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

Bakalářská práce

Komplexní testy webové aplikace

Plzeň 2022

David Kůta

ZÁPADOČESKÁ UNIVERZITA V PLZNI
Fakulta aplikovaných věd
Akademický rok: 2021/2022

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**David KŮTA**
Osobní číslo:	**A19B0118P**
Studijní program:	**B0613A140015 Informatika a výpočetní technika**
Specializace:	**Informatika**
Téma práce:	**Komplexní testy webové aplikace**
Zadávající katedra:	**Katedra informatiky a výpočetní techniky**

Zásady pro vypracování

1. Seznamte se s možnostmi, které poskytují Php frameworky pro zvyšování kvality vytvářených aplikací, např. logování, jednotkové testy, integrační testy apod. Prověřte též možnosti CI pro jeden zvolený framework.
2. Navrhněte testovací plán pro webovou aplikaci „Softwarová podpora organizace předmětů TSP“ s využitím případů užití a seznamu požadavků. Vyberte vhodné nástroje pro automatizované testování.
3. Na základě testovacího plánu realizujte všechny typy navržených testů, přičemž klaďte důraz na automatizaci testů a na optimální pokrytí, zejména pomocí jednotkových, funkcionálních a end-to-end testů.
4. Během vývoje testované aplikace průběžně ověřujte její kvalitu a o této činnosti veďte prokazatelné záznamy pomocí vhodného nástroje.

Rozsah bakalářské práce: **doporuč. 30 s. původního textu**
Rozsah grafických prací: **dle potřeby**
Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam doporučené literatury:

Dodá vedoucí bakalářské práce.

Vedoucí bakalářské práce: **Doc. Ing. Pavel Herout, Ph.D.**
Katedra informatiky a výpočetní techniky

Datum zadání bakalářské práce: **4. října 2021**
Termín odevzdání bakalářské práce: **5. května 2022**

L.S.

Doc. Ing. Miloš Železný, Ph.D.
děkan

Doc. Ing. Přemysl Brada, MSc., Ph.D.
vedoucí katedry

V Plzni dne 14. října 2021

Prohlášení

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

V Plzni dne 5. května 2022

David Kůta

Poděkování

Tímto bych rád poděkoval doc. Ing. Pavlu Heroutovi, Ph.D. za odborné vedení a cenné rady v průběhu zpracování této bakalářské práce

Abstract

This bachelor thesis entitled Comprehensive tests of a web application deals with the issue of testing and improving the quality of web applications. The result of this work is a comprehensive set of Selenium tests that optimally covers the application under test in terms of requirements and use cases. At the same time, unit testing in PHP is included in the testing process. SquashTM software was chosen to document the testing process and a test plan was created in which the testing strategy is properly described. In addition to the testing process, the capabilities of PHP frameworks to enhance the reliability of web applications such as logging, unit testing and a comparison of these capabilities for the Nette and Symfony frameworks are described.

Abstrakt

Bakalářská práce se zabývá problematikou testování a zvyšování kvality webových aplikací. Výsledkem této práce je rozsáhlá sada Selenium testů, která optimálně pokrývá testovanou aplikaci z hlediska požadavků a případů užití. Do testovacího procesu je zahrnuto i jednotkové testování v PHP. Pro dokumentaci procesu testování byl zvolen software SquashTM a vytvořen testovací plán, ve kterém je testovací strategie náležitě popsána. Kromě testovacího procesu jsou popsány možnosti PHP frameworků pro zvýšení spolehlivosti webových aplikací, jako například logování, jednotkové testy a porovnání těchto možností u frameworků Nette a Symfony.

Obsah

1 Úvod 9

2 Možnosti PHP frameworků pro zvyšování spolehlivosti vytvářených aplikací 10
2.1 Logování 10
2.1.1 Logování v Nette framework 10
2.1.2 Logování v Symfony 11
2.1.3 Logovací knihovna Monolog 12
2.2 Jednotkové testování 15
2.2.1 Testovací framework PHPUnit 16
2.2.2 Testovací framework Nette Tester 19
2.3 Kontinuální integrace a kontinuální nasazení 21
2.3.1 Kontinuální integrace 21
2.3.2 Kontinuální nasazení 22
2.3.3 Možnosti kontinuální integrace pro Nette Tester . . . 22

3 Metody a přístupy k testování webových aplikací 24
3.1 Typy testů z hlediska viditelnosti zdrojového kódu 24
3.1.1 Testování černé skříňky 24
3.1.2 Testování bílé skříňky 24
3.1.3 Testování šedé skříňky 25
3.2 Úrovně testování 25
3.2.1 Jednotkové testování 25
3.2.2 Integrační testování 25
3.2.3 Systémové testování 25
3.2.4 Akceptační testování 26
3.3 Druhy testů 26
3.3.1 Funkcionální testy 26
3.3.2 Nástroje pro funkcionální testování 26
3.3.3 End-to-End testy 27

4 Dokumentace k testování 29
4.1 Testovací plán 29
4.1.1 Struktura hlavního plánu testování dle metodiky TMap Next 29

4.1.2 Struktura detailního plánu pro jednotlivé úrovně testování dle metodiky TMap Next 30
4.1.3 Zásady při vytváření testovacího plánu 31

5 Analýza testované aplikace 32
5.1 Účel webové aplikace Podpůrný software TSP 32
5.1.1 Případy užití a požadavky na PSTSP 32
5.1.2 Výběr vhodných testovacích nástrojů 32
5.2 Testovací plán pro PSTSP 33
5.2.1 Úvod 33
5.2.2 Formulace cílů a rozsahu testovacího projektu 33
5.2.3 Použitá dokumentace 34
5.2.4 Strategie testování 34
5.2.5 Přístup k testování 35
5.2.6 Infrastruktura pro testování 37
5.2.7 Kritérium pro uvolnění systému do produkce 37
5.2.8 Řízení testovacího projektu 37
5.2.9 Plán vytváření testů 37

6 Realizace navržených testů 39
6.1 Jednotkové testy 39
6.1.1 Struktura jednotkových testů v projektu 39
6.1.2 Ukázka jednotkového testu 39
6.1.3 Výsledky testování 40
6.2 Struktura požadavků a testovacích případů v SquashTM . . 40
6.2.1 Požadavky na software 40
6.2.2 Testovací případy 41
6.3 Návrhový vzor PageObject 41
6.3.1 Příklad využití 41
6.3.2 Výhody při použití návrhového vzoru PageObject . . 43
6.4 Funkcionální testy 43
6.4.1 Struktura testovacího projektu 43
6.4.2 Výsledky testování 46
6.5 End-to-end testy 47
6.5.1 Příklad end-to-end testu 47
6.5.2 Výsledky testování 47

7 Začlenění testovacího procesu do plánu vývoje aplikace 48
7.1 Výsledky průběžného testování 48
7.1.1 Testování ve vývojovém cyklu 49

7.1.2 Testování v finální fázi vývojového cyklu 49

8 Závěr **50**

Literatura **52**

Seznam zkratek **54**

A Spouštění testů **56**
A.1 Spouštění jednotkových testů 56
A.2 Spouštění funkcionálních a E2E testů 56
A.3 Spouštění testů z příkazové řádky 56

B Statistiky k vytvořeným testům **59**
B.1 Požadavky 59
B.1.1 Provázání RQM a TC 59
B.1.2 Priorita RQM 60
B.1.3 Pokrytí priorit RQM pomocí TC 60
B.1.4 Globální úspěšnost TC dle priorit RQM 61
B.2 Testovací případy 61
B.2.1 Provázání TC a RQM 61
B.2.2 Priorita testovacích případů 62
B.2.3 Počet testovacích případů na jednotlivé priority požadavků 63
B.3 Testovací iterace 64
B.3.1 Testy během vývoje 64
B.3.2 Testy poslední verze 64
B.3.3 Počet ověřovaných RQM během testů poslední verze aplikace 65

C Obsah elektronické přílohy **66**
C.1 Adresářová struktura 66

1 Úvod

Na Katedře informatiky a výpočetní techniky vznikají zcela nové předměty Týmový softwarový projekt 1 a 2 (KIV/TSP1 a KIV/TSP2). Jedná se o týmovou práce překračující hranice ročníku, a proto je vhodné mít k dispozici softwarovou podporu (PSTSP) pro řízení celého průběhu těchto dvou, na sebe navazujících, předmětů. Souběžně s touto bakalářskou prací je aplikace vyvíjena.

Dle specifikace se jedná o robustní webovou aplikaci, která bude pravděpodobně v následujících letech rozšiřována, je nezbytně nutné vytvořit sady testů, které budou ověřovat funkcionalitu aplikace a tím prokazatelně zvyšovat její kvalitu i spolehlivost.

Jelikož v žádné činnosti nelze předpokládat, že výsledný produkt bude bezchybný, je vhodné tyto chyby co nejdříve zaznamenat a opravit. Čím déle bude chyba ve výsledném produktu přítomna, tím náročnější a nákladnější může být její odstranění. Proto je vhodné do plánu vývoje software zařadit testovací proces. Na testování software je kladen v průběhu let stále větší důraz, proto je testovací proces náležitě dokumentován pomocí vytvářeného testovacího plánu, který je rovněž předmětem této práce.

Součástí této bakalářské práce je průzkum možností PHP frameworků, které zvyšují spolehlivost vytvářených webových aplikací, jelikož tyto informace budou poskytnuty vývojáři aplikace PSTSP, který je do vývoje s rozvahou začlení.

Cílem je vytvoření rozsáhlé sady automatizovaných testů, které optimálně pokrývají aplikaci z hlediska požadavků, které jsou extrahovány z případů užití testované aplikace. Jelikož budou testy vytvářeny souběžně s aplikací, budou testy systematicky spouštěny a výsledky budou uchovávané pomocí vhodného nástroje pro organizaci testování.

2 Možnosti PHP frameworků pro zvyšování spolehlivosti vytvářených aplikací

2.1 Logování

Tento princip slouží k ladění aplikace a sledování její činnosti z dlouhodobého hlediska pomocí tzv. logovacích záznamů. Jako typicky sledované aktivity se považují například: běžná činnost programu, výskyt chyb a výjimek, konfigurace programu a její změny a cokoli dalšího, co by mohlo být užitečné pro zpětnou analýzu chování aplikace [5]. Tyto aktivity bývají sdruženy úrovní, která slouží k zařazení logovaných událostí podle jejich závažnosti.

2.1.1 Logování v Nette framework

Ve výchozí variantě, tj. Nette framework bez jakéhokoli dalšího rozšíření, lze použít nástroj Tracy. Tento nástroj je konfigurovatelný pomocí konfiguračních souborů ve formátu `NEON`, který je použit pro veškerou konfiguraci Nette aplikací. Dále je možné využít libovolnou knihovnu třetí strany, jako například KLogger, Apache Log4php a Monolog, které disponují širší možností konfigurace, což může znamenat lepší přehlednost ve výsledných logách.

Tracy

Tracy slouží zejména k ladění Nette aplikací ve vývojovém a v produkčním režimu. Pokud je spuštěna aplikace s chybou ve vývojovém režimu, tak Tracy vizualizuje lokaci a druh chyby. V produkčním režimu tyto informace loguje, takže je přesně známo, která funkce způsobila pád aplikace díky zobrazené posloupnosti volaných funkcí. Logování je zajištěno pomocí rozhraní `ILogger` a základní logger, který toto rozhraní implementuje, produkuje záznamy ve tvaru `[zpráva, úroveň logování]`. Podporované úrovně logování [16] jsou následující a můžeme je využít například takto:

- Debug – úroveň sloužící pro ladění aplikace
- Info – informační výpisy
- Warning – varovná hlášení, potencionálně nebezpečné stavy

- Error – chyby aplikace za běhu
- Exception – vyhození neočekávané výjimky
- Critical – aplikace je nepoužitelná

Pomocí tohoto nástroje lze dosáhnout pouze jednoduchého logování, které však může být pro nějaký menší projekt zcela dostačující. Pokud je ale cíleno na přehlednost a logickou strukturu logovacích zpráv, tak zde tento princip selhává. Není totiž možné bez jakékoli úpravy zaznamenávat se zprávami i jejich kontext.

Konfigurace Tracy

Jak již bylo řečeno, konfigurace nástroje Tracy probíhá pomocí konfiguračních souborů ve formátu `NEON`, kde lze nastavit například cíl logování (což bude vždy složka, do které se budou ukládat soubory podle již navrženého systému Nette aplikace), dále např. email pro notifikaci vyhození výjimky, či pádu aplikace, a spoustu dalších parametrů, o kterých se lze dočíst v dokumentaci [15].

2.1.2 Logování v Symfony

V Symfony je tento princip definován pomocí tzv. `LoggerInterface`, které obsahuje všechny metody, které se pro tuto funkcionalitu využívají. Framework je ve zkratce vybaven metodami pro záznam logů všech požadovaných úrovní a obecné metody `log`, do které se vloží z číselníku úrovní závažnosti ta požadovaná. Výhodou tohoto rozhraní je například to, že už předpokládá záznam aktivit v nějakém kontextu, oproti výše zmiňované Tracy.

PSR-3 Logger Interface

Symfony framework definuje minimalistický standard `PSR-3`, který umožňuje využívat logování bez závislosti na konkrétní implementaci logovacího frameworku. Tento princip vede k dodržení SOLID principů a k naprogramování znovu využitelného kódu. PSR-3 popisuje rozhraní, tzv. `Logger Interface`, které obsahuje logování v osmi úrovních závažnosti [6], založených na RFC 5424[1].

[1] https://datatracker.ietf.org/doc/html/rfc5424

Jsou to úrovně typu:

- Debug – úroveň sloužící pro ladění aplikace
- Info – informační výpisy
- Notice – varovná hlášení, potencionálně nebezpečné stavy
- Warning – neobvyklé nebo nepředpokládané stavy, které ale nejsou chyby
- Error – chyby, které nevyžadují okamžité vyřešení, ale měly by být sledované
- Critical – kritické stavy aplikace
- Alert – aplikace je ve stavu, který vyžaduje okamžitý zásah
- Emergency – aplikace je nepoužitelná

Tímto standardem se řídí například logovací knihovna Monolog, kterou Symfony obsahuje již ve výchozí variantě. Tato knihovna je dále popsána detailněji.

Konfigurace logování v Symfony

Symfony obsahuje pro konfiguraci již implementovanou logiku. Stačí tedy do adresáře, kam se umisťují konfigurační soubory, vložit konfigurační soubor, který představuje konfiguraci právě používaného logovacího frameworku. Můžou být použity soubory ve formátu YAML, XML či dokonce může být konfigurací samostatný PHP skript.

2.1.3 Logovací knihovna Monolog

Tato knihovna implementující PSR-3 Logger Interface je integrována ve výchozí verzi do frameworků Symfony, Laravel a mikroframeworku Lumen. Dále je integrován například do frameworku Nette, pomocí dodatečného rozšíření [2]. Konfigurace v jednotlivých integracích knihovny Monolog je totožná významově, avšak formou nikoli. Je vždy zvykem využít konfigurační službu daného frameworku, takže například pro Nette je možné psát konfiguraci ve formátu `NEON` [7].

Díky této knihovně je možné si nadefinovat více loggerů, kde každá instance loggeru má svůj handler (může mít i více handlerů), který určuje, co

[2] contributte/monolog

se stane se zaznamenávanou informací. Dále může mít každý logger i svůj formátovač určující výstupní formu zaznamenávaných informací. Možnost definice více loggerů, kde každý logger může mít více handlerů, umožňuje vytvářet sofistikovanou logiku nad principem logování. S každým záznamem může být nakládáno podle toho, jaký logger tento záznam vyprodukoval, a následně dle úrovně závažnosti.

Pokud je nastaveno více handlerů, zaznamenávaná informace jimi „probublává“, pokud je toto „probublávání“ povoleno, a je náležitě zpracovávána. Kritérium rozhodující zda se zpracovávaný log zpracuje daným handlerem, či nikoli, je úroveň závažnosti logované informace. Pokud handleru nastavíme filtr na úroveň závažnosti, tak s nastavenou úrovní se současně povolují všechny vyšší a zakazují všechny nižší úrovně závažnosti logované informace. Loggery jsou identifikovány pomocí tzv. „channelu“, což je název loggeru, který je volen dle kontextu využití daného loggeru [7]. Logovaná zpráva obsahuje nejen zprávu, kterou do ní programátor zapíše při zadání příkazu `log`, ale nese i množství dalších informací, jako například kontext zprávy, který se negeneruje automaticky, avšak dodává více významnosti dané zprávy.

Informace obsazené v logované zprávě

- level – úroveň závažnosti, definovaná knihovní konstanta (číslo)
- level_name – název úrovně závažnosti
- datetime – časová známka, kdy byl záznam vytvořen
- channel – jméno loggeru
- extra – pole, kam registrované procesory vkládají dodatečná data
- context – kontext logované informace, nějaké informace navíc. Například id přihlášeného uživatele, který danou akci provádí.
- message – popisek logované zprávy

Základní možnosti knihovny monolog

```
1 use Monolog\Logger;
2 use Monolog\Handler\BrowserConsoleHandler;
3 $logger = new Logger('nazev_loggeru');
4 $streamHandler = new StreamHandler("file.log", Logger::ERROR);
```

```
5 $lineFormat = "%datetime% %channel%.%level_name% %message% %
     extra% \n";
6 $dateFormat = "Y-m-d H:i:s";
7 $lineFormatter = new LineFormatter($lineFormat, $dateFormat);
8 $streamHandler->setFormatter($lineFormatter);
9 $logger->pushProcessor(new WebProcessor());
10 $logger->pushHandler($streamHandler);
11
12 $logger->info('Uzivatel se prihlasil', ['id' => 123]);
```

Ukázka kódu 1: Demonstrace vytvoření loggeru

Na ukázce kódu 1, je vidět základní možnost konfigurace loggeru prostřednictvím jazyka PHP. Další možnosti konfigurace pro PHP frameworky jsou významově identické, pouze je pro konfiguraci použit vždy formát souboru, se kterým daný framework pracuje. Logika konfigurace zůstává stejná. Nejprve je zde vytvořena instance třídy `Logger`, která představuje samotný logger, který je dále konfigurován. Následně je definován tzv `handler`, který určuje, co se stane s logovacími zprávami, které jsou vytvořeny pomocí výše vytvořené instance loggeru. V příkladu je použit `StreamHandler`, který proud zaznamenávaných informací posílá do souboru určeného 1. parametrem, avšak záznamy musí být alespoň úrovně `ERROR`.

Následně je zde definován formát řádky a data, protože je v zápětí použit formátovač pro řádku, tzv. `LineFormatter`. Tento formatter je dále nastaven ve výše definovaném `StreamHandleru`, což zajistí, aby záznamy, které budou pomocí loggeru vyprodukovány, měly požadovaný formát. Nakonec je pro demonstraci možností loggeru přiřazen `WebProcessor`, který dodává dodatečné informace do pole `extra`. Momentálně se jedná o URI, request method a client IP. Poslední řádka demonstruje použití loggeru s vyplněným kontextem.

Dále budou zmíněny formátovače, handlery a procesory, které jsou k dispozici.

Možnosti handlerů

Dle dokumentace Monologu je k dispozici několik handlerů. Každý handler musí implementovat rozhraní `HandlerInterface`, které je dostupné v knihovně Monolog.

Handlery jsou řazeny do skupin pro lepší přehlednost [7]. Skupiny handlerů jsou následující:

- Log to files and syslog – pro logování do souborů, či do syslogu

- Send alerts and emails – pro odesílání emailů, či notifikací, s logovanou informací, na nějakou platformu (email, slack, ...)
- Log specific servers and networked logging – využití logovacích služeb třetích stran
- Logging in development – pro logovací výpisy ve vývoji
- Log to databases – pro ukládání logů do databáze
- Wrappers / Special Handlers – handlery se speciální funkcionalitou, které obalují výše popsané handlery

Možnosti formátovačů

Aktuálně je k dispozici celkem 16 formátovačů. Všechny dodávají nějaký formát, či vzhled výstupním záznamům (logům). Jejich použití je vhodné, pokud chceme například logy dále strojově zpracovávat, což obvykle bývá žádoucí.

Například `HtmlFormatter` formátuje data do čitelné podoby ve formátu HTML tabulky. Dále výše použitý `LineFormatter` formátuje řádek tak, aby byl zapsán v definovaném formátu. A nakonec `JsonFormatter` převádí záznamy do JSON [7].

Možnosti procesorů

Momentálně je k dispozici v knihovně Monolog předdefinováno celkem 11 procesorů. Jak již bylo zmíněno, procesory vkládají dodatečné informace do pole `extra` v logované zprávě. Od položky `context` se liší tím, že je automaticky generovaná právě procesorem. Procesor může být přidán jak loggeru, tak handleru stejným způsobem, který byl popsán výše na ukázce kódu 1. Pro definici vlastního procesoru můžeme využít PHP vlastnosti `callable`, a pouze předat `callable` té instanci, ke které se přiřazuje procesor.

Například `IntrospectionProcessor` naplní pole `extra` informací, odkud byl logovací záznam vytvořen. `MemoryUsageProcessor`, přidá využitou paměť a `ProcessIdProcessor` přidá informaci o ID procesu. Další popis procesorů je dostupný v dokumentaci monologu [7].

2.2 Jednotkové testování

Tento termín označuje testování na velmi nízké úrovni. Z pohledu architektury dané aplikace se testují malé jednotky kódu. Nejčastěji to jsou entitní

třídy a metody v nich. Jedná se o postup, kdy jsou testy psány souběžně s vytvářením kódu, tudíž jsou psány vývojářem. Tímto se značně zvýší kvalita vytvářeného kódu, jelikož je většina defektů odstraněna již při vývoji. Aby byl kód jednoduše testovatelný, je vývojář nucen psát krátké metody, které mají pouze jednu činnost. V podstatě je nucen psát kód, který je z lidského pohledu čitelnější [9]. Problematika psaní čitelného a dobře navrženého kódu kódu, např. z hlediska znovupoužitelnosti je popsána například v `SOLID` pravidlech, což je sada doporučení, principů a vodítek, sloužících k vytvoření kvalitnějšího objektového návrhu [1].

Knihovna pro jednotkové testování je zpravidla pojmenována `xUnit` (pro PHP existuje knihovna PHPUnit, pro Javu existuje JUnit, atd). Podpora pro tyto knihovny bývá většinou integrována do mnoha vývojových prostředí, což umožňuje rychlé spouštění testů, či generování kostry testů ze struktury libovolné třídy. Nejprve zde bude popsán testovací framework PHPUnit, který je zabalen do balíku knihoven, které jsou integrovány do PHP frameworku Symfony. Nakonec zde bude popsán framework Nette Tester, který je integrován do Nette. Ačkoli je každá zde zmíněná testovací knihovna vždy více svázána s určitým aplikačním frameworkem, není problém použít knihovnu v jiném aplikačním frameworku, protože instalace probíhá vždy přes Composer.

2.2.1 Testovací framework PHPUnit

Knihovna může být do aplikace distribuována jako Composer balíček. Testy se standardně umisťují do adresáře `tests`, který se nachází v kořenovém adresáři projektu, ke kterému jsou psány jednotkové testy. Dále jsou testovací třídy, tj. test case (jeden test) nebo test suite (pokud obsahují více test case), umístěny v adresářích, podle jejich jmenného prostoru. Jak je na ukázce kódu 2 ukázáno, každý test, musí dědit od třídy `TestCase`, což umožňuje použití metod např. `this->assertEquals(...)`. Dále je k dispozici například generování přehledného reportu o pokrytí kódu jednotkovými testy. Další podrobnosti viz dokumentace [2]. Následuje ukázka kódu demonstrující funkcionalitu a syntaxi frameworku PHPUnit.

Ukázka testu v PHPUnit

```
namespace app\entities;
use app\entities\Person;
use PHPUnit\Framework\TestCase;
class PersonTest extends TestCase
{
  private Person $pers;

  public function setUp() : void {
    $this->pers = new Person("John");  //arrange
  }

 /**
   * @group PersonTests
   */
  public function test_getName() : void {
    $expectedName = "John";
    $actualName = $this->pers->getName(); //act
    $this->assertEquals($expectedName, $actualName); //assert
  }
  public function tearDown(): void
  {
    fwrite(STDOUT, " Vypis po kazdem testovacim pripadu\n");
  }
}
```

Ukázka kódu 2: Struktura jednotkového testu pro framework PHPUnit

Na ukázce kódu 2 je ukázán test case, nebo test suite obsahující jeden test case PersonTest doménové třídy Person. Třída má jako jediný parametr konstruktoru jméno (`string name`). Tento test demonstruje nastavení jména a poté verifikuje, zda jej vrací instanční metoda, pomocí které bylo jméno nastaveno entitní v konstruktoru.

Části jednotkového testu

Jednotkový test má mít ideálně tři části. `Arrange` (příprava testovaných dat), `Act` (vykonání akce, která je testována), `Assert`(verifikace funkcionality). Výše zmiňovaná terminologie může být známa též jako `Given, When, Then`. Jako první je na obrázku uvedena metoda `setUp()`, která se automaticky vykoná před každým testem. Dále metodu `tearDown()`, která se

vykoná po každém testu. PHPUnit dále umožňuje používat statické metody s názvem `setUpBeforeClass()` a `tearDownBeforeClass()`, které provedou akci, která je definována v těle této metody vždy na začátku testovacího balíku, tj před provedením všech testů, nebo po provedení všech testů. Tj. před provedením všech testů v dané třídě. Obdobné funkcionality lze dosáhnout použitím libovolných názvů metod s korektními anotacemi, které jsou popsány v dokumentaci [2].

Anotace

PHPUnit umožňuje využívat anotace pro specifikaci další funkcionality testů. Například na obrázku je anotace `@group`, která dává výše zmíněný test do skupiny `PersonTest`. Toto umožňuje spouštět vždy jen testy, které jsou ve skupině `PersonTest`. K dispozici jsou dále anotace, které umožňují specifikovat závislost testu na testu předchozím (`@depends`), či specifikovat metodu, která testu dodá data (`@dataProvider, @testWith`) a tím z něj vytvoří tzv. parametrický test.

Aserce

Aserce označuje vyhodnocení správnosti testů. Umožňuje vyhodnocení shodnosti, hodnot, či dokonce ověření, zda pole obsahuje daný klíč. Dále je k dispozici metoda, která vyhodnotí, zda se v poli nachází daná položka. Aserce dodává testu jednoznačný výsledek, tj, test prošel, či neprošel. Pokud test neprošel, musí být rozlišován výsledek `Error` (aserce neproběhne a test skončí například na vyhození výjimky někde uprostřed testu), nebo `Failure` (aserce proběhne, ale aktuální výsledek nesouhlasí s očekávaným). Vždy se jedná o metody typu `assertXY`, kde `XY` označuje to, co metoda kontroluje.

Spouštění testů

Testy lze spouštět jak z vývojového prostředí, pokud to dané vývojové prostředí podporuje, tak i z příkazové řádky, což umožňuje automatizované spouštění testů např. při CI/CD[3] přístupu.

Integrace do Symfony

Symfony využívá k testování balík `symfony/test-pack`, který lze stáhnout pomocí Composeru. Tento balík obsahuje PHPunit a další knihovny potřebné pro testování Symfony aplikace. Díky tomuto balíku je možné vyko-

[3] Continuous Integration/Continuous Deployment

návat testy jednotkové, integrační (zkoumají že spolu části aplikace korektně komunikují), či aplikační (zkoumají, zda se aplikace chová očekávaně, např. posílá správné odpovědi na requesty) [13].

2.2.2 Testovací framework Nette Tester

Tento testovací framework disponuje nutnou funkcionalitou potřebnou pro testování PHP aplikací. Jelikož spolu s PHPUnit tvoří frameworky pro jednotkové testování, jsou proto možnosti a syntaxe těchto frameworků obdobné. Oproti PHPUnit nabízí méně funkcionality, ale pro tvorbu jednotkových testů dostačuje. Umožňuje například generování reportu pokrytí jednotlivých jednotkových testů. Postrádá tzv. tagování testů, které bylo zajištěno v PHPUnit anotací `@group`, dále nenabízí určení pořadí vyhodnocování testů podle závislosti spouštěných testů na testech ostatních, což je ve frameworku PHPUnit docíleno anotací `@depends` [2].

Ačkoli Nette Tester neumožňuje grupování jako takové, umožňuje spouštět vybrané skupiny testů tak, že se vyskytují ve stejném adresáři. Ve větších projektech je proto zvykem testy seskupovat v adresářích pojmenovaných podle jmenného prostoru testovaných tříd. Tím dosáhneme této funkcionality a navíc se vytvoří hierarchie pro orgranizaci testů. Nedostatky tohoto testovacího frameworku jsou vyváženy jednoduchostí. Každý testovací skript (test suite) je v podstatě spustitelný PHP soubor, což může být vhodné pro následnou integraci testů do CI/CD procesu [8].

Ukázka testu v Nette Tester

```
1  class PersonTest extends Tester\TestCase
2  {  public function setUp() { } //ukazka setUp
3
4     public function tearDown() { } //ukazka tearDown
5
6     public function testPersonName() {
7        $person= new Person("John"); //arrange + act
8        Assert::same("John", $person->getName()); //assert
9
10    }
11
12    //test ocekavaneho vyhozeni vyjimky
13     /** @throws InvalidArgumentException */
14    public function testPersonNoName() {
```

```
15         $person= new Person("");
16     }
17 }
18 // Spusteni testovacich metod
19 (new PersonTest)->run();
```

Ukázka kódu 3: Struktura jednotkového testu pro framework Nette Tester

Jak je ukázáno na ukázce kódu 3, Nette Tester disponuje systémem anotací jako PHPunit. Možnosti anotací jsou velice podobné, navíc lze použít například anotaci `@throws`, která automaticky zajistí, že test projde, pokud bude vyhozena výjimka typu `InvalidArgumentException`. Dále je ukázáno, že každý test suite (viz výše) musí dědit od třídy `TestCase`, což umožní například spouštění testu. Dále je předvedena funkcionalita připravení a úklidu po testech pomocí funkcí `setUp()` a `tearDown()`, které se spustí vždy před a po vykonání metody *test-názevTestu*, či metody s anotací `@test`.

Aserce

Aserce neboli vyhodnocení výsledku testu, jsou tvořeny pomocí statických metod, které se vyskytují ve třídě `Assert`. Název statické metody označuje vždy zkoumané kritérium. Nette tester nabízí celkem 23 typů asercí. Dále tento testovací framework nabízí tzv. očekávání pomocí třídy `Expect`. Očekávání lze použít, pokud je testována metoda, jejíž výstup závisí pouze na korektním formátu, či entitní třídě, ve kterém je výstup vracen. Například lze očekávat, že se jedná o instanci třídy `DateTime` nebo řetězec musí být číslo v hexadecimální podobě [8].

Spouštění testů

Testy vytvořené nástrojem Nette Tester lze spouštět hromadně pomocí příkazové řádky prostřednictvím nástroje Nette Tester, nebo jednotlivě jako samostatné PHP skripty. Tímto je umožněno spouštění testů dokonce i z prohlížeče, ačkoli to není standardní způsob. Dále je v dokumentaci popsána možnost pro spouštění testů pomocí nástroje Travis CI, který zajistí spouštění testů při nastavené CI/CD pipeline[4].

Integrace do Nette

Testovací framework je do aplikace dodán pomocí balíčku Composer. Defaultně se testy umisťují do adresáře `tests`, kde se nachází `bootstrap.php`

[4]Posloupnost kroků, které se provádí při začlenění nové verze software

pro definici testovacího prostředí a následně testy.

2.3 Kontinuální integrace a kontinuální nasazení

Tyto metody nejsou zcela vlastnosti PHP frameworků, avšak poskytují vhodnou funkcionalitu k zefektivnění vývoje, za využití PHP frameworků.

2.3.1 Kontinuální integrace

Kontinuální integrace neboli *continuous integration* (zkráceně CI) je pojem definující metodu vývoje softwaru, kde členové vývojového týmu, nebo více týmů často integrují své úpravy do vyvíjené aplikace. Postup integrace je za použití CI automatizovaný. Každá integrace je ověřena automatickým sestavením (včetně testování), což značně zvýší efektivitu vývoje a kvalitu výsledného projektu, neboť jsou včas zachyceny defekty, na jejichž opravu by se v budoucnu muselo vynaložit úsilí, které by jinak mohlo být věnováno vývoji další části projektu [12].

Proces kontinuální integrace

Ve vývojové větvi, ve které je vytvořen nový kus kódu, který má být později integrován do projektu, nejprve vývojář spustí testy lokálně. Pokud všechny projdou, vývojář využije CI a kód integruje pomocí systému pro správu verzí do vyvíjené aplikace. CI server změny zkontroluje pomocí vytvořených testů a vytvoří z nich nový build. Pokud kontrola selže na jedné z definovaných oblastí, které CI kontroluje, vývojář a jeho tým je informován o místě a oblasti, kde selhala kontrola právě vytvářeného přírůstku k aplikaci. Pokud nový build neprojde kontrolou, může být ihned pracováno na opravě, což značně zefektivní rychlost vývoje [4].

Pokud build projde, je integrován do vytvářené aplikace. Oblast lze specifikovat při konfiguraci CI procesu, která je dostupná na CI serveru, či na systému pro správu verzí, který zahrnuje funkcionalitu CI/CD. CI server je dnes integrován do většiny systémů pro správu verzí. Kontinuální integrace bývá často spojená s pojmem kontinuální nasazení. Tyto dvě techniky označují zkratku CI/CD, která je často vidět ve webových rozhraní mnoha nástrojů pro správu verzí.

2.3.2 Kontinuální nasazení

Jak již bylo řečeno, kontinuální nasazení je technika, která je spjatá s kontinuální integrací. Kontinuální nasazení je povětšinou použito, když úspěšně proběhla kontinuální integrace. Jedná se o dodávku kódu do cílového prostředí softwarového produktu [12]. Prostředí může být popsáno tak, že se softwarový produkt dostane do kontaktu s uživatelem softwaru.

2.3.3 Možnosti kontinuální integrace pro Nette Tester

Jak je výše zmíněno, Nette Tester je framework pro jednotkové testování webových aplikací. Nette Tester lze provázat pomocí nástroje Travis CI (jedná se o CI server). Jedná se o externí službu, pro kterou je nutno nastavení tzv. webhooku, který odchytí veškeré pokusy o integraci, které dále obslouží. Tato služba je k dispozici zdarma a nabízí automatizované spouštění testů po pokusu o integraci (merge do hlavní větve) do repozitáře. Travis CI je integrován do systému GitHub (integrace s GitLab, BitBucket a Assembla jsou v beta verzi) [8].

Nastavení Travis CI

Travis CI se nastavuje pomocí konfiguračního souboru `.travis.yml`, který je nutno umístit do kořenového adresáře projektu. V tomto souboru je nutno odsazovat pouze mezerami, k zachování validity konfiguračního souboru. Konfigurační soubor může obsahovat bloky `before_install`, `install`, `before_script`, `script`, `after_success`, `after_failure` a `after_script`. Tyto bloky jsou dále spouštěny ve zde uvedeném pořadí. Důležitý je zde krok s názvem script, který definuje příkaz ke spouštění jednotkových testů. Pokud by v konfiguračním souboru nebyl uveden, nástroj Travis CI by předpokládal, že se budou testy spouštět pomocí nástroje `PHPUnit`. Následuje ukázka konfiguračního souboru, která je podrobně popsána [17].

```
1  language: php
2  php:
3    - 7.1
4    - 7.4
5    - nightly
6  env:
7    - TESTER_PHP_BIN="php"
8  matrix:
9    allow_failures:
10     - php: nightly
```

```
11   exclude:
12     - php: 7.1
13       env: TESTER_PHP_BIN="php-cgi"
14 before_install:
15   - composer self-update
16 install:
17   - composer install --no-interaction --prefer-source
18 before_script:
19   - mysql -u root -e 'CREATE DATABASE testdb;'
20   - mysql -u root testdb < tests/testdb.sql
21 script:
22   - ./vendor/bin/tester -p $TESTER_PHP_BIN -c ./tests/php.ini
      -s ./tests/
23 after_failure:
24   - for i in $(find ./tests -name \*.actual); do echo "--- $i"
      ; cat $i; echo; echo; done
```

Ukázka kódu 4: Struktura konfiguračního souboru pro Travis CI

Na ukázce kódu 4 je první definovaná sekce `language`, která určuje, pro jaký jazyk se má testovací prostředí nastavit. Jako další je sekce `php`, která definuje, na jakých verzích jazyka PHP se testy budou spouštět. Verze `nightly` odpovídá verzi testovaného projektu. Dále následuje sekce `env`, kde se definují další parametry testovacího prostředí. V konkrétním případě to je nastavení binárního souboru jazyka PHP pro spouštění testů. Nynější nastavení provede spouštění testů celkem 3x (pro každou variantu sekce `env` a `php`, což definuje množinu testů, která je dále v konfiguračním souboru nazývána matice (matrix). Jelikož je Composer již součástí Travis CI, tak v sekci `before_install` je composer aktualizován a v sekci `install` jsou nainstalovány závislosti testovaného projektu [17].

Následně je podrobněji specifikována matice testů pomocí sekce `matrix`, která obsahuje vynechání zde popsaných kombinací z matice testů a povolení selhávání testů na specifické verzi PHP pomocí příkazu `allow_failures`. Pokud tedy libovolný test spouštěný s vývojovou verzí PHP neprojde, build je i tak označen jako procházející a projde testy. Pokud testy spolupracují s databází (zde se předpokládá MySQL, protože je součástí TravisCI), je zde definována sekce `before_script`, která před spuštěním testů vytvoří testovací databázi. Dále je pomocí sekce `script` definován příkaz pro spouštení testů. Zde je nutno používat proměnou nastavenou v sekci `env`, pro vytvoření výše popsané matice testů. Nakonec je v sekci `after_failure` definována akce, která se provede, pokud testy neprojdou [17].

3 Metody a přístupy k testování webových aplikací

Webové aplikace lze testovat širokou škálou testů, avšak jejich cíl je vždy stejný, nebo alespoň podobný. Ověřit, zda očekávané fungování aplikace vypovídá reálnému stavu. Chování aplikace se zkoumá v různých testovacích prostředích, které se více, či méně podobají cílovému produkčnímu prostředí, dle metody používané při testování. Tyto metody bývají většinou kategorizovány jako tzv. testy černé, či bílé skříňky, které jsou popsány dále. Testy pro webové aplikace mohou ověřovat i další dimenze kvality, jako například testy použitelnosti, testy spolehlivosti, výkonnostní testy, bezpečnostní testy a zátěžové testy, avšak tyto specifičtější typy testů nebudou v této práci dále popisovány a používány.

3.1 Typy testů z hlediska viditelnosti zdrojového kódu

3.1.1 Testování černé skříňky

Tento princip může být definován tak, že tester vidí chování testovaného software. Vidí odezvy na všechny provedené operace a může tak odhadovat, jak se software zachová příště. Tester nevidí vnitřní logiku aplikace a nezná strukturu zdrojového kódu. Interaguje s aplikací pouze na základě nějakého uživatelského rozhraní tak, jako s ní bude interagovat cílový uživatel. Nevýhodou tohoto způsobu je, že často vede k nadměrnému testování některých částí aplikace a k nedostatečnému otestování jiných částí aplikace [10].

3.1.2 Testování bílé skříňky

Testování bílé skříňky označuje proces, kdy vnitřní logika aplikace a zdrojový kód slouží k lepšímu návrhu testovacích případů. To znamená, že lze snadněji odchytit a ověřit korektní fungování v částech software, které jsou závislé na částech ostatních. Tester musí být v tomto případě osoba, která se dobře vyzná ve zdrojovém kódu daného software a má k němu přístup [10].

3.1.3 Testování šedé skříňky

Tato metoda je z hlediska přístupnosti zdrojového kódu někde na pomezí testování černé a bílé skříňky, kde je implementace částečně známa. Je to typická metoda pro testování webových aplikací, neboť je díky webovým prohlížečům dostupný HTML kód aktuálně zobrazené stránky. Výhodou je, že testy lze navrhovat se zaměřením na funkčnost systému s optimálním pokrytím potenciálních slabých míst (či míst vyšší složitostí), které by byly jinak s největší pravděpodobností přehlédnuty. [10].

3.2 Úrovně testování

3.2.1 Jednotkové testování

Jednotkové testování neboli testování jednotek se zaměřuje na malé jednotky kódu. Tato úroveň je popsána v kapitole 2.2.

3.2.2 Integrační testování

Tato úroveň testování se zaměřuje na ověření bezchybné spolupráce jednotlivých aplikačních komponent. Zkoumá tedy jejich integraci. V podstatě to vypadá tak, že se nejprve zkoumá integrace výše zmiňovaných jednotek kódu, a postupně se přidává množství jednotek, či komponent. Tuto spolupráci lze ověřit jak mezi aplikačními komponentami, tak i například mezi komponentou a operačním systémem. V menších projektech bývá tato úroveň často vynechávána, jelikož výslednou bezporuchovost lze ověřit například i systémovým testováním [11].

3.2.3 Systémové testování

Systémové testování bývá také někdy označováno jako systémové integrační testování. S aplikací je v této úrovni testů pracováno jako s jedním funkčním celkem a je s ní interagováno tak, jako by s ní interagoval zákazník. Práce s aplikací probíhá za pomoci připravených testovacích scénářů, které představují různé činnosti a stavy, se kterými se v aplikaci musí počítat. Součástí této úrovně jsou například funkcionální testy. Pokud by byla tato úroveň vynechána, je ohrožena výsledná bezporuchovost výsledného produktu [11].

3.2.4 Akceptační testování

Akceptační testování probíhá za účelem akceptace daného softwarového produktu zákazníkem. Testování probíhá pomocí předpřipravených testovacích scénářů, které byly připraveny společně se zákazníkem a dodavatelem testovaného software. Tyto scénáře nejčastěji vykonává zákazník (není to vždy pravidlem) a na základě výsledku se rozhoduje, zda výsledný software v aktuálním stavu přijme, či nikoli [11].

3.3 Druhy testů

3.3.1 Funkcionální testy

Funkcionální testování neboli dynamické testování černé skříňky je postup, při kterém je s aplikací interagováno takovým způsobem, jako by ji obsluhoval běžný uživatel. Do software jsou zadávány vstupy, ke kterým je následně kontrolována odezva. Například v kontextu webových aplikací to může být potvrzující hláška po odhlášení uživatele. Funkcionální testy můžou být testy splněním, nebo testy selháním. Testy splněním neboli pozitivní testy, reprezentují akce, které provádí běžný uživatel, tj. není snaha software za všech okolností dostat do nedefinovaného stavu, nebo jej dokonce shodit.

Hlavním cílem je ověřit správnou funkčnost při běžném použití. Dále testy selháním neboli negativní testy reprezentují akce, které mají za cíl aplikaci shodit, nebo dostat do nějakého extrémního, neošetřeného stavu, případně ověřit správnou reakci na tyto stavy. Pro otestování software volíme množinu testů dle situace, ve které se nacházíme. Pokud testujeme při vývoji, je vhodné začít s pozitivními testy, a poté využít testy negativní. Pokud tvoříme testovací plán pro již existující aplikaci, je vhodné zvolit oba výše zmíněné typy testů ve vhodných poměrech.

3.3.2 Nástroje pro funkcionální testování

Jelikož je hlavní úsilí vedeno směrem automatizovaných testů, dále budou zmíněny pouze možnosti testování webových aplikací nástroji, kterými lze webové aplikace ovládat, je tedy cíleno na automatizaci testovacího procesu. Jelikož se k webovým aplikacím přistupuje přes webový prohlížeč, ve kterém je snadno zjistitelný HTML kód dané stránky, využívá se proto lokalizace pomocí `XPath`, jednoznačných identifikátorů (`ID`), typů elementů, CSS tříd a dalších vlastností DOM modelu.

Testovací framework Selenium WebDriver

Nejprve je nutno říct, že se jedná pouze o framework, který umožní ovládání webové aplikace a získat informace, které webové aplikace vypisují. Pro výsledné vyhodnocení testů musí být doplněn libovolným nástrojem pro spouštění a vyhodnocení testů. Poskytuje ovládání širokému spektru webových prohlížečů včetně Google Chrome, Firefox, Opera, Safari, Microsoft Edge, atd... Může běžet například i v tzv headless režimu, což je prohlížeč, který ovládá webovou aplikaci bez nutnosti jejího zobrazení. Tento framework je k dispozici pro `Javu`, `Python`, `C#` a `Ruby`. Díky značně rozsáhlému API lze vytvářet ovládání webové stránky, které s efektivním návrhem odstraní rozsáhlé opakování kódu [14].

Testovací nástroj Selenium IDE

Jedná se o doplněk do webových prohlížečů Google Chrome a Firefox, který lze ve značně omezené míře použít na funkcionální testování. Od počátku byl zamýšlen jako nástroj pro prototypování a rychlou automatizaci stále stejných činností, opakujících se činností. Velkou výhodou tohoto nástroje je jednoduchost. Testovací skripty lze vytvářet zaznamenáním právě prováděné akce. Pokud by byl tento nástroj použit pro testování nějaké větší aplikace, vyvstanou nevýhody, jako například opakování kódu. V podstatě nenabízí možnosti genericity při vytváření testů, jako například Selenium Webdriver [14].

3.3.3 End-to-End testy

End-to-End neboli E2E testování je druh testování, který testuje softwarový produkt od počátečního bodu (například přihlášení) do koncového bodu (konec nějaké posloupnosti akcí) [11]. Je sledována určitá entita (objekt, data, ...) po celou dobu její životnosti napříč celým systémem (od jednoho konce ke druhému). Aplikace je testována jako celek, se všemi závislostmi a včetně všech aplikačních komponent. Tato testovací technika má za cíl vyzkoušet, zda spolu všechny části aplikace spolupracují podle očekávání. Hlavní myšlenkou end-to-end testování je otestovat funkcionalitu z pohledu koncového uživatele, jelikož jednotlivé testy jsou složené z kontrolovaných posloupností akcí, které na sebe navzájem navazují a zároveň se ovlivňují. Tyto opakující se posloupnosti jsou nazývány testovací scénáře.

Testovací scénář

Testovací scénáře by měly být vytvářeny v souladu s požadavky na danou aplikaci, či v souladu s jejími případy užití, nebo na základě testovacího plánu. Pokud budou tyto dokumenty brány v potaz při návrhu testovacích scénářů, může být značně zvýšeno pokrytí testované aplikace testy. Test může v kontextu e-shopu vypadat například tak, že uživatel přidá zboží do košíku, vyplní a odešle objednávku se všemi náležitostmi. Entita zboží je tím sledována až do konce posloupnosti akcí, které provádí zákazník. S každým krokem je zde kontrolován stav systému, aby bylo zachyceno případně co nejvíce chyb.

Vykonávání end-to-end testů

Tento proces zahrnuje množství akcí, které můžou být vykonávány a zároveň kontrolovány manuálně, avšak je vhodné tento proces automatizovat, jelikož scénáře můžou být dlouhé a při vyhodnocení testu by mohl lidský faktor jednou chybou negativně ovlivnit výsledek celého testu. Používají se proto různé frameworky pro ovládání webových stránek, jako například výše popsaný Selenium Webdriver, nebo například cypress.io, což je JavaScript framework pro automatizaci ovládání webových stránek.

4 Dokumentace k testování

Testovací dokumentace je dle [3] a [10] neodmyslitelná součást testování, která se vyskytuje v různých podobách. Tuto podobu určuje například standard `IEEE 829-2008` pro tvorbu dokumentace k testování, nebo standard `ISO 9001:2000`, či různých metodik pro tvorbu testových dokumentů, jako například `TMap Next a ISTQB` [3]. Použití standardizovaných, či jinak zavedených metodik pro tvorbu dokumentace má své výhody, jako například jednoznačnost a jasný proces kontroly kvality. Nevýhodou můžou být však nákladnost tvorby dokumentace a nepružnost v případech, kdy dochází k častým nebo výrazným změnám v požadavcích na systém.

4.1 Testovací plán

Jako nejdůležitější dokument pro testování může být považován plán testování, který řídí celý proces testování. Dle [3] je rozlišován hlavní plán testování (Master test plan) a plán testování (Test plan), kdy je hlavní plán testování zaměřen na testování více úrovní testů a plán testování pouze na jednu úroveň. Struktura hlavního plánu testování je dle knihy [3] následující.

4.1.1 Struktura hlavního plánu testování dle metodiky TMap Next

- **Záznam o schvalování s klientem**
- **Manažerské shrnutí**
- **Úvod** – cíl projektu, cíl dokumentu, autoři dokumentu a připomínkující osoby
- **Formulace cílů a rozsahu testovacího projektu** – specifikace klienta a dodavatele, co je cílem testování a co bude výstupem v rámci testovacích aktivit, definice rozsahu testování, výchozí podmínky a předpoklady, kdo je akceptující strana a jaká jsou akceptační kritéria testovaného systému.
- **Použitá dokumentace** – použité standardy, soupis vstupních informací pro návrh testovacích scénářů (přehled dokumentů se specifikací atd.), další použité dokumenty

- **Strategie testování** – konkrétní cíle testování, analýza rizik testované aplikace, úrovně testování, části aplikace, které se budou testovat a intenzita testování v nich

- **Přístup k testování** – seznam úrovní testování, pro každou úroveň pak:

  - konkrétní cíle testování v dané úrovni
  - popis úrovně a kdo je zodpovědný za danou úroveň testování
  - jaké produkty testování vzniknou
  - kdo je bude revidovat
  - vstupní a výstupní kritéria pro úroveň testování
  - postup při rozhodování na konci úrovně testování, zda testovaný systém uvolnit do další testovací úrovně nebo do produkce.

- **Organizace testovacího projektu** – organizační struktura testovacího podprojektu, role, úkoly a zodpovědnosti, struktura a plán projektových schůzek, přehled reportů, které budou v průběhu testování vznikat, procedura po dokončení celého testování,

- **Infrastruktura pro testování** – potřebné testovací prostředí, potřebné testovací nástroje

- **Řízení testovacího projektu** – popis pravidel pro řízení testovacího procesu, popis pravidel pro řízení testovací infrastruktury, procedura pro správu a řízení chyb

- **Seznam rizik a opatření proti nim**

- **Odhady a (projektový) plán** – odhady pracnosti testů, plán testů, hlavní milníky

4.1.2 Struktura detailního plánu pro jednotlivé úrovně testování dle metodiky TMap Next

Následuje ukázka detailního plánu pro jednotlivé úrovně testování dle metodiky `TMap Next`, který slouží pro specifikaci testování dané úrovně v rámci hlavního plánu testů. Dle [3] struktura pro záznam o schvalování s klientem, manažerské shrnutí, cíl dokumentu, formulace cílů a rozsah testovacího projektu je dle TMap strukturně shodná s hlavním plánem testování. Dále plán pokračuje následující strukturou.

- **Použitá dokumentace** – přehled dokumentů, ž nichž vychází plán testování pro danou úroveň testování, soupis vstupních informací pro návrh testovacích scénářů(specifikace požadavků, případy užití, . . .)

- **Strategie testování** – odkaz na konkrétní testovací strategii pro danou úroveň testování do hlavního plánu testování, popis a zdůvodnění případných odchylek

- **Přístup k testování** – popis použitých testovacích technik a technik pro návrh testovacích scénářů, popis procedury pro převzetí testovaného systému a dalších náležitostí podle vstupních kritérií, vstupní a výstupní kritéria pro úroveň testování.

- **Infrastruktura pro testování** – Potřebné testovací prostředí

- **Řízení testovacího projektu v dané úrovni testování** – popis pravidel pro řízení testovacího procesu, procedura pro správu řízení a chyb

- **Odhady a projektový plán** – odhady pracnosti test a plán testů, ve smyslu projektového plánu

4.1.3 Zásady při vytváření testovacího plánu

Testovací plán by měl být psán stručně, a pokud možno by měl být udržován aktuální. Budoucím testerům přispěje k efektivitě práce stručnost a jasnost a věcnost informací zanesených v plánu, neboť se můžeme setkat s tím, že velice obsáhlý plán, z hlediska počtu stran, nebude nikdo chtít číst.

V praxi se často stává, že se nalezne zastaralá verze plánu testů, neboť jej účastníci vývoje přestanou brát vážně a začnou si informace předávat ústně, nebo například pomocí emailů. Této situaci se dá dle [3] zabránit tak, že se plán testování průběžně aktualizuje a struktura se zvolí tak, aby obsahovala pouze podstatné informace. Případně je možné zachovat strukturu určenou definovanou metodikou a do určité úrovně nadpisu nevyužívané části napsat důvod, proč je nevyužita [3]. Tento bod také potvrzuje [10], který zmiňuje, že důležitý je proces plánování, a ne výsledný dokument. Dále se dá této situaci předejít zvolením vhodným způsobem umístnění dokumentu (plán jako stránky na firemní wiki). Nakonec je nutné zdůraznit nutnost konkretizování informací v dokumentu, a zanesení pouze takových informací, které jsou relevantní k testovanému projektu.

5 Analýza testované aplikace

5.1 Účel webové aplikace Podpůrný software TSP

Aplikace Podpůrný software TSP (dále PSTSP) má sloužit jako softwarová podpora pro předměty z katedry informatiky Týmový softwarový projekt 1 a 2, které budou vyučovány v magisterském studiu od akademického roku 2022/23. Tyto předměty budou představovat týmovou práci na projektech překračující hranice ročníku, jelikož TSP1 bude v letním semestru a TSP2 bude končit obhajobou celého projektu v zimním semestru. Na práci se budou podílet týmy studentů dozorované různými mentory. Zadání projektů bude od různých zadavatelů. Projekt bude během zpracování procházet různými fázemi, což bude v aplikaci evidováno. Cíloví uživatelé aplikace jsou tedy vedoucí jednotlivých týmů, mentoři, kteří evidují a kontrolují postup jednotlivých týmů, které mentorují a garant, který je zodpovědný za organizaci předmětů a mj. spravuje chod celé aplikace (vkládá a spravuje témata, konta, ...). Celá aplikace bude navíc fungovat jako rezervační systém, kde vedoucí týmu budou mít možnost vyjádřit zájem o téma (které do aplikace vloží garant a následně dle zájmů přiřadí jednotlivým týmům) a mentoři si budou moc i přiřadit téma, které dosud nemá mentora. Po přiřazení tématu s mentorem k týmu se z týmu bez projektu stává tým s projektem.

5.1.1 Případy užití a požadavky na PSTSP

K webové aplikaci jsou zpracovávány případy užití, ve formě `XML` souborů, které obsahují detailní popis daného případu užití. Tyto případy užití budou následně transformovány v požadavky (RQM), ze kterých budou odvozeny jednotlivé testovací případy. K uchovávání požadavků a k organizaci testovacích případů byl zvolen nástroj SquashTM, díky jednoduché instalaci a jednoduchému ovládání. Zároveň jsou s nástrojem zkušenosti z KIV/OKS.

5.1.2 Výběr vhodných testovacích nástrojů

Pro realizaci funkcionálního a end to end testování byl zvolen programovací jazyk Java společně s nástrojem Selenium WebDriver v kombinaci s frameworkem JUnit5 pro vyhodnocení výsledku testů, pro možnosti modularity a znovupoužitelnosti kódu pro ovládání aplikace přes webový prohlížeč.

Pro jednotkové testování byl zvolen PHP framework Nette Tester, pro jeho jednoduchost a rychlost psaní testovacích skriptů.

5.2 Testovací plán pro PSTSP

Pro vytvoření testovacího plánu bude využita kostra hlavního testovacího plánu, ze které budou vynechány body týkající se managementu osob, které se starají o testovací proces, jelikož testy budou vytvářeny pouze jedním testerem. Naopak v bodě přístup k testování budou jednotlivé úrovně více rozvinuty. Aplikace bude testována pouze na třech úrovních, a to funkcionálními a end-to-end testy(systémové testy) a zároveň manuálními akceptačními testy, které bude provádět zadavatel, tj. vedoucí bakalářské práce. Poslední úroveň bude představovat vývojářské jednotkové testování, které bude netradičně prováděno testerem aplikace. Funkcionální, end-to-end a jednotkové testy budou náležitě automatizovány, a zadavateli bude poskytnuto hlášení o průběhu těchto testů, kde úspěšnost testů bude též představovat kritérium pro přijetí části aplikace.

5.2.1 Úvod

Projekt má sloužit jako softwarová podpora pro organizaci předmětů TSP1 a TSP2, které budou vyučovány v magisterském studiu od akademického roku 2022/23 na katedře informatiky. Tento plán testů má sloužit k organizaci procesu testování pro verzi aplikace 0.9.0.

5.2.2 Formulace cílů a rozsahu testovacího projektu

Nějakým druhem testů (specifikováno dále) bude testována každá akce, kterou lze v aplikaci provést a proběhne v pořádku, tj. zadají se korektní data. Případy, kdy se zadají nekorektní data se vyskytují zřídka, jelikož se v aplikaci vyskytuje zanedbatelné množství vstupních polí, kde je možné nekorektní data zadat. I přes to bude aplikace pokryta negativními testy v části, kam bude přistupovat nejvíce uživatelů. Testy by měly dosahovat vysoké hodnoty pokrytí, pokud bude uvažováno pokrytí jednotlivých požadavků testovacími případy. Pokrytí kódu jednotkovými testy bude téměř zanedbatelné, díky tomu, že je aplikace vytvořena v Nette framework, který nabízí určité abstrakce nad databází, takže se testovatelné jednotky kódu vyskytují pouze zřídka.

5.2.3 Použitá dokumentace

K aplikaci je dostupná dokumentace, které tvoří případy užití, ze kterých se následně vyjme popis úspěšné akce, a z něj se standardním způsobem vytvoří požadavky do SquashTM, na které se poté tvoří testovací případy. Ty představují vyzkoušení definované množiny vstupních dat z hlediska optimálního rozdělení tříd ekvivalence.

5.2.4 Strategie testování

Cíle testování

ID	Cíl testování
1	Prokázání kvality a korektního chování u role vedoucí
1.1	Funkční správa týmu
1.2	Funkční editace stavu projektu
1.3	Funkční možnost vyjádřit zájem o téma
2	Prokázání kvality a korektního chování u role mentor
2.1	Funkční změna stavu u mentorovaného projektu
2.2	Funkční možnost mentorování zadání
3	Prokázání kvality a korektního chování u role garant
3.1	Funkční správa aplikace (témata, týmy, mentoři)
4	Prokázání kvality a korektního chování u role nepřihlášený uživatel
4.1	Funkční přihlášení
5	Prokázání dostatečného pokrytí požadavků testy
6	Prokázání dostatečného pokrytí vybraných částí kódu jednotkovými testy.

Tabulka 5.1: Cíle testování

Rizika aplikace

Rizika aplikace jsou zaneseny do systému SquashTM, kde jsou zaznamenány veškeré požadavky s jejich prioritami. Pokud má tedy požadavek velkou prioritu, tak by měl při selhání i velký dopad na celý systém. Přednostně tedy byly připravovány testy s vysokou prioritou. Pokud vezmeme v úvahu jako možné riziko mírné zastoupení negativních testů, není tomu tak, jelikož se v aplikaci nachází zanedbatelné množství vstupních polí, jejichž funkcionality nejsou pro aplikaci nějak zásadní.

Úrovně testování

Aplikace se bude testovat na třech úrovních, a to na úrovni systémových (funkcionální a end-to-end testy) testů, akceptačních testů (manuální, prováděné zadavatelem) a vývojářských jednotkových testů.

Části aplikace

Modul vedoucího týmu, modul mentora, modul garanta a modul nepřihlášeného uživatele.

5.2.5 Přístup k testování

Pro úroveň jednotkového testování platí cíl s ID 6. Pro ostatní dvě úrovně platí všechny ostatní cíle.

Úroveň jednotkového testování

- **Popis** – Jednotkové testování je v aplikaci řešeno v malém rozsahu, avšak dostatečně, jelikož díky využití Nette Framework pro vývoj, aplikace téměř neobsahuje funkcionality, které by byly jednotkově testovatelné. S vývojářem byla dohodnuta konvence ohledně umisťování jednotkově testovatelných metod do třídy `Utils`. Ta musí být pokryta jednotkovými testy.
- **Cíl** – dosáhnout značného pokrytí v rámci třídy, kde je jednotkové testování využito
- **Produkt** – Produktem testování bude výsledný report o pokrytí jmenované testovací třídy a report o výsledcích testů.
- **Řízení testovacího projektu** – pro tuto úroveň testování nebylo zřízeno žádné řízení testovacího projektu. Důležité je, aby bylo dosaženo cíle.
- **Testovací prostředí** – Pro tuto úroveň je nejdůležitější vlastnost prostředí mít nainstalované PHP verze 8, jelikož jsou spouštěny prostřednictvím lokálního PHP interpretru.
- **Testovací nástroj** – Nette Tester

Úroveň systémového testování

- **Popis** – Systémové testování je hlavním pilířem procesu testování dané aplikace. Testování probíhá na úrovni pozitivních a částečně negativních testů, které jsou realizovány funkcionálními testy. Ty provedou jednu změnu a poté změny zkontrolují na místech, které jsou změnou ovlivněny. Následně testování probíhá na úrovni negativních funkcionálních testů, kterými se pokryje modul vedoucího týmu. Dále tato úroveň obsahuje end-to-end testy, pomocí kterých se testují pouze entity v aplikaci, jejichž činnost lze nějakým způsobem sledovat při průchodu aplikací. Sledované entity můžou být například schůzka, zadání, tým, ... Pro zachování opakovatelnosti jednotlivých testů je definován stav aplikace (pomocí SQL skriptu), do kterého je aplikace uvedena před provedením jednotlivých testů.
- **Produkt** – Produktem této testovací úrovně budou reporty o výsledcích jednotlivých testů ve SquashTM a vizualizace pokrytí požadavků testovacími připady.
- **Řízení testovacího projektu** – Pro řízení testovacího projektu je využit SquashTM, kde jsou definovány veškeré požadavky a testovací případy označené písmenem C (pro pozitivní funkcionální testy), D (pro negativní funkcionální testy) a E (pro end-to-end testy).
- **Testovací prostředí** – Webový server Apache s nainstalovaným PHP verze 8, dále definovaný stav databáze pro možnost opakování testů.
- **Testovací nástroj** – Selenium WebDriver v kombinaci s JUnit5

Úroveň manuálního akceptačního testování

- **Popis** – Tato úroveň je zde reprezentována manuálním testováním aplikace zadavatelem. Vždy po dokončení dané části aplikace je funkcionalita aplikace prověřena zadavatelem, který následně vznese připomínky (jak k funkčnímu, tak k vzhledovému charakteru aplikace).
- **Produkt** – Produktem této testovací úrovně budou připomínky, které jsou zaznamenány ve formě *Issues* v GitLabu, nebo slovní vyjádření spokojenosti s částí aplikace na projektové schůzce.
- **Řízení výsledku testů** – Pro řízení testovacího projektu je využit modul v GitLabu, který se stará o *Issues*, kde zadavatel případně vytvoří jednotlivé *Issue*, které zabraňují k akceptaci jednotlivé části aplikace.

- **Testovací prostředí** – Webový server Apache s nainstalovaným PHP verze 8.

5.2.6 Infrastruktura pro testování

Jako testovací prostředí slouží webový server Apache s PHP verze 8, dále Selenium WebDriver v kombinaci s JUnit a NetteTester. Pro zaznamenávání požadavků vzniklých po provedení akceptačních testů slouží modul zaznamenávající Issues, který je integrovaný v GitLabu.

5.2.7 Kritérium pro uvolnění systému do produkce

Systém musí splňovat všechny cíle testování stanovené v bodě 5.2.4. a musí projít akceptačním testováním. Dále musí všechny navržené testovací případy s vysokou a střední prioritou projít.

5.2.8 Řízení testovacího projektu

Struktura dle jednotlivých druhů testů

Jednotkové testy jsou v adresáři `tests` přímo u aplikace. Dále jsou udržovány testovací projekty PSTSP-Support, který obsahuje veškerou logiku pro ovládání webové stránky a projekt PSTSP-functional-tests, který obsahuje veškeré typy testů, které využívají PSTSP-Support, tj funkcionální a end-to-end testy.

Procedura při nalezení defektu, či selhání při akceptačním testováním

Při výskytu selhání je testující povinen oznámit toto selhání vývojáři spolu s testovacím případem, který selhal. Dále probíhá nalezení defektu a jeho oprava na straně vývojáře, po které následuje retest. Pokud se defekt znovu neprojeví, defekt je brán jako odstraněný. Pokud zadavatel nesouhlasí se stavem aplikace, část aplikace nepřijme a povede záznam do GitLabu ve formě Issues, které jsou popsány v bodě 5.2.5 – Úroveň akceptačního testování – Řízení výsledku testů.

5.2.9 Plán vytváření testů

Pokud se objeví nějaká nová metoda, která lze jednotkově testovat, musí být ihned otestována. Dále budou testy pro moduly aplikace vytvářené v následujícím pořadí. Funkcionální testy nejprve pro modul nepřihlášeného uži-

vatele, dále pro modul vedoucího, poté pro modul mentora a nakonec pro modul garanta. V tomto pořadí bude též probíhat schvalování jednotlivých částí aplikace zadavatelem. V úplném závěru budou vytvořeny end-to-end testy, které ověří spolupráci výše popsaných modulů.

6 Realizace navržených testů

Pro vývoj výše popsaných úrovní testů byly zvoleny následující vývojová prostředí. PHPStorm pro vývoj jednotkových testů a IntelliJ Idea pro vývoj funkcionálních testů a end-to-end testů. Při návrhu testů byl kladen důraz na důkladné pokrytí testované aplikace.

6.1 Jednotkové testy

S vývojářem webové aplikace byla dohodnuta konvence, že veškeré jednotkově testovatelné metody bude umisťovat do specifické třídy `Utils`. Díky využití vývoji aplikace v pokročilém aplikačním frameworku, byl vývojář odstíněn z valné většiny od nízké úrovně, kterou lze testovat jednotkovými testy.

6.1.1 Struktura jednotkových testů v projektu

Jednotkové testy jsou umístěny v projektu s aplikací v adresáři `tests`. Dále jsou zde rozděleny dle jmenného prostoru testované třídy do adresářů, ve kterých jsou třídy reprezentující testy pojmenovány dle jmen metod, jelikož každý `test suite` testuje pouze jednu metodu tak, aby bylo dosaženo 100% pokrytí testované třídy. Tato konvence byla zavedena s důsledkem snazší orientaci v jednotkových testech.

6.1.2 Ukázka jednotkového testu

Na následující ukázce je demonstrováno testování metody pro ověření, zda se skutečně jedná o řetězec obsahující URL adresu.

```
1 <?php
2 class IsStringUrlTest extends Tester\TestCase
3 {
4    /** @test */
5    public function testIsUrl() {
6         $url = "https://www.fav.zcu.cz";
7         $expected = true;
8         $actual = Utils::isStringUrl($url);
9         Assert::equal($expected,$actual,"Url: ".$url." is not
     valid url!\n");
```

```
10        }
11    }
```

Ukázka kódu 5: Ukázka jednotkového testu v PSTSP

6.1.3 Výsledky testování

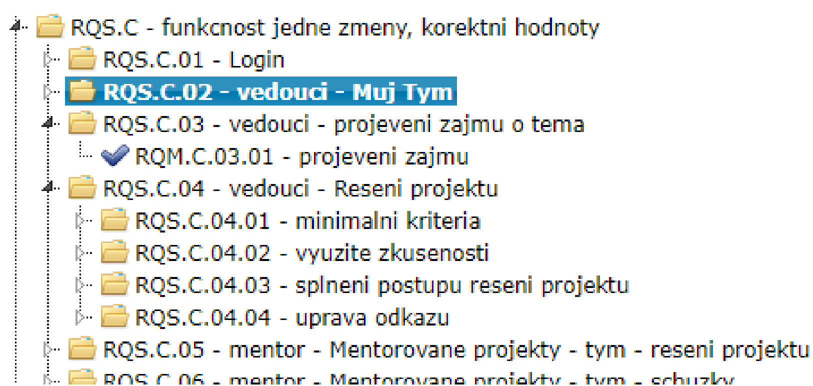
V rámci jednotkového testování bylo vytvořeno celkem 43 jednotkových testů, čímž bylo docíleno 100% pokrytí v rámci dané testované třídy. Informace o počtu a úspěšnosti jednotkových testů a informace o pokrytí jsou generovány nástrojem Nette Tester, který byl zároveň použit pro jednotkové testování.

6.2 Struktura požadavků a testovacích případů v SquashTM

Pro správu a návrh kostry funkcionálních a E2E testů byly využity případy užití (UC) testované aplikace, které byly následně transformovány na požadavky na software. Vždy existuje alespoň 1 požadavek pro každý UC.

6.2.1 Požadavky na software

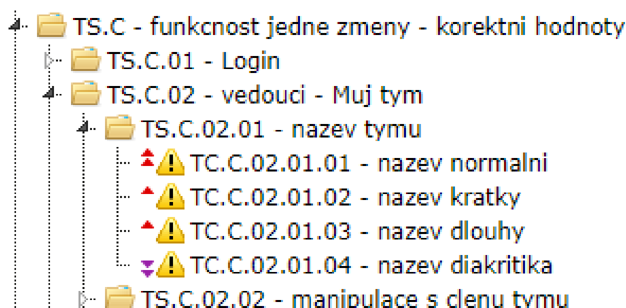
Požadavky jsou hierarchicky uspořádány vzhledem k stránce a činnosti, která se na stránce provádí. Jednotlivé požadavky jsou označeny jako RQM a skupiny požadavků jsou označeny jako RQS (requirement suite). Každý požadavek má jednotný identifikátor, který obsahuje RQM nebo RQS a dále číslování, které zajistí unikátnost daného požadavku, či skupiny požadavků. Zároveň každý požadavek obsahuje informaci, který případ užití provádí.



Obrázek 6.1: Struktura požadavků v nástroji SquashTM

6.2.2 Testovací případy

K výše popsaným požadavkům jsou následně vytvořeny jednotlivé testovací případy, které jsou uspořádány rovněž hierarchicky stejným způsobem. Jednotlivé testovací případy mají tříúrovňovou prioritu a jsou označovány jako TC a skupiny testovacích případů jsou označovány jako TS (test suite). Dále v obou případech následuje za pomlčkou poznámka pro lepší přehlednost. Následuje obrázek 6.2, kde je struktura testovacích případů ukázána.



Obrázek 6.2: Struktura požadavků v nástroji SquashTM

6.3 Návrhový vzor PageObject

Pro realizaci ovládání webových stránek při funkcionálních a end-to-end testech byl využit návrhový vzor PageObject. Při využití tohoto návrhového vzoru, je vhodné definovat předka, který volá statickou metodu `PageFactory.initElements()` s parametry aktuálně načteného driveru a instancí třídy pro vyhledávání elementů. Metoda vyhledá všechny atributy pageobjectu označené anotací `@FindBy` a nad nimi zavolá metodu `findElements()`, která je v Seleniu dostupná nad každým prvkem označeným touto anotací. Tato metoda vrátí seznam s hledaným elementem, nebo prázdný seznam. Tento přístup umožňuje soustředit veškeré ovládání dané webové stránky do jedné třídy. Pokud by byla stránka ovládána bez návrhového vzoru PageObject, vzniknl by z programátorského hlediska výrazně méně čitelný kód a opakující se kód [14].

6.3.1 Příklad využití

Na fiktivní stránce `ExamplePage` existuje textové pole s identifikátorem (ID) `examplePage-wordT` a tlačítko s identifikátorem (ID) `examplePage-button`. Cíl je vyplnit textové pole textem a kliknout na tlačítko pomocí příkazů Selenium WebDriver. V následujících ukázkách je předpokládáno, že správně

proběhla konfigurace, driver je inicializovaný a URL stránky se nachází na výše zmíněné ExamplePage.

Ukázka akce bez PageObjectu

```
// vyplneni textoveho pole
List<WebElement> textFieldList =
         driver.findElements(By.id("examplePage-wordT "));
if (passwordList.isEmpty()) {
  Fail.failTestDueToMissingElement("examplePage-wordT ");
}
WebElement wordT = textFieldList.get(0);
wordT.clear();
wordT.sendKeys("random text");

// klik na tlacitko
List<WebElement> buttonList =
         driver.findElements(By.id("examplePage-button"));
if (buttonList.isEmpty()) {
  Fail.failTestDueToMissingElement("examplePage-button");

}
WebElement button = buttonList.get(0);
button.click();
```

Ukázka kódu 6: Ukázka akce bez využití návrhového vzoru PageObject

Ukázka akce s využitím PageObject

Nejprve je nutno definovat pageobject pro výše popsanou stránku a následně provést akci, což je ukázáno na ukázce kódu 7.

```
public class Example_Page extends PageObject {
  // ID's on web page
  public static final String WORD_T = "examplePage-wordT";
  public static final String BUTTON = "examplePage-button";

  @FindBy(id = WORD_T)
  private List<WebElement> wordT;

  public void setWord(String word) {
```

```
10     // getElement obsahuje nalezeni elementu v seznamu
11     WebElement element = getElement(wordT, WORD_T);
12     element.clear();
13     element.sendKeys(word);
14   }
15 
16   @FindBy(id = BUTTON)
17   private List<WebElement> buttonBTN;
18   public PageObject clickOnButton() {
19    getElement(buttonBTN, BUTTON).click();
20    return new PageObject();
21   }
22 }
23 
24 //Provedeni akce
25 Example_Page ep = new Example_Page();
26 ep.setWord("nahodne slovo")
27 ep.clickOnButton();
```

Ukázka kódu 7: Ukázka akce s využítím návrhového vzoru PageObject

6.3.2 Výhody při použití návrhového vzoru PageObject

Pokud je použit návrhový vzor PageObject, dojde k razantnímu zefektivnění modulu pro ovládání stránky. Pokud by v aplikaci existovala nějaká stránka s takovými elementy, jako např výše popsaná `Example_Page`, určitě by byla zahrnuta ve více testech. Vytvoření návrhového vzoru PageObject výrazně zpřehlední další ovládání této stránky.

6.4 Funkcionální testy

Funkcionální testy jsou v aplikaci realizovány tzv. *one change* testy, které provedou jednu změnu a poté zkontrolují místa v aplikaci, které byly touto změnou ovlivněny.

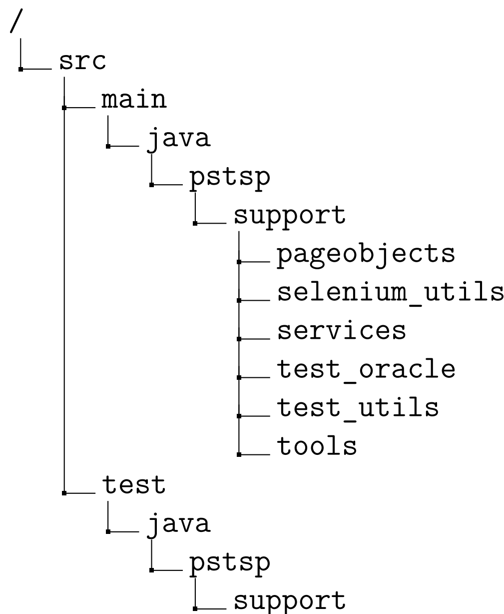
6.4.1 Struktura testovacího projektu

Projekt, který byl vytvořen pro automatizované testy je složen z dvou podprojektů, které nesou názvy `PSTSP-support` (dále jen support), knihovna,

která zajišťuje ovládání webové aplikace, a `PSTSP-functional-tests` (dále jen functional), balík obsahující testy. Toto rozdělení bylo zavedeno z důvodu předpokladu, že na vytváření testů může v budoucnu spolupracovat více lidí. Také je odděleno ovládání aplikace od testů, což ústí ve větší přehlednost. Propojení mezi těmito dvěma projekty je zajištěno pomocí nástroje Maven, což je nástroj pro automatizaci buildu aplikací, nejvíce podporovaný pro jazyk Java. Kostra pro tyto projekty byla poskytnuta vedoucím práce.

Projekt support

Jak již bylo řečeno, projekt support zajišťuje ovládání PSTSP pomocí Selenium WebDriver. Konkrétní stránky jsou reprezentovány PageObjecty. Veškeré akce se stránkou jsou v tomto balíku náležitě otestovány. Jedná se o jednotkové testy samotných PageObjectů, které jsou v balíku `test`. Tyto testy nebyly dosud dříve zmiňovány. Dále jsou zde implementované třídy, které logují celý proces testování pro pozdější využití ve statistikách, nebo zavádějí driver webového prohlížeče jen jednou do paměti, jelikož se jedná o časově náročný proces. Tato práce s driverem je doporučována i v dokumentaci [14]. Nastavení celého projektu lze konfigurovat pomocí textového souboru. Obsahuje například cesty k ovladačům webových prohlížečů potřebné pro Selenium WebDriver. Následuje ukázka struktury projektu.



V následujícím seznamu jsou popsány jednotlivé balíky z výše zmíněné adresářové struktury.

- `basic` – třídy pro načtení konfigurace, pro navigaci URL adres a konstant, které se v projektu vyskytují.
- `pageobjects` – veškeré pageobjecty k PSTSP, což umožní její následné

ovládání v `services`

- `selenium_utils` – třídy pro práci s webovou stránkou, například kliknutí a čekání na určitý element, práce s tabulkami, javascript alerty, atd
- `services` – třídy, které využívají balík pageobjects a vykonávají na stránce specifické akce, které obsahují typické kroky nějakého testovacího případu tak, jak jsou napsané v SquashTM. Tyto metody se dále používají testech.
- `test_oracle` – určuje tzv. orákulum, pro vyhodnocení testů. V projektu je využito například pro kontroly načítání konfigurací, seznamu studentů apod. Samotné orákulum používané v testech tvoří parametry metod, které jsou v balíku `services`.
- `test_utils` – funkcionalita pro správu chodu testů. Definice tagů testů. Sofistikované metody pro kontrolu, při kterých se počítají jednotlivé asserty, jejichž součet se vypíše po průběhu testů. Dále například třídy pro report při selhání testu, či pro zajištění logování průběhu testů a jednotného zavedení selenium driveru do paměti.
- `tools` – logování, či samotné vytváření driveru.

Projekt functional-tests

Tento balík využívá akce, které jsou za pomocí PageObject definovány v projektu `support`, v balíku `services`. Jak je výše popsáno, metody vykonávají kroky, které jsou zapotřebí k vykonání testovacího případu společně s kontrolou právě vykonané akce na místech v aplikaci, kde se daná akce provedla. Na ukázce kódu 8 je ukázán funkcionální test, který provede přidá člena do týmu a zkontroluje, zda se změna opravdu provedla.

```
1  @ExtendWith(TestSetting.class)
2  @ExtendWith(TestEvaluation.class)
3  @Tag(PstspTags.ONE_CHANGE)
4  @Tag(PstspTags.ACTIVE)
5  @Tag(PstspTags.LEADER)
6  @TestMethodOrder(MethodOrderer.OrderAnnotation.class)
7  @DisplayName("TS.C.02.02: Edit team members")
8  public class TS_C_02_02 {
9
10
```

```java
    @BeforeAll
    public static void setup() {
        Home_Actions.resetDB();
    }
    @AfterAll
    public static void tearDownAfterAll() {
        Login_Actions.logoutIfPossible();
    }

    @Order(1)
    @Test
    @Tag(PstspTags.IMP_CRITICAL)
    @DisplayName("TC.C.02.02.01: add one member")
    void test_1() {
        Set<String> expected = new HashSet<>();
        expected.add("Jakub Dohnal (A19B1234P)");
        Team_Actions.editTeamMembersAsLeaderSuccess(expected,"
   leader");
    }
}
```

Ukázka kódu 8: Ukázka funkcionálního testu

V tomto testu je nejprve připraveno testovací prostředí uvedením databáze do stavu, které testy předpokládají, pomocí akce, která v testovací verzi aplikace klikne na tlačítko, které tuto akci provede. Dále je test rozšířen anotací `@ExtendWith` o třídy, které se starají o nastavení driveru a logování průběhu testů. Všechny testy jsou náležitě otagovány pomocí anotace `@Tag`, což umožní seskupování testů, tj. spouštět testy, které mají stejný tag. Nakonec je u testu anotace `@DisplayName`, která identifikuje test tak, jak je popsán v SquashTM. Testy drží stejnou adresářovou strukturu, jako v SquashTM. Pokud bychom chtěli provést tento test s jinými parametry, tak tato metoda bude pouze zavolána znovu s jinými parametry. Toto může být rovněž považováno za výhody použití návrhového vzoru PageObject, jelikož tyto metody (akce) PageObjecty využívají.

6.4.2 Výsledky testování

Testovací případy pro funkcionální testy byly tvořeny z požadavků, jejichž pokrytí navrženými testy dosahovalo 100%. Jednotlivé požadavky byly systematicky extrahovány ze všech případů užití, takže lze zmínit, že funkcio-

nální testy dosahují 100% pokrytí z hlediska případů užití. V aktuální verzi PSTSP 0.9.0 se nenachází test, který by selhal. Výsledky funkcionálního testování byly zanášeny do SquashTM.

6.5 End-to-end testy

Pro end-to-end testy jsou v aplikaci vytipovány jednotlivé entity, které lze sledovat v rámci aplikačního cyklu. Jednotlivé posloupnosti akcí, které souvisí s vytipovanými entitami lze skládat z již definovaných akcí, které byly vytvořeny pro funkcionální testy. Sledované entity jsou: tým, schůzka, zadání, mentor.

6.5.1 Příklad end-to-end testu

Vedoucí nově vytvořeného týmu se přihlásí, přidá členy do týmu, tým označí jako kompletní a projeví zájem o téma. Nebo například Garant vytvoří tým, následně ho upraví a označí jako nekompletní. Pro vytvoření této sekvence akcí jsou využity metody z projektu `PSTSP-support` z balíku `services`.

6.5.2 Výsledky testování

V aktuální verzi PSTSP 0.9.0 se nenachází E2E test, který by selhal. Výsledky E2E testů jsou řešeny stejně, jako výsledky funkcionálních testů a jsou zanášeny do SquashTM.

7 Začlenění testovacího procesu do plánu vývoje aplikace

Jelikož byly testy vyvíjeny souběžně s aplikací, tak testování aplikace probíhalo vždy po vývoji specifické části aplikace, vyjma jednotkových testů, které byly vytvářeny při vývoji jednotlivých částí aplikace. Pro zaznamenávání defektů byl na začátku vývoje zřízen software MantisBT, avšak kvůli zanedbatelnému množství nalezených defektů funkčního charakteru nebyl tento software využíván. Později se přešlo na nástroj integrovaný do verzovacího systému GitLab, který umožňuje spravovat jednotlivé problémy (Issues), a přiřazovat je jednotlivým vývojářům k opravení. Informace o nalezených selháních byly předávány tímto způsobem, případně pomocí jiných komunikačních nástrojů.

7.1 Výsledky průběžného testování

Aplikace byla testována v několika iteracích dle pořadí, které je určeno testovacím plánem v bodě 5.2.9. U daného modulu testy probíhaly vždy ve dvou, či více iteracích, pokud se selhání testů opakovalo. Pro uchovávání výsledků testů byl zvolen SquashTM. Squash TM nabízí vytváření tzv kampaní, ve kterých se definují veškeré TC, které se mají spouštět. Dále se tyto TC spouští v tzv iteracích. Výhodou je uchování veškerý iterací (jednotlivých spouštění testů), které byly na projektu kdy provedeny. Díky tomu je možné na obrázcích dále, že se pomocí testování skutečně zlepšovala kvalita aplikace a zároveň byly splněné cíle testování definované v testovacím plánu v bodě 5.2.4

7.1.1 Testování ve vývojovém cyklu

V průběhu vývoje (vždy po dokončení části aplikace) byla aplikace testována v celkem 16 významných iteracích (spouštění testů během vývoje jednotlivých modulů zde do statistik není uvažováno). Na následujícím obrázku jsou zmíněné iterace zobrazeny i s úspěšností testů.

Iteration	Ready	Running	Passed	Failure	Blocked	Untestable
V.0 - Neprihlaseny C - 1	0	0	6	0	0	0
V.0.1 - Neprihlaseny C- 2	0	0	6	0	0	0
V.1 - Neprihlaseny D - 1	0	0	6	0	0	0
V.1.1 - Neprihlaseny D - 2	0	0	6	0	0	0
V.2 - Vedouci C - 1	0	0	86	0	0	0
V.2.1 - Vedouci C - 2	0	0	86	0	0	0
V.3 - Vedouci D - 1	0	0	8	0	0	0
V.3.1 - Vedouci D - 2	0	0	8	0	0	0
V.4 - Mentor C - 1	0	0	13	14	0	0
V.4.1 - Mentor C - 2	0	0	27	0	0	0
V.4.2 - Mentor C - 3	0	0	27	0	0	0
V.5 - Garant C - 1	0	0	80	66	0	0
V.5.1 - Garant C - 2	0	0	146	0	0	0
V.5.2 - Garant C - 3	0	0	146	0	0	0
V.6 - E2E testy	0	0	9	0	0	0
V.7 - Vsechny testy	0	0	291	0	0	0
Total	0	0	951	80	0	0

Obrázek 7.1: Zobrazení úspěšnosti jednotlivých iterací ve vývojovém cyklu

7.1.2 Testování v finální fázi vývojového cyklu

Testování verze 0.9.0 probíhalo v 6 iteracích, kde byly spuštěny testy pro každý modul a následně všechny testy. Na následujícím obrázku jsou zobrazeny dříve zmíněné iterace.

Iteration	Ready	Running	Passed	Failure	Blocked	Untestable
F.1 - Neprihlaseny	0	0	12	0	0	0
F.2 - Vedouci	0	0	94	0	0	0
F.3 - Mentor	0	0	28	0	0	0
F.4 - Garant	0	0	146	0	0	0
F.5 - E2E	0	0	9	0	0	0
F.6 - Vsechny testy	0	0	291	0	0	0
Total	0	0	580	0	0	0

Obrázek 7.2: Zobrazení úspěšnosti jednotlivých iterací po vývoji aplikace

8 Závěr

V bakalářské práci byla automatizovanými testy důkladně otestována souběžně vyvíjená webová aplikace. Pro vývoj aplikace byl využit framework Nette, takže jednotkové testy byly psány v PHP. Pro funkcionální a end-to-end testy byla použita Java společně orkem Selenium 4 a frameworkem JUnit 5.

Aplikace je středního rozsahu a svojí komplexností poměrně výrazně přesáhla původní odhady, což se samozřejmě odrazilo i v rozsahu testování. Pro jednotkové testy bylo vytvořeno 43 testů, o celkové velikosti 18,3 KB zdrojového kódu. Pro funkcionální testy a end-to-end testy byla na základě dřívějších zkušeností nejdříve vytvořena knihovna Support, využívající Seleniový návrhový vzor PageObject. Tato knihovna slouží pro ovládání webových strámek aplikace a tedy jako přímá podpora pro psaní testů. Díky své modularitě a robustnímu návrhu může v budoucnu sloužit i pro jiný typ automatizovaných testů, např. testy akceptační. Rozsah knihovny je 132 tříd celkové velikosti 572 KB zdrojového kódu. Knihovna Support byla během svého vývoje průběžně samostatně otestována vlastními jednotkovými testy. Jejich počet je 275.

Vlastní testování probíhalo na základě připraveného testovacího plánu za podpory dalších prostředků. Jako jeden z hlavních prostředků byl zvolen systém SquashTM, ve kterém byly zpracovány seznamy všech požadavků a na ně navazující testovací případy. Pomocí SquashTM byly taktéž řízeny následné testovací kampaně a jejich výsledky byly do SquashTM automaticky zpětně zaznamenávány.

Pro vlastní automatizované testy bylo vytvořeno 549 automatizovaných testů, o celkové velikosti 201 KB zdrojového kódu. Tyto testy byly namapovány na testovací případy ve SquashTM, jejichž počet byl 291, jelikož některé testy byly parametrizované a tudíž se do SquashTM zanesla pouze jedna iterace testu.

Díky existenci podrobných případů užití (vytvořených v jiné BP) bylo možné nepřímo stanovit jejich pokrytí. Z UC byly systematicky extrahovány jednotlivé požadavky, jejichž pokrytí automatizovanými testy je 100%. Je tedy možné nepřímo prokázat, že automatizovanými testy bylo pokryto i 100 % případů užití. Vizualizace pokrytí požadavků jsou součástí přílohy.

Tato BP opět potvrdila, že komplexní testování je svojí náročností plně srovnatelné s vlastním vývojem.

Provedené testy poskytují uspokojivou míru jistoty, že vyvinutá apli-

kace svojí realizovanou funkcionalitou splňuje všechny požadavky na nasazení v předmětu KIV/TSP1.

Dosud byla aplikace testována na vývojovém prostředí. Po nasazení aplikace na katedrální server, které je plánováno v období srpen až září 2022, opět proběhnou všechny její automatizované testy. Díky tomu lze oprávněně očekávat, že již v zimním semestru akademického roku 2022/23 budou moci aplikaci využívat vyučující i studenti předmětu KIV/TSP1.

Literatura

[1] ARORA, G. *Solid Principles Succinctly.* CreateSpace Independent Publishing Platform, 2017. ISBN 9781542809511.

[2] BERGMANN, S. *PHPUnit Manual — PHPUnit 9.5 Manual* [online]. Sebastian Bergmann, 2022. [cit. 2022/03/31]. Dostupné z: `https://phpunit.readthedocs.io/en/9.5/`.

[3] BUREŠ, M. et al. *Efektivní testování softwaru: Klíčové Otázky pro Efektivitu Testovacího Procesu.* Grada, 2016. ISBN 978-80-247-5594-6.

[4] FOWLER, M. *Continuous integration* [online]. Martin Fowler, 2006. [cit. 2022/04/03]. Dostupné z: `https://www.martinfowler.com/articles/continuousIntegration.html`.

[5] *Logging - OWASP Cheat Sheet Series* [online]. OWASP, 2021. [cit. 2022/02/25]. Dostupné z: `https://cheatsheetseries.owasp.org/cheatsheets/Logging_Cheat_Sheet.html`.

[6] *PHP Master | Logging with PSR-3 to Improve Reusability* [online]. Patrick Mulvey, 2013. [cit. 2022/02/26]. Dostupné z: `https://www.sitepoint.com/logging-with-psr-3-to-improve-reusability/`.

[7] *monolog/01-usage.md* [online]. Jordi Boggiano, 2021. [cit. 2022/02/26]. Dostupné z: `https://github.com/Seldaek/monolog/blob/main/doc/01-usage.md`.

[8] *Nette Tester* [online]. Nette Foundation, 2022. [cit. 2022/04/01]. Dostupné z: `https://tester.nette.org/cs/guide`.

[9] OLAN, M. Unit Testing: Test Early, Test Often. *J. Comput. Sci. Coll.* dec 2003, 19, 2, s. 319–328. ISSN 1937-4771.

[10] PATTON, R. *Testování softwaru.* Computer Press, 2002. ISBN 80-7226-636-5.

[11] ROUDENSKÝ, P. – HAVLÍČKOVÁ, A. *Řízení kvality softwaru: Průvodce testováním.* Computer Press, 2013. ISBN 978-80-2513-816-8.

[12] SHAHIN, M. – ALI BABAR, M. – ZHU, L. Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. *IEEE Access.* 2017, 5, s. 3909–3943. doi: 10.1109/ACCESS.2017.2685629.

[13] *Testing (Symfony Docs)* [online]. Symfony™, 2021. [cit. 2022/03/31]. Dostupné z: `https://symfony.com/doc/current/testing.html`.

[14] *The Selenium Browser Automation Project* [online]. Software Freedom Conservancy, 2022. [cit. 2022/04/15]. Dostupné z: `https://www.selenium.dev/documentation/`.

[15] *Konfigurace Tracy | Tracy Ladicí Nástroj* [online]. Nette Foundation, 2022. [cit. 2022/02/25]. Dostupné z: `https://tracy.nette.org/cs/configuring`.

[16] *Tracy*
ILogger | Tracy API [online]. Nette Foundation, 2022. [cit. 2022/02/25]. Dostupné z: `https://api.nette.org/tracy/master/Tracy/ILogger.html`.

[17] *Travis CI User Documentation* [online]. TRAVIS CI, GMBH, 2022. [cit. 2022/04/03]. Dostupné z: `https://docs.travis-ci.com`.

Seznam zkratek

API Application Programming Interface.

CD Continuous Deployment.

CI Continuous Integration.

CSS Cascading Style Sheets.

DOM Document Object Model.

E2E End-to-End.

HTML Hypertext Markup Language.

ID Identifier.

IEEE Institute of Electrical and Electronics Engineers.

IP Internet Protocol.

ISO International Organization for Standardization.

ISTQB International Software Testing Qualifications Board.

JSON JavaScript Object Notation.

KIV Katedra informatiky a výpočetní techniky.

NEON Nette Object Notation.

OKS Ověřování kvality software.

PHP Hypertext Preprocessor.

PSTSP Podpůrný software TSP.

RFC Request for Comments.

RQM Requirement.

RQS Requirement Suite.

SQL Structured Query Language.

TC Test Case.

TMap Test Management Approach.

TS Test Suite.

TSP Týmový softwarový projekt.

UC Use Case.

URI Uniform Resource Identifier.

URL Uniform Resource Locator.

VM Virtual Machine.

XML Extensible Markup Language.

XPath XML Path Language.

YAML Ain't Markup Language.

A Spouštění testů

Navržené testy je třeba nějak spustit, aby se skutečně ověřila funkčnost aplikace. Veškeré automatizované testy lze spouštět z příkazové řádky. Některé testy lze spouštět dokonce přímo ve vývojovém prostředí.

A.1 Spouštění jednotkových testů

Jednotkové testy byly realizovány pomocí PHP frameworku Nette Tester a mohou se spouštět jednotlivě, jako samostatné PHP skripty. Druhou možností je využít spouštěč testů, který je součástí Nette Testeru. Ten spustí veškeré jednotkové testy v aplikaci (soubory s koncovkou `.phpt`, nebo s názvem `*Test.php`) a zároveň vyhodnotí veškeré nastavení, které je u jednotlivých testů provedeno například pomocí anotací [8]. Veškeré spouštění tedy probíhá příkazem: `.\vendor\bin\tester .` , spouštěným v kořenovém adresáři aplikace. Pro spuštění a vygenerování reportu pokrytí se použije stejný příkaz, pouze s parametry:
`--coverage .\temp\cov.html -c .\tests\php.ini`, kde první parametr udává, kam se má report o pokrytí uložit a druhý nastavuje cestu pro `php.ini` s nakonfigurovaným rozšířením Xdebug.

A.2 Spouštění funkcionálních a E2E testů

Funkcionální a E2E testy lze spouštět z vývojového prostředí a z příkazové řádky. Spouštění z vývojového prostředí zde nebude dále detailněji popisováno. V IntelliJ Idea se testy spouští kliknutím pravým tlačítkem myši na balík testů a zvolení možnosti *Run tests in „balíktestů“*.

A.3 Spouštění testů z příkazové řádky

Nejprve je nutno v projektu `PSTSP-support`, v adresáři, kde se nachází soubor `pom.xml`, spustit příkaz `mvn install`, což nainstaluje ovládání webové stránky z projektu `PSTSP-support` do projektu `PSTSP-functional-tests`. Poté je nutno se přesunout do projektu `PSTSP-functional-tests` a vytvořit konfigurační soubor. K dispozici je ukázkový konfigurační soubor `configurations.txt.default`, který je umístěn v kořenovém adresáři projektu. Tento

soubor slouží pro konfiguraci projektu PSTSP-support, který slouží k ovládání webové stránky. Nachází se v něm například cesty k jednotlivým driverům webových prohlížečů, nastavení timeoutu pro čekání na prvek na stránce, apod. Tento soubor musí být umístěn v kořenovém adresáři projektu PSTSP-functional-tests, nebo k němu musí být zadaná cesta, která je určena VM parametrem -Dpstsp.configuration.uri=cesta/k/konfiguraci.

Tímto způsobem lze přepsat i konfigurace. Například typ spuštěného prohlížeče lze upravit pomocí -DWebBrowserType=type. Všechny testy lze spouštět pomocí třídy RunAnyTest, která se nachází v balíku run_tests v projektu PSTSP-functional-tests.

Třídu lze spouštět s povinnými parametry p, který udává celé jméno balíku, nebo c, který udává celé jméno testovací třídy, která má být spuštěna. Dále obsahuje volitelný parametr i, který určuje jaké testy, dle jejich tagů má spouštěč zahrnout do testovací iterace a dále volitelný parametr e, který určuje, jaké testy má spouštěč z iterace vyloučit. Použité tagy pro testy jsou následující.

- ACTIVE – všechny typy testů (C, D, E), typy testů popsány v kapitole 5.2.5
- ONE_CHANGE – všechny testy typu C
- NEGATIVE – všechny testy typu D
- END_TO_END – všechny E2E testy
- BOUNDARY – typ C a D pro testy hraničních hodnot
- MENTOR – všechny testy akcí mentora
- GARANT – všechny testy akcí garanta
- LEADER – všechny testy akcí vedoucího
- FAIL – test, který vždy selže (pro účely debuggingu)
- UNLOGGED – všechny testy akcí nepřihlášeného uživatele
- CRITICAL – testy velmi vysoké priority
- MAJOR – testy vysoké priority
- MINOR – testy nízké priority

Třídu lze spouštět například takto:
`RunAnyTest p pstsp.functional_tests i ACTIVE e GARANT`, což zajistí spuštění všech `ACTIVE` testů vyjma testů akcí garanta. Adresářová struktura projektu je odvozená ze struktury testovacích případů v SquashTM.

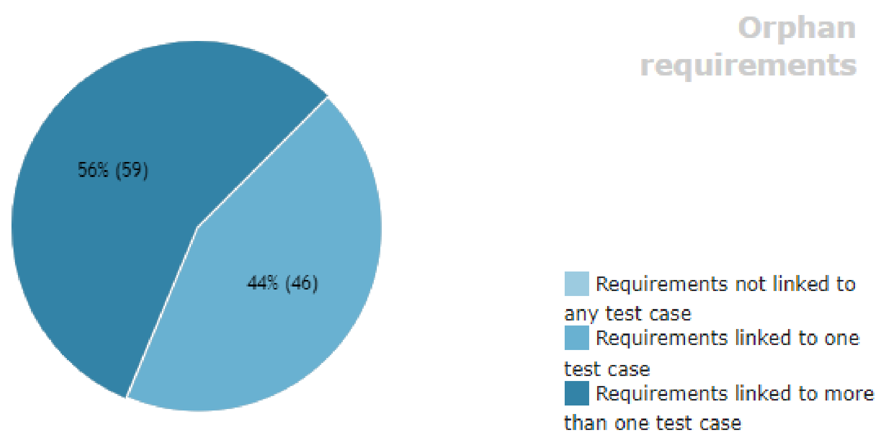
B Statistiky k vytvořeným testům

Díky použití SquashTM jsou k dispozici generované statistiky, které budou níže interpretovány.

B.1 Požadavky

Z celkem 59 UC bylo systematicky vytvořeno 105 RQM. Informace o požadavcích budou níže interpretovány.

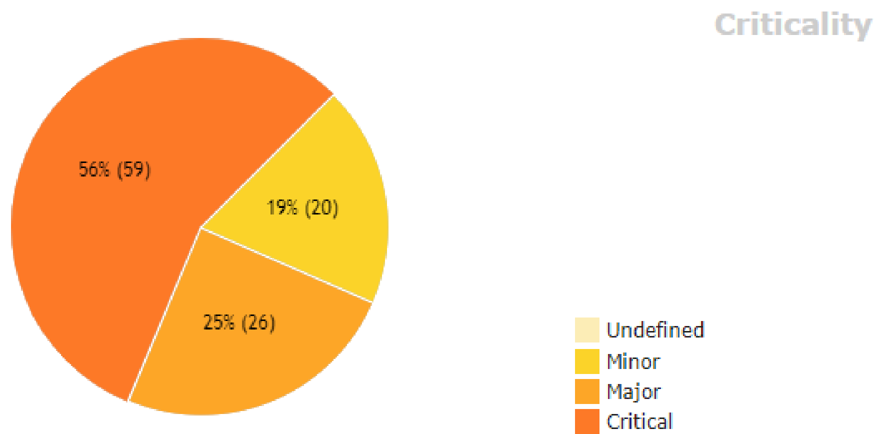
B.1.1 Provázání RQM a TC



Obrázek B.1: Provázání RQM a TC

Z grafu B.1 je vidět, že 56% RQM je ověřována více než jedním TC. To je způsobeno zejména četným zastoupení parametrizovaných testů, kdy je test, který se ve skutečnosti opakuje například 20x ve SquashTM zaznamenán pouze jedním TC.

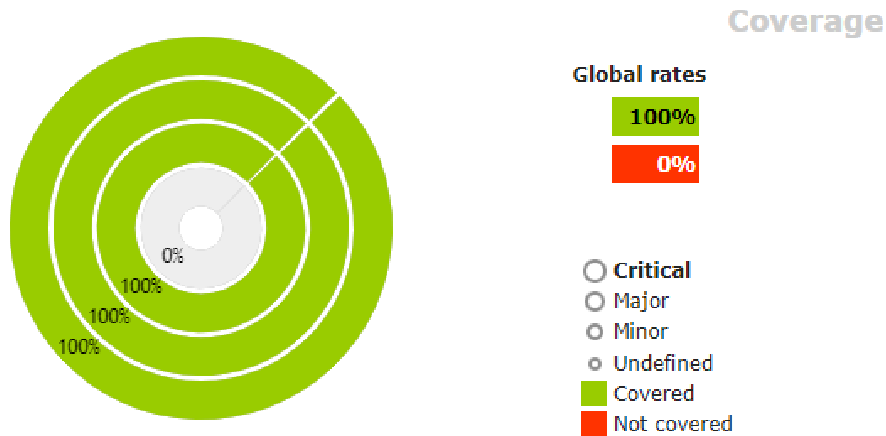
B.1.2 Priorita RQM



Obrázek B.2: Vizualizace priorit RQM

Z grafu B.2 je vidět, že se v testovacím plánu vyskytuje nadpoloviční množství požadavků s velmi vysokou prioritou (*critical*). Tyto požadavky by měly při selhání velký dopad na celý systém a způsobily by jeho nepoužitelnost.

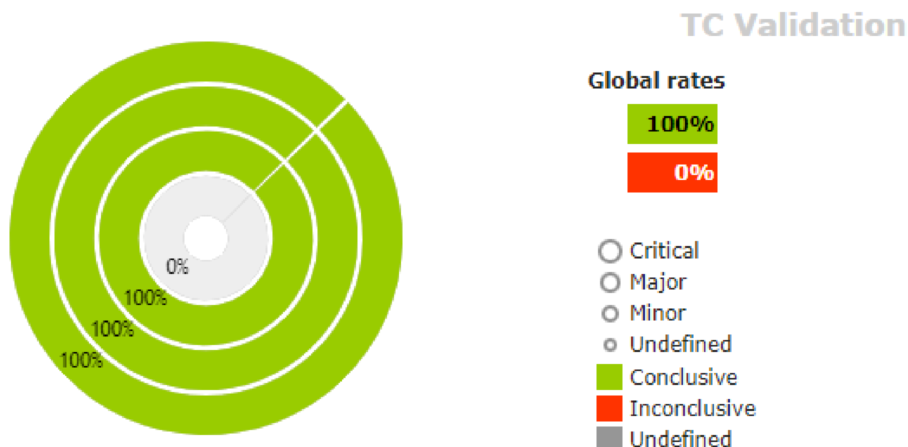
B.1.3 Pokrytí priorit RQM pomocí TC



Obrázek B.3: Vizualizace pokrytí RQM

Z vizualizace B.3 je vidět, že všechny priority požadavků jsou 100% pokryty testy. Detailnější rozpis pokrytí je součástí elektronické přílohy.

B.1.4 Globální úspěšnost TC dle priorit RQM



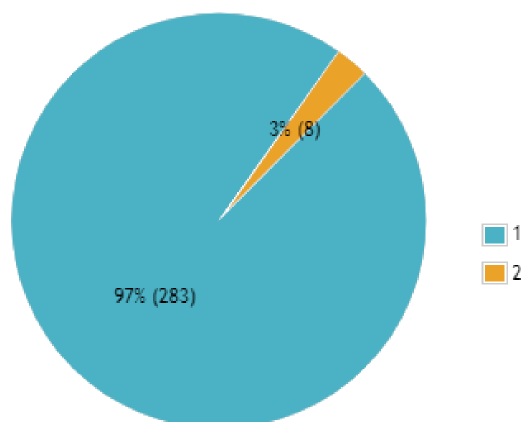
Obrázek B.4: Vizualizace pokrytí RQM

Z vizualizace B.4 je vidět, že v poslední verzi aplikace 0.9.0 je globální úspěšnost testů 100%. Pokud tedy vezmeme v potaz předchozí vizualizaci B.3, tak jsou pokryty všechny priority požadavků s úspěšností testů 100%.

B.2 Testovací případy

Celkový počet 105 RQM je pokryt 291 TC.

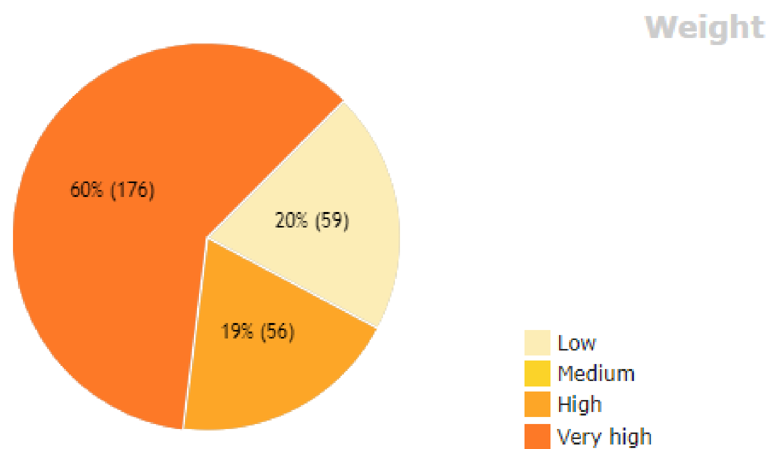
B.2.1 Provázání TC a RQM



Obrázek B.5: Počet provázaných RQM na jednotlivé TC

Z vizualizace B.5 je vidět, že 97% TC je provázáno s jedním požadavkem a 3% TC jsou provázány s dvěma požadavky.

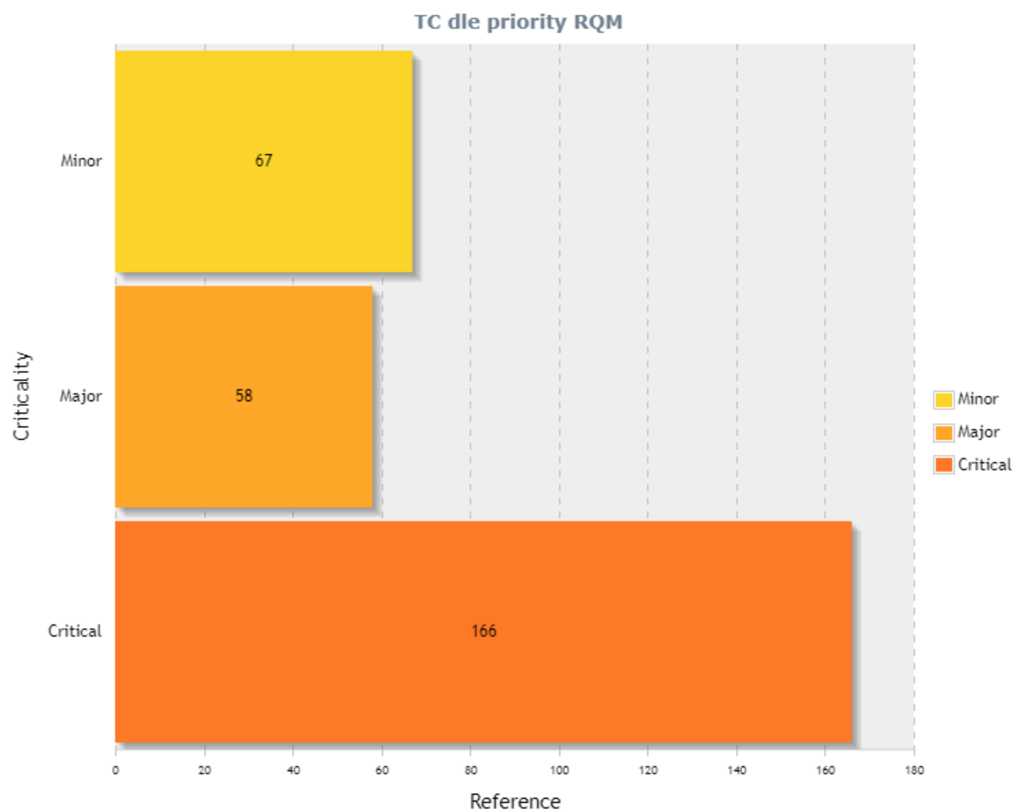
B.2.2 Priorita testovacích případů



Obrázek B.6: Rozložení priority testovacích případů

Jak je vidět z vizualizace, testovací případy byly pro jednoduchost označovány pouze třemi prioritami. Nízká, vysoká a velmi vysoká. Nejvyšší zastoupení v testech mají testy s nejvyšší prioritou. To může být způsobeno tím, že byly vytvářeny přednostně. Další důvod může být ten, že vytvořeno největší množství požadavků s největší prioritou, jak je vidět na obrázku B.2.

B.2.3 Počet testovacích případů na jednotlivé priority požadavků



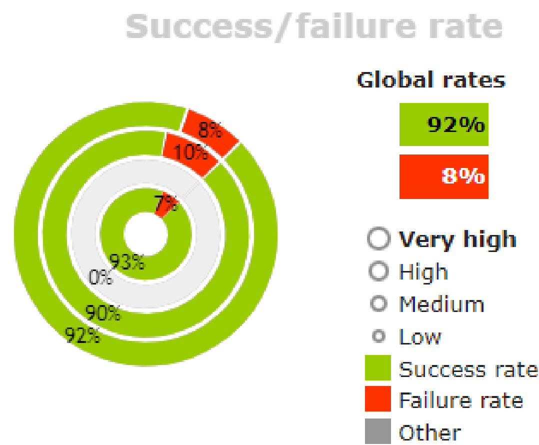
Obrázek B.7: Počet testovacích případů na jednotlivé priority požadavků

Na grafu B.7 je vidět, že existuje nejvíce testovacích případů ověřující nejvyšší prioritu požadavků.

B.3 Testovací iterace

Testy byly spouštěny ve dvou kampaních. Během a po završení vývoje.

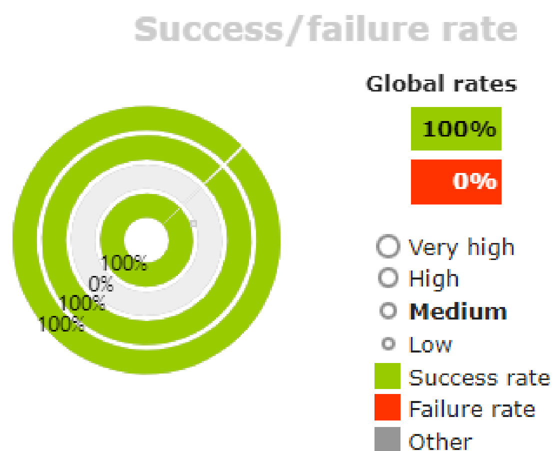
B.3.1 Testy během vývoje



Obrázek B.8: Míra selhání testů během vývoje

Testy během vývoje selhávaly poměrně zřídka. Z vizualizace B.8 je vidět, že nejčastěji selhávaly testy s vysokou prioritou.

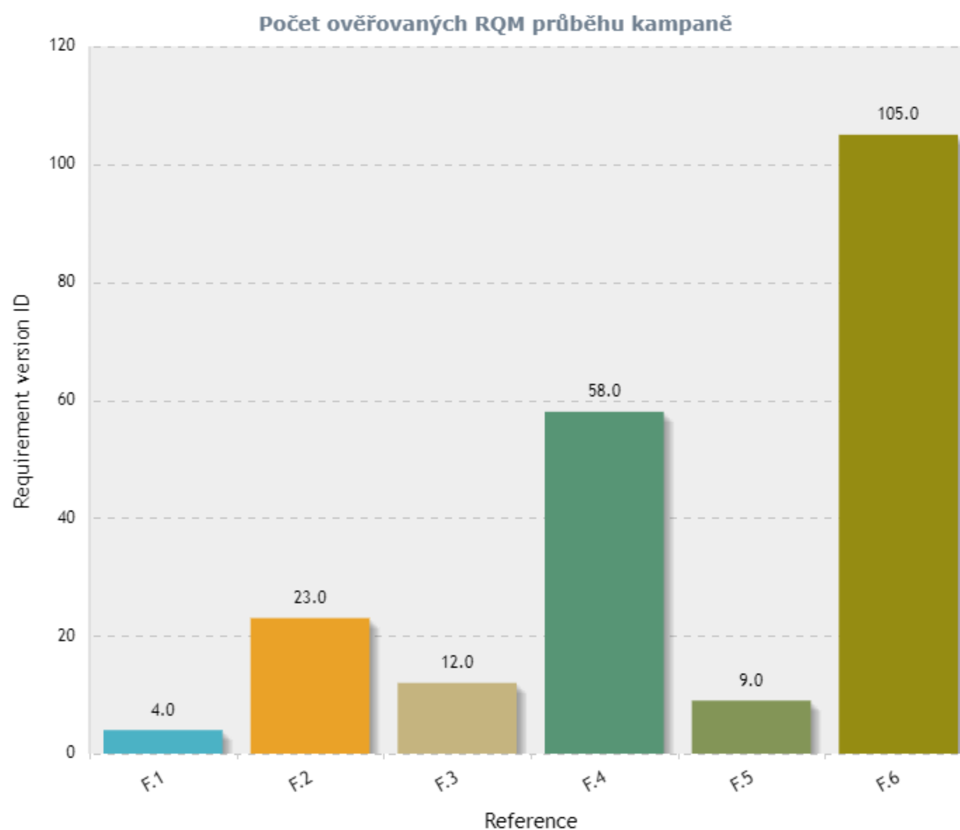
B.3.2 Testy poslední verze



Obrázek B.9: Míra selhání testů po vývoji

Jak je vidět z vizualizace B.9, testy po završení vývoje aplikace nejeví známky selhání.

B.3.3 Počet ověřovaných RQM během testů poslední verze aplikace



Obrázek B.10: Četnost ověřovaných RQM v iteracích kampaně

Na grafu B.10 je vidět postupně měnící se počet ověřovaných požadavků v průběhu kampaně pro ověření aplikace ve verzi 0.9.0. Hodnoty na ose X, tj. hodnoty F.1 - F.6 označují identifikátory jednotlivých testovacích iterací. Stejné iterace lze vidět na obrázku 7.2.

C Obsah elektronické přílohy

C.1 Adresářová struktura

- /prilohy_bp
  - Aplikace_a_knihovny
    - PSTSP-support – Projekt pro ovládání webové stránky
    - PSTSP-functional-tests – Projekt pro funkcionální testy
  - Text_prace
    - tex – zdrojové soubory TeXu včetně přiložených obrázků
    - A19B0118P.pdf – Bakalářská práce ve formátu pdf
  - Vysledky
    - SquashTM
      - CPG_F_Testy_verze_0.9.0.csv – export kampaně z poslední verze aplikace
      - CPG_V_Vsechny_testy_v_prubehu_vyvoje.csv – export kampaně z průběhu vývoje
      - PSTSP-RQM.json – požadavky ve formátu JSON pro import do SquashTM
      - PSTSP-TC.json – testovací případy ve formátu JSON pro import do SquashTM
      - Requirement report.pdf – struktura RQM
      - RQM-pokryti.pdf – tabulka pokrytí RQM testovacími případy
      - Test case report.pdf – struktura TC
    - zaklad-pro-testy
      - studenti-test.txt – testovací soubor pro import studentů
      - testDatabase.sql – stav databáze, který je předpokládán u selenium testů
    - pokryti-unit-testy.html – pokrytí třídy Utils.php jednotkovými testy
    - beh-selenim-testu.html – ukázka záznamu o běhu selenium testů generovaným nástrojem IntelliJ Idea
  - readme.txt – popis adresářové struktury