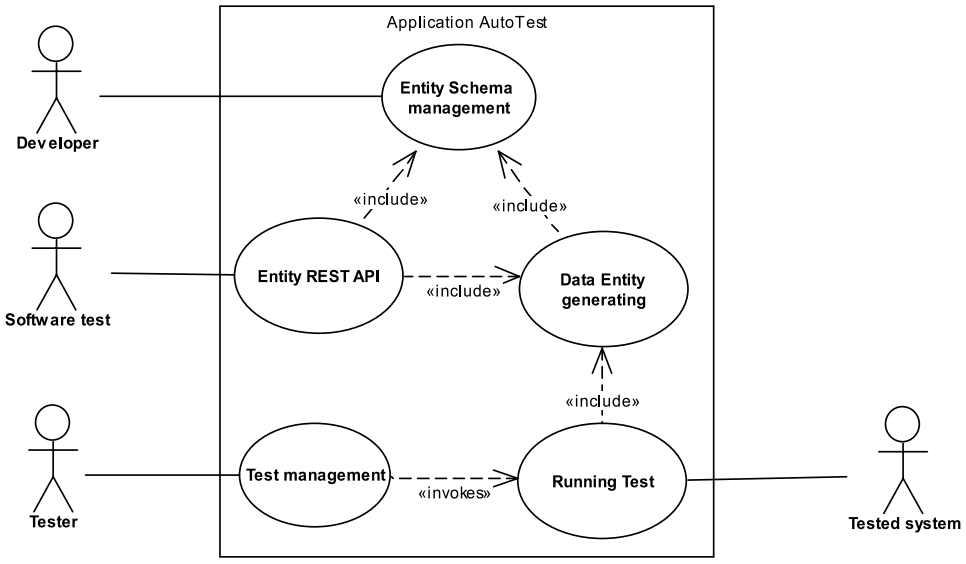
Následující seznam nefunkčních požadavků (Tabulka 4) blíže popisuje omezení kladená na vývoj a výslednou podobu aplikace AutoTest.

Tabulka 4: Nefunkční požadavky aplikace AutoTest

Požadavek	**Název**	**Popis**
REQ101	Provoz aplikace	Použité technologie na tvorbu aplikace budou voleny, tak aby i bylo možné výsledné řešení provozovat jak lokálně, tak i z prostředí vzdáleného webového serveru.
Požadavek	**Název**	**Popis**
REQ102	Závislost na prostředí (server)	Spuštění aplikace bude umožněno v rámci obecně nejpoužívanějších operačních systému – Linux, Windows, macOS.
REQ103	Závislost na prostředí (klient)	Klient bude realizován pomocí webového rozhraní s podporou 90 % nejvíce používaných webových prohlížečů jak na stolních počítačích, tak i na mobilních zařízeních.
REQ104	Ovladatelnost uživatelského rozhraní	V rámci uživatelské rozhraní bude možné obsluhovat a realizovat testy i bez potřeby znalosti programování.

8.5 Případy užití a hranice systému

Schéma případu užití (Obrázek 11) jasně vymezuje hranici problematiky, kterou se aplikace AutoTest zabývá. Dále jsou v rámci schématu vymezeni aktéři – představující zdroje vnějších interakcí se systémem.



Obrázek 11: Obecné případy užití aplikace AutoTest

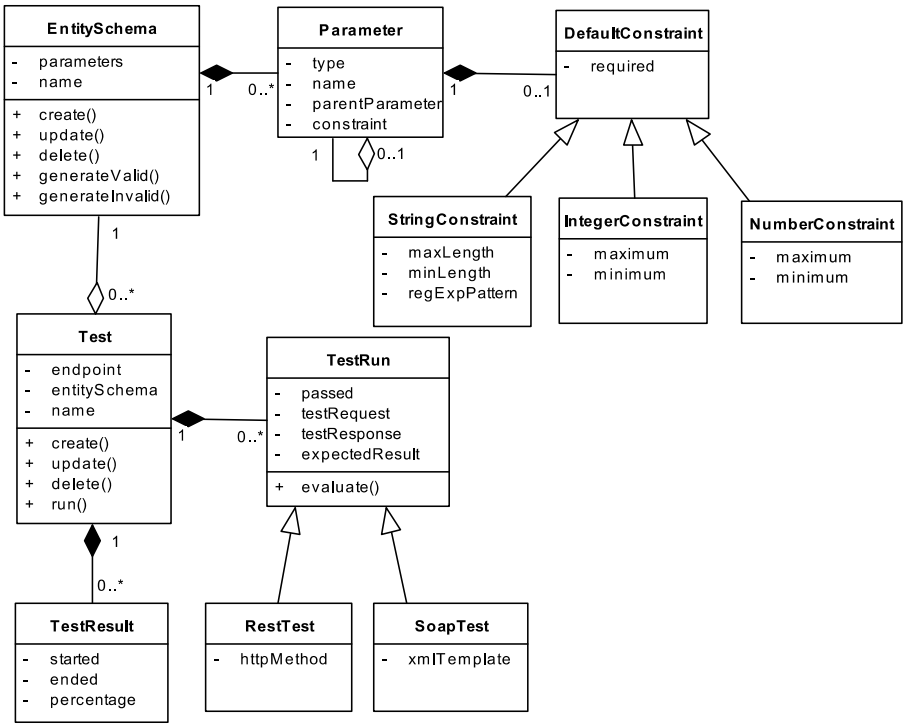
Následuje výčet jednotlivých případu užití:

- **Data entity generating** – Generování datových entit na základě předem definovaných pravidel a struktury – entitního schématu (viz skupina požadavků REQ003).
- **Entity REST API** – Pro každé vytvořené entitní schéma je pomocí aplikačního rozhraní architektury REST vytvořen datový zdroj, pomocí něhož lze zpřístupnit generované entity i mimo hranice aplikace AutoTest.
- **Entity Schema management** – Uživatel v hlavní navigaci aplikace vybere položku „Entity Schemas“, následně je uživateli zobrazen přehled existujících entitních schémat. Každá vyobrazená položka obsahuje tlačítka „Update“ pro úpravu a „Delete“ pro odstranění daného entitního schématu. (viz Obrázek 13).
    - **Create** – Nad přehledem je zobrazeno tlačítko pro tvorbu nového entitního schématu. Po kliknutí na něj je uživateli zobrazen formulář pro zadání názvu a definice nového entitního schématu (viz Obrázek 14). Změnu struktury dat, lze realizovat přetažením položky parametru na nový rodičovský parametr. Tvorbu nového entitního schématu je třeba potvrdit tlačítkem „Create“.
    - **Update** – Stisknutím tlačítka „Update“ lze zobrazit formulář pro úpravu entitního schématu. Práce s formulářem probíhá stejně jako v případě tvorby nového entitního schématu. Úpravu je následně potvrzena tlačítkem „Update“ a upravená položka je zobrazena v přehledu.
    - **Delete** – Kliknutím na tlačítko „Delete“ je zobrazeno upozornění a výzva o potvrzení smazání daného entitního schématu. Po potvrzení stiskem tlačítka „OK“ je záznam odstraněn ze přehledu zobrazených položek.
- **Running Test** – Spuštěním testu dochází k vytvoření testovacích běhů. V rámci každého běhu je vygenerována datová entita, která je ve formě HTTP požadavku odeslána na vstup do koncového bodu aplikačního rozhraní, který je definován v rámci tvorby testu. Na základě výsledku validity datové entity a odpovědi serveru (respektive jeho HTTP statusu) je běh testu označen jako úspěšný nebo neúspěšný. Souhrnné výsledky z jednotlivých běhů jsou pak uloženy do přehledu testu.
- **Test management** – Správa testů probíhá obdobným způsobem jako v případě užití **Entity Schema management** s tím rozdílem, že vytvořené testy je možné i spouštět.

- **Run** – Výběrem možnosti „Run“ v přehledu vytvořených testů lze test spustit. Test je realizován serverovou částí. Poté co je test vyhodnocen, je souhrnný výsledek přidán do sekce „Test results“, která je obsažena v hlavní navigaci aplikace.

8.6 Analytický model tříd

Analytický (nebo také doménový) model tříd aplikace AutoTest (viz Obrázek 12) vznikl na základě textového popisu řešené problematiky – nejedná se o vstup do implementace. Vyhledáváním podstatných jmen v popisu požadavků (kapitola 8.3) a podnikové problematiky (jenž je blíže popsaná v kapitole 9) byly odvozeny kandidátní třídy, popřípadě jejich atributy. Pomocí sloves, tvořících přísudek k vybraným třídám jsou odvozeny operace a typ vazby mezi třídami.

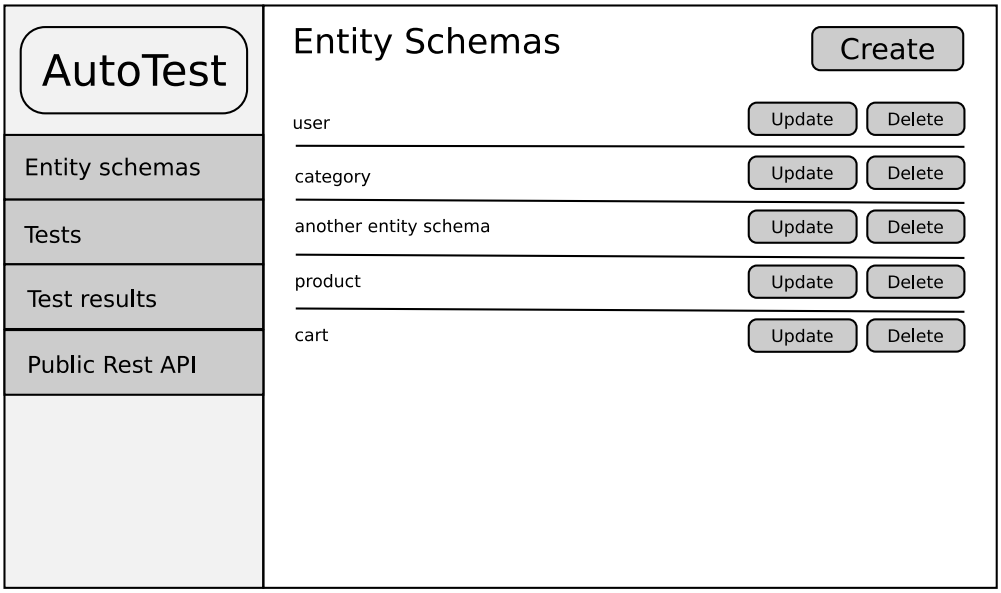


Obrázek 12: Analytický model tříd aplikace AutoTest

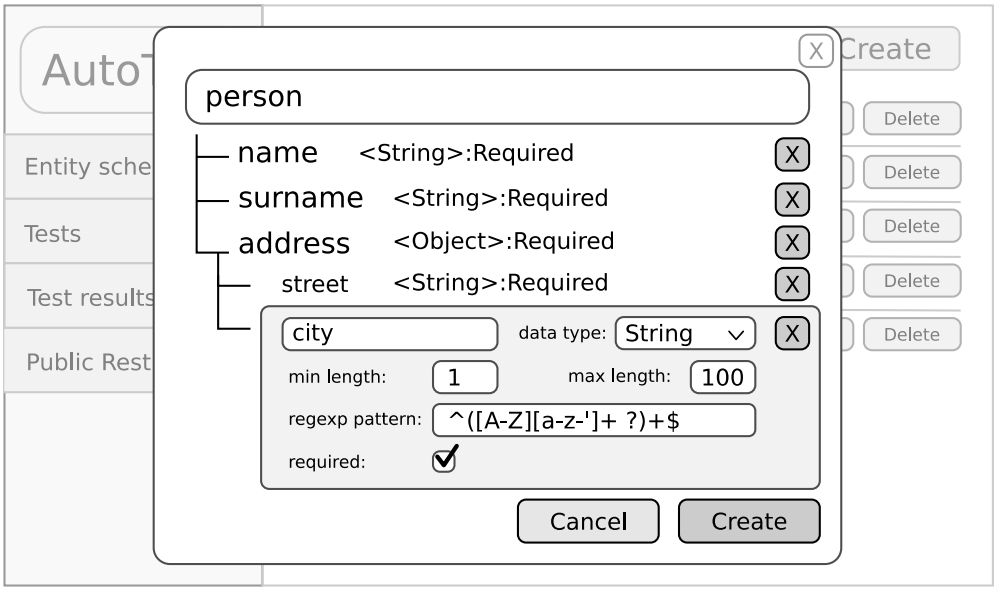
8.7 Návrh uživatelského rozhraní

Požadavek REQ103 uvádí, že uživatelské rozhraní aplikace AutoTest má být realizováno pomocí webového rozhraní. Z tohoto důvodu je při rozvržení grafického designu přihlédnuto

k problematice související s návrhem moderních webových aplikací. To se projevuje zejména při volbě a umístění jednotlivých komponent takovým způsobem, aby byla aplikace dobře použitelná na různých typech zařízeních (s odlišným rozlišením, jemností obrazu nebo orientací displeje na výšku/šířku). Pro lepší představu o následném grafickém zpracování byly vytvořeny wireframy uživatelského rozhraní aplikace AutoTest (Obrázek 13 a Obrázek 14).



Obrázek 13: Wireframe – Přehled entitních schémat



Obrázek 14: Wireframe – Tvorba nového entitního schématu

9 APLIKACE AUTOTEST

Aplikace AutoTest byla navržena s cílem vyplnit místo na poli automatizace testů informačních systémů založených na webových službách. Testování je realizováno pomocí automaticky generované sady testů individuálně pro jednotlivé koncové body (endpoints) testovaného webového rozhraní. Koncové body mohou být realizovaný pomocí webové služby SOAP nebo architektury REST API.

Pro samotné testování je využíváno popisu vstupních dat, jenž je do aplikace nejdříve zadána uživatelem ve formě entitních schémat. Samotný test pak propojuje entitní schéma s konkrétním koncovým bodem testovaného rozhraní. Při spuštění testu dochází k průběžnému generovaní variací vstupních dat a testuje se odpověď testovaného systému. Na základě informace o tom, zda jsou generovaná data (dle entitního schématu) validní a odpovědi ze strany testovaného rozhraní, je následně rozhodnuto o úspěšnosti provedeného testu. Poté co jsou otestovaný všechna generovaná vstupní data. Je uživateli zobrazen výsledek ve formě procentuální úspěšnosti celého testovacího běhu.

9.1 Vývojové prostředky

Pří výběru technologií pro vývoj aplikace AutiTest bylo přihlíženo k nefunkčním požadavkům, které plynuly především z potřeby provozu aplikace nezávisle na operačním systému, realizace uživatelského rozhraní prostřednictvím webového rozhraní.

Celkové řešení aplikace se skládá ze dvou subaplikací, pro jejichž vývoj byly použity (částečně) odlišné implementační prostředky. Níže jsou uvedeny vybrané technologie, jazyky a nástroje, které byly využity pro vývoj aplikace .

9.1.1 Node.js a Koa

Node.js umožňuje tvorbu serverových aplikací pomocí jazyka JavaScript, známého původně z prostředí webových prohlížečů. Existuje řada frameworků, které poskytují knihovny, nástroje a postupy pro řešení problematiky vývoje serverových webových aplikací (v podobě routování adres, tvorby vlastního middlewaru apod.). Pravděpodobně neznámějším rámcovým řešením

pro tvorbu dynamických webových stránek nebo aplikačních rozhraní v prostřední Node.js je framework Express[14].

Pro vývoj aplikace AutoTest byl použit framework Koa[15]. Jedná se o odlehčenou verzi frameworku pro tvorbu webových aplikací, který od počátku svého vzniku nabízel lepší podporu syntaxe dle specifikace ECMAScript verze 6. Tím přináší například i snadnější manipulaci s asynchronními událostmi než rozšířenější framework Express. Jedná se minimalistickou verzi nástroje, jehož rozsah zdrojového kódu nepřesahuje jednotky stovek řádků kódu. Koa se dlouhodobě umísťuje na prvních příčkách v rámci výkonnostních testů mezi frameworky pro webové aplikace postavených na platformě Node.js.

9.1.2 Ant Design Pro

Uživatelské rozhraní je realizováno formou webové aplikace. Webové prostředí je postaveno na knihovně Ant Design Pro[16] (respektive její nástavbě AntD Admin[17]), která dále využívá knihovnu pro tvorbu uživatelských rozhraní React[18] a framework pro řízení stavu (pomocí stavového kontejneru) aplikace a komunikaci mezi vizuálními komponentami – DvaJS[19].

Jedná se o souhrn frontendových knihoven a komponent určených pro tvorbu webových řešení, které jsou upraveny do podoby jednotného vizuálního stylu. Nevýhodu může představovat komunita stojící za vývojem knihovny, která komunikuje a píše dokumentaci převážně v čínském jazyce.

9.1.3 JSON Schema Faker

Knihovna JSON Schema Faker[20] slouží pro generování datových objektů ve formátu JSON na základě popisu struktury a pravidel popsaných pomocí JSON Schema. V rámci aplikace AutoTest je tato knihovna využívána pro generování datových entit. Uživatelem definované entitní schéma je převedeno do formátu JSON Schema, které je následně přetransformováno do výsledné podoby datové entity. Knihovna sama o sobě umožňuje generovat výsledek pouze ve validní podobě. V rámci aplikace AutoTest je tedy funkce rozšířena, tak aby bylo umožněno generovat data i v nevalidní podobě.

---

[14] Webové stránky frameworku Express.js - https://expressjs.com/
[15] Webové stránky frameworku Koa - https://koajs.com/
[16] Oficiální stránky knihovny Ant Design - https://pro.ant.design/
[17] Dokumentace řešení pro tvorbu frontendových aplikací AntD Admin - https://doc.antd-admin.zuiidea.com
[18] Oficiální stránky knihovny React - https://reactjs.org/
[19] Webové stránky projektu DvaJS - https://dvajs.com/
[20] Repozitář projektu JSON Schema Faker - https://github.com/json-schema-faker/json-schema-faker/

9.1.4 MongoDB

MongoDB je dokumentově orientovaný NoSQL databázový systém. Oproti tradičnějším SQL systémům, kde jsou data ukládána ve formě tabulek a relací mezi nimi, jsou data v případě databázového systému MongoDB realizována ve formě tzv. dokumentů. Dokument je reprezentován strukturou, která nemusí mít pevně danou formu, jako je tomu v případě tabulek. Záznamy, které jsou spojeny pomocí kompozitního vztahu je tak možné načítat najednou v rámci jednoho dotazu. Odpadá tak potřeba restrukturalizace v případě jejich načítání nebo ukládání, což se významně podepisuje na rychlosti celého databázového systému, zvláště v případech, kdy je databázoví server umístěn vzdáleně.

9.1.5 Docker

Moderní vývoj využívá specifické formy virtualizace – softwarové kontejnery. Nejznámější implementací je nástroj Docker, který je použit také v této práci pro virtualizaci jednotlivých vývojových prostředí pomocí tzv. kontejnerů, tedy izolovaných nezávislých prostředí, pro které lze pomocí konfiguračních souborů nastavit pravidla pro vzájemné viditelnosti. V rámci každého kontejneru je daná aplikace spuštěna nezávisle na operačním systému. Díky tomu je možné v rámci jednoho vývojového stroje spouštět více instancí aplikací, kdy každá má přístup ke svým vlastním datovým a systémovým prostředkům. Toto opatření minimalizuje narušení integrity systému z vnějšího prostředí a zároveň umožňuje snadný vývoj v rámci jednoho fyzického zařízení.

9.1.6 Vývojové prostředí

Vývoj všech částí aplikace probíhal ve vývojovém prostředí IntelliJ IDEA[21]. Prostředí je původně určeno pro programovací jazyk Java a nabízí podporu pro související technologie (např. Maven, Gradle nebo Tomcat), avšak pomocí vhodných zásuvných modulů je možné rozšířit možnosti prostředí o další jazyky a vývojové nástroje. IntelliJ IDEA disponuje možnostmi analýzy zdrojového kódu a následného našeptávání čímž vývojáři značně zvyšuje komfort při vývoji.

[21] Webové stránky vývojového prostředí InteliJ IDEA - https://www.jetbrains.com/idea/

9.2 Struktura řešení praktické části práce

Řešení praktické části diplomové práce, které je přiložené na datovém médiu lze rozdělit do čtyř částí, které jsou blíže popsány v následujících podkapitolách:

- **server** – serverová část aplikace AutoTest (backend),
- **client** – webové, uživatelské rozhraní AutoTest (frontend),
- **test-server** – testovaná aplikace,
- **software-test** – softwarový test vytvořený pomocí testovacího frameworku Jest. Slouží pro demonstraci využití veřejného rozhraní z prostředí softwarového testu.

Postup zprovoznění jednotlivých částí je popsán v příloze A.

9.2.1 Webové rozhraní (client)

Webové rozhraní je realizováno pomocí samostatně běžící aplikace (single-page application), která zasílá požadavky na serverovou část. V rámci implementace jsou využity vizuální komponenty obsažené uvnitř použitého frameworku Ant Design Pro. Za účelem zpracování datových struktur specifických pro aplikaci AutoTest byly vytvořeny vlastní vizuální komponenty (například pro definici a zobrazení entitního schématu).

Hlavní navigace webového rozhraní je rozdělena do čtyř částí:

- **Entity schemas** – tato část aplikace je určena pro správu entitních schémat.
- **Tests** – v této sekci je možné spravovat a spouštět testy.
- **Test results** – v této části jsou uveřejněné výsledky dokončených testů, obsahující údaje z jednotlivých testových běhů.
- **Public REST API** – tato sekce aplikace slouží jako jednoduchá technická dokumentace pro možnosti využívat dynamického aplikačního rozhraní aplikace AutoTest založeného na vytvořených entitních schématech.

9.2.2 Serverová část aplikace AutoTest (server)

Ačkoliv lze webové uživatelské rozhraní (kapitola 9.2.1) spustit samostatně, není možné jej reálně provozovat bez datové složky, kterou v modelu třívrstvé architektury zprostředkovává aplikační server. V rámci serverové části aplikace AutoTest probíhá vyhodnocení veškeré obchodní (business) logiky a práce s perzistentní datovou vrstvou celé aplikace. Tím je zajištěna stabilita a datová konzistence celého řešení.

Výsledné řešení dodržuje zásady vývoje založené na základě architektury MVC (Model, View, Controller) a obsahuje rozhraní založené na architektuře REST, primárně určené pro volání z klientské části aplikace:

- **Model** – modelová vrstva pracující s daty je z velké části realizována pomocí knihovny Mongoose – řeší komunikaci s perzistentní datovou vrstvou aplikace (databázový systém MongoDB). Knihovna dále umožňuje práci s datovými objekty pomocí objektově orientovaného přístupu.
- **View** – tato část je blíže představena v kapitole 9.2.1 – Webové rozhraní (client).
- **Controller** – Představuje programovou komponentu aplikace, která je volána na základě klientského požadavku. Zde je také obsažena veškerá logika aplikace.

9.2.3 Testované rozhraní (tested-server)

Testovaným rozhraním se v tomto případě rozumí jednoduchá aplikace. Byla vytvořena pouze za účelem demonstrovat proces testování pomocí aplikace AutoTest. Implementace testované aplikace obsahuje jednoduché rozhraní architektury REST, které obsahuje pouze jeden koncový bod (endpoint). Funkční implementace testované aplikace, záměrně obsahuje chyby/nedodělky, které by při testování pomocí aplikace AutoTest měly být odhaleny (Příloha D).

Testované rozhraní také splňuje omezení, která jsou popsána v rámci případu užití „Running Test“ (kapitola 8.5) – tedy aplikace je schopna ověřit validitu vstupu a zároveň vyhodnotit chybu, ať už interní (způsobenou nepozornosti při implementaci) nebo uživatelskou pomocí vhodně zvoleného HTTP statusu v rámci odpovědi (viz kapitola 7.2).

9.2.4 Veřejné rozhraní datových zdrojů

Veřejné rozhraní datových zdrojů je dynamicky generované rozhraní aplikace AutoTest založené na architektuře REST, které lze využít například v rámci scénáře softwarového testu. Tato část rozhraní není v rámci webového uživatelského rozhraní využívána a slouží primárně pro simulaci práce s datovými zdroji, jež jsou založeny na vytvořených entitních schématech.

Tento přístup umožňuje v rámci softwarového testu realizovat testovací scénáře založené na krocích spojených s využíváním datových zdrojů, které ale v době spuštění testu buď nejsou vyvinuté, anebo nejsou dostupné.

Využitím této veřejné časti aplikačního rozhraní je možné centralizovat zdroje dat pro potřeby softwarových testů. Tímto způsobem dochází ke snížení počtu potřebných zásahů do zdrojového kódu softwarového testu v případě, že je potřeba změnit strukturální podobu vstupních dat do vyvíjeného systému. Rozhraní je generováno dle standardních operací s datovými zdroji prostřednictvím http požadavků využívající metody GET, POST, PUT a DELETE (více popsáno v kapitole 7.2).

9.3 Tvorba vstupních dat pro testování systému

Tvorba vstupní dat pro testování systému pomocí aplikace AutoTest je realizována na základě vytvořeného entitního schématu. Uživatelská tvorba entitních schémat je umožněna pouze pomocí formuláře v uživatelském rozhraní aplikace.

9.4 Generování testovacích běhu

Počet jednotlivých testovacích běhů je závislý na složitosti (struktuře a počtu parametrů) entitního schématu. Na základě kombinatorických pravidel lze odvodit počet testovacích běhů na základě vztahu:

$$n_{TR} = 2^{n_M} \cdot 3^{n_{NM}}$$

kde: n_{TR} je počet testovacích běhů,

n_M značí počet povinných parametrů,

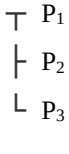
n_{NM} značí počet nepovinných parametrů (non-mandatory).

Příklad: Mějme entitní schéma o třech parametrech kde jeden z nich má příznak nepovinného parametru (označený symbolem *****). Označme stavy hodnot parametrů hodnotami: **V** – validní, **N** – nevalidní, **0** – není obsažen. Pak lze sestavit tabulka validity pro zadané struktury entitního schématu (Tabulka 5) a vypočítat množství testovacích běhů:

Tabulka 5: Závislost počtu generovaných běhů na složitosti entitního schématu

	Struktura	Variační tabulka generovaných stavů parametrů	Výpočet počtu běhů testu

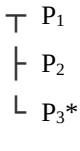
1)



P₁	V	V	V	V	N	N	N	N
P₂	V	V	N	N	V	N	N	V
P₃	V	N	V	N	V	V	N	V

$$n_{TR} = 2^3 \cdot 3^0 = 8$$

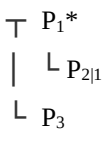
2)



P₁	V	V	V	V	V	V	N	N	N	N	N	N
P₂	V	V	V	N	V	V	N	N	N	V	V	V
P₃*	V	N	0	V	N	0	V	N	0	V	N	0

$$n_{TR} = 2^2 \cdot 3^1 = 12$$

3)



P₁*	V	V	V	V	N	N	N	N	0	0
P₂|₁	V	V	N	N	V	V	N	N	0	0
P₃	V	N	V	N	V	N	V	N	V	N

$$n_{TR} = 2^2 \cdot 3^1 - (2^1 \cdot 3^0)$$

$$n_{TR} = 10$$

V prvním příkladu je popsána struktura entitního schématu obsahující pouze jednu úroveň parametrů. Žádný z parametrů schématu není označen jako nepovinný. Z pohledu počtu nagenerovaných datových entit tento případ popisuje nejjednodušší možnou strukturu pro daný počet parametrů.

Druhý příklad již obsahuje nepovinný parametr, což navyšuje počet generovaných datových entit a tím i počet vytvořených testovacích běhů. Poslední ukázka popisuje princip výpočtu pro strukturu entitního schématu obsahující parametry ve více úrovních. Třetí příklad je také významný tím, že obsahuje podstrom s parametrem, který je závislý na vykreslení svého předchůdce, neboť ten má příznak nepovinného parametru. Tuto skutečnosti je v případě vyhodnocení potřeba zavést do celkového výpočtu testovacích běhů, a to odečtením počtu variací uvnitř podmíněně zobrazovaného podstromu.

10 DISKUZE – APLIKACE AUTOTEST

Aplikace AutoTest přistupuje k testování aplikačních rozhraní postavených na webových službách jiným způsobem, než je v rámci automatizovaných softwarových testů běžné. Neprověřuje samotnou funkčnost testovaného systému, nýbrž pouze ověřuje, zda chování aplikace, při rozsáhle variaci vstupních dat, odpovídá běžně očekávanému chování (tedy například končí úspěchem při validních vstupech a chybou při neplatném zadání). V určitých případech je schopna odhalit některé implementační chyby plynoucí z nepozornosti během vývoje nebo nepodchycení všech možných scénářů v souvislosti se vstupními daty.

10.1 Testované rozhraní

Pro demonstraci všech možností aplikace AutoTest je potřeba aby vybrané testovací rozhraní splňovalo následující podmínky:

- Práce se s testovaným systémem je přístupná prostřednictvím webových služeb nebo skrze aplikační rozhraní architektury SOAP.
- Rozhraní systému má pevně definovanou (a pokud možno i validovanou) podobu vstupních dat.
- V případě, že je odpověď systému chybový stav, bude v hlavičce pro hodnotu HTTP statusu, v souladu s protokolem, uvedena hodnota větší než 299.
- Rozhraní musí obsahovat nejméně jednu operaci, kterou je možné realizovat pomocí HTTP požadavku s metodou POST.

Ačkoliv mnoho zmíněných požadavků vyplývá z doporučení pro návrh webového aplikačního rozhraní založeného na architektuře REST (Fielding, 2000), nejedná se o pevný standard a řada vývojářů některá tato doporučení nerespektuje. Lze tak například narazit na webová rozhraní, kde standardní operace (CRUD) jsou volány vždy pomocí HTTP požadavku s metodou POST a jednotlivé operace se od sebe odlišují použitím jiné URI. Po rozsáhlé rešerši bylo testování veřejně dostupného rozhraní pro demonstrační účely odmítnuto a bylo nahrazeno tvorbou vlastního testovaného prostředí.

Za účelem demonstrace testovacích schopností aplikace AutoTest bylo vytvořeno samostatné webové rozhraní, které splňuje všechny výše uvedené podmínky. (kapitola 9.2.3).

10.2 Cíle

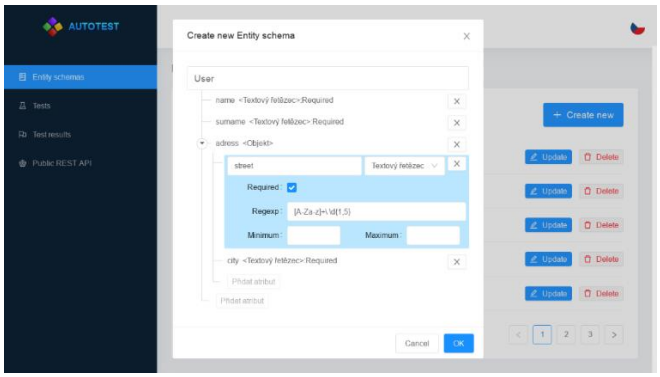
Cílem návrhu a vývoje aplikace AutoTest bylo vytvoření takového řešení, na kterém bude možné demonstrovat metodu testování koncových bodů aplikačního rozhraní softwaru pomocí pseudonáhodně generovaných variací datových vstupů.

Další schopností aplikace AutoTest je poskytovat na základě definovaných entitních schémat veřejné aplikační rozhraní architektury REST. Primárním účelem této funkce je poskytovat softwarovým testům předstíranou formu datového zdroje, nad kterým lze provádět běžné operace (CRUD). Výsledkem je funkční REST rozhraní aplikace představující neperzistentní datový zdroj s náhodně generovanými hodnotami odvozenými od parametrů definovaných pomocí entitních schémat.

Proto, aby bylo možné ověřit dosažení cílů, bylo přistoupeno k tvorbě vlastního testovaného rozhraní. To obsahuje právě jeden přístupový bod. Při implementaci testovaného rozhraní bylo třeba definovat entitní schéma ve vhodné podobě. Další etapou přípravy testovaného prostředí byla tvorba testu, která spočívala v propojení entitního schématu s koncovým bodem REST rozhraní.

10.3 Webové uživatelské rozhraní

Uživatelské prostředí je díky využití běžně známých vizuálních komponent intuitivní a uživatelsky přívětivé. Součástí návrhu uživatelského rozhraní je zohlednění responzivity vzhledem k různým zařízením. Aplikace není primárně určena pro použití na mobilních telefonech, proto trpí drobnými neduhy, které snižují uživatelský komfort (nutnost horizontálního posunu na opravdu malých displejích).



Obrázek 15: Realizace webového uživatelského rozhraní – tvorba entitního schématu

10.4 Výsledky

Pomocí aplikace bylo otestováno webového rozhraní popsané v kapitole 10.1. Záměrně vytvořená chyba byla testovacím nástrojem úspěšně odhalena a pomocí metody porovnání validity vstupních dat v požadavku oproti odpovědi také úspěšně zanesena jako podezřelý výsledek do výsledku testů. Během testu však došlo k vyššímu počtu testovacích běhů, než bylo původně zamýšleno. Na vině je pravděpodobně nepřesný způsob výpočtu, který nezohledňoval použití složitějších struktur zahrnujících Došlo tak k nárustu počtu testovacích běhu asi o 4%, což v případě testování softwaru není nijak .

10.5 Prostor pro rozšíření aplikace

Z určitého pohledu, nepříjemnou vlastností aplikace AutoTest, že neumožňuje tvorbu uživatelských testovacích scénářů řízených statickými hodnotami. Data pro vstup do koncových bodů testovaného systému jsou vždy generována náhodně. Tímto způsobem tedy není možné provádět testování, založené na testovacích případech, známé z jiných testovacích nástrojů (například SoapUI – kapitola 6.1). Aplikace také neumožňuje udržovat vnitřní stav mezi jednotlivými testovacími běhy.

Aplikace AutoTest testuje software na základě entitních schémat, která jsou do systému zadána uživatelem. Prvotní tvorba entitních schémat a testů vyžaduje intenzivní interakci s uživatelem, který je nucen vkládat data manuálně. Systémy poskytující provádění operací prostřednictvím webových služeb mohou k popisu vstupních zpráv využívat tzv. popisné soubory (typicky WSDL pro SOAP a WADL pro REST API). Popisný soubor zahrnuje informace o operacích a struktuře vstupních dat, které lze prostřednictvím testované webové služby využívat. Vhodným rozšířením testovací aplikace by bylo umožnění importu souborů obsahujících popis webové služby nebo rozhraní a jeho automatická transformace do podoby entitního schématu nebo JSON Schema. Tímto způsobem by mohlo být dosaženo vyšší míry automatizace.

V současné době aplikace AutoTest neobsahuje žádná definovaná omezení ohledně maximální velikosti entitních schémat nebo počtu provedených požadavků na server. Počet testovacích běhů roste exponenciálně s počtem parametrů uvnitř entitního schématu. V případě složitějších (složitost – viz kapitola 9.4) entitních schémat, která budou obsahovat přes 20 parametrů dosahuje počet testovacích běhů hodnoty přesahující jeden milion.

ZÁVĚR

V rámci diplomové práce byla popsána obecná problematika testování softwarových produktů se zaměřením na systémy využívající komunikaci pomocí webových služeb SOAP nebo aplikační rozhraní založené na architektuře REST. Dále byla provedena rešerše předních nástrojů využívající automatizaci testů pro podporu vývoje softwarových systémů, vytváření testovacích scénářů nebo samotnou realizaci testovacích běhů. Pozornost byla věnována přípravě vstupních dat v rámci softwarových testů a popisu struktury zápisu programových testů včetně znázornění jejich životního cyklu.

Během zpracování diplomové práce byly využity poznatky ohledně vytváření modelů, artefaktů a popisu návrhu vlastní aplikace ze studia předmětu Projektování SW systémů. Pro popis praktické části práce bylo využito základů kombinatorického počtu a podmíněné pravděpodobnosti, při vyčíslení hraničního počtu testovacích běhů v závislosti na složitosti entitního schématu, z předmětu Teorie pravděpodobnosti a matematické statistika.

Předmětem praktické části práce bylo vytvoření vlastního prototypu aplikačního řešení pro testování přístupových bodů webových aplikačních rozhraní. Myšlenka pro přístup k testování, pomocí vyvinuté aplikace AutoTest, vychází z popisu metody Concolic testing (kapitola 5.1.1), která je převedena do prostředí systémů využívajících webové služby. Pokrytí co možná největší části zdrojového kódu (kapitola 5.1.2) pomocí testovacích scénářů je v případě vyvinuté aplikace dosaženo na základě generování sady variací vstupních dat odvozených na základě popisu vstupu do konkrétního koncového bodu rozhraní systému. Tímto způsobem je původní přístup white box testování převeden do prostředí black box testování, který je mnohem přirozenější pro testování aplikačních rozhraní a webových služeb.

Pro demonstraci vytvořené aplikace byla zároveň vyvinuta jednoduchá aplikace reprezentující testovaný systém, jenž v sobě obsahuje implementační chybu (kapitola 9.2.3). Tato chyba, která představuje běžný problém při využívání nepovinného vstupu do systému bez ověření jeho přítomnosti, byla aplikací úspěšně ověřena. Žádný test nebyl falešně negativní, ani falešně pozitivní.

Provedením rešerší, provedením analýzy, návrhu aplikace, její implementací a demonstrací jejího fungování došlo ke splnění zadání.

POUŽITÁ LITERATURA

[1] BEECHAM, Sarah, Nathan BADDOO, Tracy HALL, Hugh ROBINSON a Helen SHARP. Motivation in Software Engineering: A systematic literature review. *Information and Software Technology* [online]. 2008, **50**(9-10), 860-878 [cit. 2019-08-08]. DOI: 10.1016/j.infsof.2007.09.004. ISSN 09505849. Dostupné z: https://linkinghub.elsevier.com/retrieve/pii/S0950584907001097

[2] BRUNA, S. Learning GraphQL and Relay. *Packt Publishing*, 2016 ISBN 978-1-78646-575-7.

[3] CRUZES, Daniela S., Nils B. MOE a Tore DYBA. Communication between Developers and Testers in Distributed Continuous Agile Testing. In*: 2016 IEEE 11th International Conference on Global Software Engineering (ICGSE)* [online]. IEEE, 2016, 2016, s. 59-68 [cit. 2019-08-07]. DOI: 10.1109/ICGSE.2016.27. ISBN 978-1-5090-2680-7. Dostupné z: http://ieeexplore.ieee.org/document/7577420/

[4] CURBERA, F., M. DUFTLER, R. KHALAF, W. NAGY, N. MUKHI a S. WEERAWARANA. Unraveling the Web services web: an introduction to SOAP, WSDL, and UDDI. *IEEE Internet Computing* [online]. 2002, **6**(2), 86-93 [cit. 2019-08-04]. DOI: 10.1109/4236.991449. ISSN 1089-7801. Dostupné z: http://ieeexplore.ieee.org/document/991449/

[5] DESIKAN, S. a RAMESH, G. Software Testing: Principles and Practice. Pearson Education Canada, 2006 ISBN 9788177581218.

[6] DUVALL, Paul M., Steve MATYAS a Andrew GLOVER. *Continuous integration: improving software quality and reducing risk.* Upper Saddle River, NJ: Addison-Wesley, c2007. ISBN 978-0-321-33638-5.

[7] FIELDING, Roy T. *Architectural Styles and the Design of Network-based Software Architectures* [online]. Irvine, 2000 [cit. 2019-07-22]. Dostupné z: https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf. Disertace. University of California Irvine.

[8] FILIPOVIC, Milorad, Goran SLADIC, Branko MILOSAVLJEVIC a Gordana MILOSAVLJEVIC. Applying SMT algorithms to code analysis. In: *Proccedings of the ICAIIT2016* [online]. University "St. Kliment Ohridski" Bitola, Macedonia, 2016, 2016, s. 163-170 [cit. 2019-08-12]. DOI: 10.20544/AIIT2016.20. ISBN 9789989870750. Dostupné z: http://eprints.fikt.edu.mk/129

[9] GAROUSI, Vahid a Frank ELBERZHAGER. Test Automation: Not Just for Test Execution. *IEEE Software* [online]. 2017, **34**(2), 90-96 [cit. 2019-08-05]. DOI: 10.1109/MS.2017.34. ISSN 0740-7459. Dostupné z: http://ieeexplore.ieee.org/document/7888399/

[10] GODEFROID, Patrice, Nils KLARLUND a Koushik SEN. DART. In: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation - PLDI '05 [online]. New York, USA: ACM Press, 2005, s. 213-223 [cit. 2019-07-30]. DOI: 10.1145/1065010.1065036. ISBN 1595930566. Dostupné z: http://portal.acm.org/citation.cfm?doid=1065010.10650361065036

[11] HUMBLE, Jez a David FARLEY. *Continuous delivery: reliable software releases through build, test, and deployment automation.* Upper Saddle River, NJ: Addison-Wesley, 2010. ISBN 978–0–321–60191–9.

[12] IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990*. 1990, s. 1–84. ISBN 978-0-7381-0391-4.

[13] KARHU, Katja, Tiina REPO, Ossi TAIPALE a Kari SMOLANDER. Empirical Observations on Software Testing Automation. In: *2009 International Conference on Software Testing Verification and Validation* [online]. IEEE, 2009, 2009, s. 201-209 [cit. 2019-08-08]. DOI: 10.1109/ICST.2009.16. ISBN 978-1-4244-3775-7. Dostupné z: http://ieeexplore.ieee.org/document/4815352/

[14] LEWIS, Robert O. *Independent verification and validation: a life cycle engineering process for quality software*. New York: Wiley, c1992. ISBN 978-0-471-57011-0. CRAIG, Rick D. a Stefan P. JASKIEL. *Systematic software testing*. Boston: Artech House, 2002. ISBN 1-58053-508-9.

[15] LUO, Qingzhou, Farah HARIRI, Lamyaa ELOUSSI a Darko MARINOV. An empirical analysis of flaky tests. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014* [online]. New York, New York, USA: ACM Press, 2014, 2014, s. 643-653 [cit. 2019-08-06]. DOI: 10.1145/2635868.2635920. ISBN 9781450330565. Dostupné z: http://dl.acm.org/citation.cfm?doid=2635868.2635920

[16] MANDEL, Lawrence. *IBM* [online]. IBM, 2008 [cit. 2019-08-04]. Dostupné z: https://www.ibm.com/developerworks/webservices/library/ws-restwsdl/

[17] NAIK, Kshirasagar a Priyadarshi. *Software testing and quality assurance: theory and practice*. Hoboken, N.J.: John Wiley, c2008. ISBN 978-0-471-78911-6.

[18] O'BROIN, Caoimh. *The Impact of Test Automation on Software Testers* [online]. Dublin, 2015 [cit. 2019-07-20]. Dostupné z: https://scss.tcd.ie/publications/theses/diss/2015/TCD-SCSS-DISSERTATION-2015-012.pdf. Disertace. Trinity College Dublin.

[19] PEŠKA, Martin. *Responzivní uživatelské rozhraní v kontextu moderního webového designu* [online]. Brno, 2016 [cit. 2019-08-19]. Dostupné z: https://is.muni.cz/th/txsqi/Diplomova_prace.pdf. Diplomová práce. Masarykova univerzita. Vedoucí práce RNDr. Barbora Kozlíková, Ph.D.

[20] RANA, Kuldeep. ArtOfTesting. *Types of Testing* [online]. 2019 [cit. 2019-08-02]. Dostupné z: https://artoftesting.com/manualTesting/types-of-testing.html

[21] SCHNEIDEWIND, N.F. The State of Software Maintenance. *IEEE Transactions on Software Engineering* [online]. 1987, **SE-13**(3), 303-310 [cit. 2019-08-08]. DOI: 10.1109/TSE.1987.233161. ISSN 0098-5589. Dostupné z: http://ieeexplore.ieee.org/document/1702216/

PŘÍLOHY

Příloha A – Zprovoznění Praktické práce 71
Příloha B – Selenium IDE export testu (Mocha) 72
Příloha C – Zdroj. kód testované aplikace (REST) 73

PŘÍLOHA A – ZPROVOZNĚNÍ PRAKTICKÉ PRÁCE

Následující text popisuje postup, jakým lze spustit výsledné řešení:

1. **Příprava:**
    1. Nejprve překopírujte všechny soubory z adresáře „zdrojove_kody“, který naleznete na přiloženém datovém médiu k této diplomové práci.
    2. Stáhněte a nainstalujte Node.js[22] (automaticky měl nainstalovat i správně balíčků a knihoven pro Node.js - NPM).
        - Ověřte, že máte vše správně nainstalováno pomocí příkazové řádky OS:

```
node -v
npm -v
```

        Výstupem by měli být informace o nainstalované verzi softwaru
    3. Stáhněte a nainstalujte Docker[23].
2. **Server, Client, Tested-server, Software-test:**
    1. Postupně procházejte jednotlivé adresáře a nainstalujte všechny potřebné balíčky pro danou aplikaci pomocí příkazu:

```
npm install
```

3. **Spuštění prostředí**
    1. Vraťte se zpět do adresáře „zdrojove_kody“
    2. Spusťte všechna prostředí pomocí příkazu:

```
docker-compose up
```

Aplikace by nyní měli běžet na následujících portech:

Client: 8080 Server: 3000 Tested-server:4000

[22] Oficiální webové stránky Node.js s možností stažení - https://nodejs.org/en/
[23] Oficiální webové stránky projektu Docker - https://www.docker.com/

PŘÍLOHA B – SELENIUM IDE EXPORT TESTU (MOCHA)

Scénář vytvořený pomocí nástroje Selenium IDE a následně vyexportovaný do zápisu testovacího frameworku Mocha (programovací jazyk - Node.js):

```
// Generated by Selenium IDE
const { Builder, By, Key, until } = require('selenium-webdriver')
const assert = require('assert')

describe('Look for "Informační technologie"', function() {
  this.timeout(30000)
  let driver
  let vars
  beforeEach(async function() {
    // Web browser driver
    driver = await new Builder().forBrowser('chrome').build()
    vars = {}
  })

  afterEach(async function() {
    await driver.quit()
  })

  it('Informační technologie', async function() {
    await driver.get("https://www.upce.cz/")

    // Search button
    await driver.findElement(By.id("search-open")).click()

    // Wait 1s for search form to load
    await driver.sleep(1000)

    // Write search phrase
    await driver.findElement(By.id("search-field"))
     .sendKeys("Informační technologie")

    // Submit search form
    await driver.findElement(By.id("obsah-radio")).click()

    // Wait 1s for new page to load
    await driver.sleep(1000)

    // Select search result
    await driver.findElement(
      By.linkText("Informační technologie")
    ).click()

    // Check heading text
    assert(
      await driver.findElement(
        By.css(".node_banner_title")
      ).getText() == "Informační technologie"
    )
  })
})
```

PŘÍLOHA C – ZDROJ. KÓD TESTOVANÉ APLIKACE (REST)

```
const express = require('express');
const bodyParser = require('body-parser');

const { Validator, ValidationError } = require('express-json-validator-
middleware');

const validator = new Validator({allErrors: true});
const validate = validator.validate;

// Define a validation JSON Schema
const testSchema = {
  type: 'object',
  required: ['name', 'age'],
  properties: {
    name: {
      type: 'string',
      minLength: 1,
      maxLength: 20
    },
    cardId: {
      type: 'string'
    },
    age: {
      type: 'integer'
    },
    active: {
      type: "boolean"
    }
  }
};

const app = express();

app.use(bodyParser.json());

// This route validates req.body against the testSchema
app.post('/test-endpoint/', validate({body: testSchema}), function(req,
res) {
  const result = {
    name: req.body.name,
    // business mistake
    // Unhandled undefined value for cardId
    cardId: req.body.cardId.toUpperCase(), // wrong implementation

    // cardId: req.body.cardId ? req.body.cardId.toUpperCase() : null,
    age: req.body.cardId,
    active: req.body.active || false
  };
  res.send(result);
});
```