Návrh jednotkových a integračních testů pro otestování WUI Framework

Bc. Hana Malíková

Diplomová práce
2018

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
akademický rok: 2017/2018

ZADÁNÍ DIPLOMOVÉ PRÁCE
(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Hana Malíková**
Osobní číslo:	**A16545**
Studijní program:	**N3902 Inženýrská informatika**
Studijní obor:	**Informační technologie**
Forma studia:	**kombinovaná**
Téma práce:	**Návrh jednotkových a integračních testů pro otestování WUI Frameworku**
Téma anglicky:	**Designing Unit and Integration Tests for Testing the WUI Framework**

Zásady pro vypracování:

1. **Zpracujte rešerši problematiky testování softwaru.**
2. **Proveďte komparaci zvolených nástrojů pro testování softwaru.**
3. **Popište WUI Framework, jeho vlastnosti a možnosti využití v dané oblasti.**
4. **Navrhněte strukturu a metodiku testování.**
5. **Implementujte testy pro ověření WUI Frameworku.**
6. **Zhodnoťte získané výsledky.**

Rozsah diplomové práce:

Rozsah příloh:

Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam odborné literatury:

1. **SPILLNER, Andreas, Tilo LINZ a Hans SCHAEFER. Software Testing Foundations. O'Reilly Media, 2006. ISBN 978-3898643634.**
2. **Wui Framework [online]. 2014 [cit. 2017-11-19]. Dostupné z: https://bitbucket.org/wuiframework/com-wui-framework/wiki/Home.**
3. **BUREŠ, Miroslav, Miroslav RENDA a Michal DOLEŽEL. Efektivní testování softwaru. ISBN 978-80-247-5594-6.**
4. **NATHAN, Rozentals. Mastering TypeScript. Packt Publishing, 2015. ISBN 978-1-78439-966-5.**
5. **STEPHENS, Matt. Testování softwaru řízené návrhem. Computer Press. ISBN 978-80-251-3607-2.**

Vedoucí diplomové práce: **Ing. Petr Žáček**
Ústav informatiky a umělé inteligence

Datum zadání diplomové práce: **1. prosince 2017**

Termín odevzdání diplomové práce: **16. května 2018**

Ve Zlíně dne 11. prosince 2017

doc. Mgr. Milan Adámek, Ph.D.
děkan

prof. Mgr. Roman Jašek, Ph.D.
garant oboru

Prohlašuji, že

- beru na vědomí, že odevzdáním diplomové/bakalářské práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že diplomová/bakalářská práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové/bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně a jeden výtisk bude uložen u vedoucího práce;
- byl/a jsem seznámen/a s tím, že na moji diplomovou/bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – diplomovou/bakalářskou práci nebo poskytnout licenci k jejímu využití jen připouští-li tak licenční smlouva uzavřená mezi mnou a Univerzitou Tomáše Bati ve Zlíně s tím, že vyrovnání případného přiměřeného příspěvku na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše) bude rovněž předmětem této licenční smlouvy;
- beru na vědomí, že pokud bylo k vypracování diplomové/bakalářské práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky diplomové/bakalářské práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem diplomové/bakalářské práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na diplomové/bakalářské práci pracovala samostatně a použitou literaturu jsem citovala. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně, dne Bc. Hana Maláková

…….
podpis diplomanta

ABSTRAKT

V této diplomové práci mám za cíl testování softwaru. Práce vysvětluje některé základní aspekty testování softwaru. To zahrnuje detailní popis úrovní testování. Práce pokračuje komplexním pohledem na nástroje testování. Za účelem vhodného výběru jsou zkoumány některé teoretické vlastnosti nástrojů. Je zde popsán návrh struktury a metodiky testování a vhodných testovacích případů. Testovací případy musí být navrženy tak, aby pokryly všechny aspekty tj. zabezpečení, funkčnost, uživatelské rozhraní.

Klíčová slova: Testování softwaru, Testovací případ, Nástroje testování, Assert

ABSTRACT

In this thesis I aim to test the software. The thesis explains some of the basic aspects of software testing. This includes a detailed description of the testing levels. Work continues with a comprehensive view of the testing tools. Some theoretical properties of the instruments are examined in order to ensure appropriate selection. It describes the design of the structure and methodology of testing and the appropriate test cases. Test cases must be designed to cover all aspects, i.e. Security, functionality, user interface.

Keywords: Software testing, Test case, Testing tools, Assert

Chtěla bych poděkovat Ing. Jakubovi Cieslarovi za poskytnutí cenných rad při tvorbě této práce. Dále můj dík patří mému manželovi za podporu poskytnutou v průběhu celého studia.

Prohlašuji, že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

OBSAH

ÚVOD

V uplynulých letech software zaznamenal obrovské rozšíření. Nezbývají téměř žádné stroje nebo zařízení, které nejsou řízeny softwarem nebo alespoň obsahují software. S touto měrou závislosti zároveň roste potřeba správného fungování těchto aplikací, neboť jejich funkčnost, anebo naopak nefunkčnost, může významně ovlivnit konkurenceschopnost firmy. V rámci odvětví (vestavných a komerčních softwarových systémů) se kvalita softwaru stala nejdůležitějším faktorem úspěchu produktů nebo podniků. Jedním ze způsobů, jak tohoto cíle dosáhnout, je systematické vyhodnocování a testování vyvinutého softwaru, které je ústředním tématem této diplomové práce.

V diplomové práci jsou popsány základní pojmy, které testera provází každý den. Diskutuje se o tom, které testovací aktivity by měly být prováděny během procesu vývoje softwaru. Jsou popsány základní úrovně testů.

Testování softwaru bez vhodné podpory nástrojů je velmi náročné na práci a čas. Druhá kapitola této práce uvádí různé nástroje pro podporu testování a rady pro výběr a implementaci nástrojů.

Dále bude následovat praktičtější pohled na testování, který zahrnuje plánování testů, návrh testovacích případů a různé rady a návody pro testování.

Provádění a vyhodnocení testů. Analýza zajišťuje efektivní vedení testů, zahrnuje měření rozsahu testování a další informace jako je počet testů atd.

Na závěr je popsán celkový dopad testů na projekt.

Diplomová práce je zaměřena na návrh jednotkových a integračních testů pro aplikaci WUI Framework s využitím programovacího jazyka TypeScript. Toto téma jsem zvolila, neboť již několik let pracuji při studiu na pozici testera a součástí mé práce je mimo jiné vytváření a návrh jednotkových a integračních testů. Cílem této práce je přiblížit metodiku a problematiku vývoje testů.

I. TEORETICKÁ ČÁST

1 ROZBOR PROBLEMATIKY TESTOVÁNÍ SOFTWARU

1.1 Obecné zásady testování

Během posledních 40 let bylo přijato několik zásad pro testování jako pravidla testování.

a) **Testování ukazuje přítomnost chyb, nikoli jejich nepřítomnost -** testování může ukázat, že jsou chyby přítomny, ale nemůže dokázat, že v softwaru nejsou žádné chyby. Testování snižuje pravděpodobnost, že v softwaru zůstanou neobjevené chyby, avšak nenalezení žádné chyby stále není důkaz bezchybnosti.
b) **Vyčerpávající testování není možné -** testování všeho (všech kombinací vstupů a vstupních podmínek) není realizovatelné s výjimkou triviálních případů. Namísto vyčerpávajícího testování by měly být k určení hlavního předmětu testovacího úsilí použity analýza rizik a stanovení priorit.
c) **Testovací aktivity by měly začít co nejdříve -** pro včasné nacházení chyb musí testovací aktivity začít v rámci životního cyklu vývoje softwaru nebo systému co nejdříve, jak je to možné, a musí být zaměřeny na definované cíle.
d) **Chyby mají tendenci se shlukovat dohromady -** testování musí být zaměřené proporčně na očekávanou a později zjištěnou hustotu chyb v modulech. Velmi malé množství modulů obvykle obsahuje většinu chyb zjištěných v průběhu testování, před uvolněním, nebo je zodpovědné za nejvíce provozních selhání.
e) **Pokud se stejné testy opakují znovu a znovu, mají tendenci ztrácet účinnost -** jsou-li stále opakovány tytéž testy, časem stejný soubor testovacích případů nenalezne žádné nové chyby. K překonání tohoto „pesticidního paradoxu“ je potřeba existující testovací případy pravidelně revidovat a upravovat. Zároveň je potřeba napsat nové, odlišné testy na vykonávání jiných částí softwaru nebo systému pro případné odhalení dalších chyb.
f) **Test je závislý na kontextu -** testování je vykonáváno odlišně v různých kontextech. Například software kritický z pohledu bezpečnosti se testuje jiným způsobem než webové stránky elektronického obchodu.
g) **Je omylem předpokládat, že žádné chyby, znamená užitečný systém -** nalezení a opravení chyb nepomůže, pokud je vytvořený systém nepoužitelný a nesplňuje potřeby a očekávání uživatelů. [1, s.31, 32]

1.2 Testovací případ

Testovací případ je soubor podmínek nebo proměnných, na základě kterých testující určí, zda testovaný systém splňuje požadavky nebo pracuje správně. Proces vytváření zkušebních případů může také pomoci najít problémy v požadavcích nebo návrhu aplikace.[12]

1.3 Terminologie chyby

Jedna společná definice softwarové chyby je nesoulad mezi programem a jeho specifikací. Jinými slovy, můžeme říci, že v programu se vyskytuje softwarová chyba, když program nedělá to, co očekává jeho koncový uživatel. Například výstup je špatný nebo aplikace zhavaruje. Selhání je nesplnění daného požadavku, rozdíl mezi skutečným výsledkem nebo chováním (definovaným při provádění testu) a očekávaným výsledkem nebo chováním (definovaným ve specifikacích nebo požadavcích). Bug je programátory používaný výraz pro vnitřní chybu, například nesprávně naprogramovaný nebo zapomenutý kód v aplikaci. [1, s.7]

1.4 Modely životního cyklu

Aby bylo možné dosáhnout strukturovaného a kontrolovatelného vývoje softwaru, používají se modely vývoje softwaru a vývojové procesy. Existuje mnoho různých modelů. Příkladem jsou Vodopádový model, V-model, Spirálový model, různé inkrementální nebo evoluční modely a agilní nebo lehké metody jako XP (Extrémní programování), které jsou populární v dnešní době. Pro vývoj objektově orientovaných softwarových systémů se diskutuje o racionálním sjednoceném procesu. [1, s.16]

1.5 Testovací skript

V terminologii standardu IEEE 610 je pojem testovací skript synonymem s pojmem testovací procedura, přičemž testovací procedura je zde definována jako „detailní instrukce pro nastavení, provedení a vyhodnocení výsledků daného testovacího případu“, nebo také jako „dokumentace specifikující posloupnost akcí pro vykonání testu“. [17]

1.6 Statické testování

Často podceňovanou testovací metodou je takzvaný statický test, sestávající z manuální kontroly a statické analýzy. Na rozdíl od dynamického testu se testovaný objekt nespouští s testovacími daty, ale analyzuje. Tato analýza může být provedena pomocí jednoho nebo

několika lidí, kteří intenzivně kontrolují dokumenty nebo pomocí konkrétních nástrojů. Veškeré dokumenty v projektu vývoje softwaru lze kontrolovat ručně. Statická analýza podporovaná nástroji může být provedena pouze s dokumenty dodržujícími pravidla, díky kterým může být kontrola automatizovaná. Hlavním cílem všech testů je najít chyby a odchylky od stávajících specifikací, definovaných standardů nebo i od plánu projektu. [1, s.73]

1.7 Dynamické testování

Ve většině případů je testování softwaru považováno za spuštění testovaného objektu v počítači. Pro další objasnění se používá fráze dynamická analýza. Testovaný objekt (program) musí být spustitelný. Před provedením je opatřen vstupními daty. V nižších zkušebních stádiích (testování částí a integrace) nemůže být testovaný objekt spuštěn samotný, ale musí být vložen do testovacího lůžka pro získání spustitelného programu. **Testovací lůžko je nutné.** Testovací objekt obvykle zavolá různé části programu prostřednictvím předdefinovaných rozhraní „interfaces“. Tester musí často vytvářet zkušební lože sám nebo testovací nástroj musí rozšiřovat nebo modifikovat standardní (generické) testovací lůžka, které upravují na rozhraní testovaného objektu. Mohou být použity také generátory testovacích lůžek. [1, s.97, 98]

Systematický přístup při stanovení testovacích případů.

Při provádění programu musí testování odhalit chyby a ověřit co nejvíce požadavků s co možná nejmenšími náklady. K dosažení tohoto cíle je nutné zvolit systematický přístup. Nestrukturované testování obvykle nenabízí žádnou záruku. Tester by měl otestovat co nejvíce situací, ale nejlépe všechny situace, které testovaný objekt zpracovává odlišně.

Přírůstkový přístup

Pro provedení testů jsou nezbytné následující kroky:

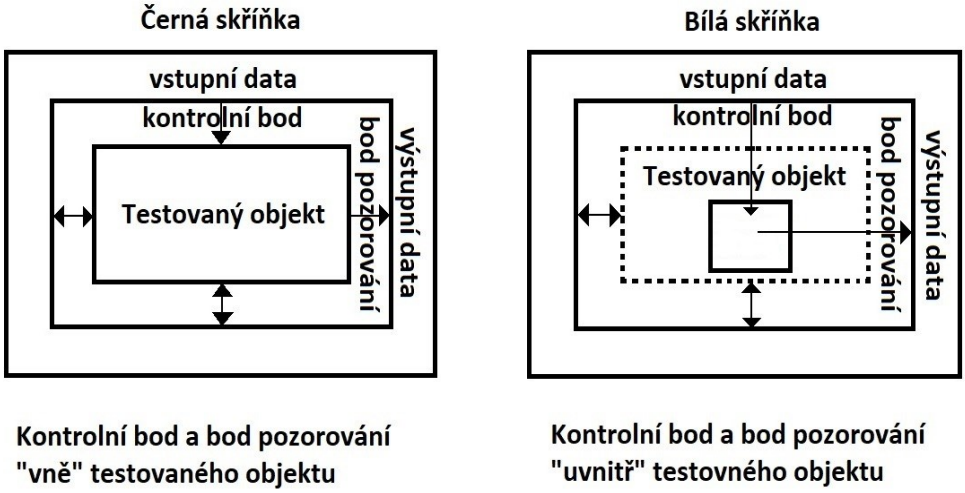
- Určit podmínky a předpoklady pro test a cíle, kterých je třeba dosáhnout.
- Zadat jednotlivé zkušební případy.
- Určit, jak provádět testy (obvykle se spojuje několik tetovacích případů).

Technika černé skříňky a bílé skříňky

Existuje několik různých přístupů k testování testovaného objektu. Mohou být rozděleny do dvou skupin: testování černé skříňky a testování bílé skříňky. Přesněji řečeno: Techniky návrhu zkušebních případů.

Při testování černých skříněk se zkoumaný objekt považuje za černou skříňku. Testovací případy jsou odvozeny ze specifikace testovaného objektu. Zachycení testovaného objektu je sledováno zvenku (PoO – Bod pozorování je mimo testovaný objekt). Kromě výběru vhodných vstupních testovacích dat, není možné kontrolovat operační sekvenci testovaného objektu. (PoC – Kontrolní bod se nachází mimo testovaný objekt). Testovací případy jsou navrženy pouze pomocí specifikace nebo požadavků testovaného objektu.

U testování bílé skříňky je zdrojový kód znám a používán pro testování. Při spouštění testovacích případů, je analyzováno interní zpracování testovaného objektu i výstupu (Bod pozorování je uvnitř testovaného objektu). Přímý zásah do procesu testovaného objektu je možný, ale měl by být použit ve zvláštních situacích. Například u spuštění negativního testování, když interface komponenty není schopen iniciovat podnícené selhání. (Bod kontroly může být umístěn uvnitř testovaného objektu.) Testovací případy jsou navrženy tak, aby pokryly programovou strukturu testovaného objektu. [1, s.100]



Obr. 1 Černá skříňka, Bílá skříňka [1, s.100]

Testování bílé skříňky se také nazývá strukturální testování, protože zkušební návrhář zvažuje strukturu (hierarchii komponent, řízení toku, tok dat) testovaného objektu.

1.8 Testování jednotlivých úrovní

1.8.1 Jednotkový test

V rámci první úrovně testování jsou softwarové jednotky systematicky testovány poprvé. V závislosti na programovacím jazyce jsou tyto softwarové jednotky nazývány různě, např. moduly, jednotky, programy nebo funkce. V objektově orientovaném programování se nazývají třídy. Příslušné testy se proto nazývají modulové, jednotkové, programové nebo třídní testy. Obecně mluvíme o softwarových jednotkách nebo komponentách. [1, s.38]

Objekty testování

Softwarové jednotky jsou testovány jednotlivě a izolovány od všech ostatních, aby se zabránilo vnějším vlivům na jednotku. Pokud testování zjistí problém, pochází určitě ze samotné jednotky. [1, s.38]

Testovací prostředí

Je zřejmé, že na této zkušební úrovni se testování provádí v těsné spolupráci s vývojem. Aby bylo možné psát testovací softwarové ovladače, je zapotřebí programovat s dovedností a znalostí testované jednotky. Zdrojový kód testovaného objektu (např. funkce třídy) musí být k dispozici a pochopen testerem tak, aby volání testovaného objektu bylo správně naprogramováno v testovacím ovladači. Generické testovací ovladače jsou vytvořeny specifické pro projekt nebo jsou k dispozici na trhu (např. xunit). Pokud se používají obecné testovací ovladače, testování týmovými kolegy, kteří nejsou obeznámeni s konkrétním prostředkem a programovacím prostředím, je jednodušší. Testovací ovladač by měl například poskytovat příkazové rozhraní a komfortní mechanismy pro zpracování testovacích dat a pro zaznamenávání a analyzování testů. [1, s.40,41]

Cíle testování

Testovací úroveň jednotkového testu není charakterizována pouze druhem testovaných objektů a testovacím prostředím, tester také sleduje cíle testování, které jsou specifické pro tuto fázi. Nejdůležitějším úkolem je zaručit, že konkrétní testovaný objekt provede všechny jeho funkce správně a úplně, jak vyžaduje specifikace (viz Testování funkčnosti). Zde funkce znamená funkci „Vstup/Výstup“ – chování testovaného objektu. Za účelem ověření správnosti a neporušenosti implementace je jednotka testována sérií zkušebních případů, kde každý zkušební případ pokrývá konkrétní vstupní/výstupní kombinaci (částečná funkčnost). Typické vady softwaru zjištěné během testování funkčních součástí jsou nesprávné výpočty, chybějící nebo nesprávně zvolená cesta programu (např. Zvláštní případy, které byly

zapomenuty nebo nesprávně vyloženy). Později, když je celý systém integrovaný, každá softwarová komponenta musí spolupracovat s mnoha sousedními komponentami a vyměňovat si s nimi data. Není možné vyloučit možnost, že komponenta bude volána (nebo použita) způsobem, který není v souladu s její specifikací, tj., komponenta je zneužita. V takových případech součást, která je volána by pozastavila svoje služby nebo způsobila zhroucení celého systému. Měl by být schopen zvládnout chybovou situaci schopným a robustním způsobem. [1, s.41]

Z tohoto důvodu je testování robustnosti dalším velmi důležitým aspektem testu komponent. Způsob, jak to udělat, připomíná funkční testy. Volání funkcí a testovaných dat je použito záměrně špatně, nebo ve speciálních případech, které nejsou uvedeny ve specifikaci. Takové testovací případy se také nazývají – negativní testy. Reakce komponenty by měla být vhodným řešením výjimek. Není-li taková výjimka zachována, nesprávné vstupy mohou spustit chyby, jako je dělení nulou nebo přístup přes nulový ukazatel. Takové chyby by mohly vést k havárii programu.

Některé zajímavé aspekty jsou jasné:

- Existuje přinejmenším tolik rozumných negativních testů jako pozitivních.
- Testovací ovladač musí být rozšířen, aby bylo možné vyhodnotit zacházení s výjimkou testovaného objektu.
- Zpracování výjimek testovaného objektu vyžaduje další funkce. Často více než 50 % kódu programu se zabývá zpracováním výjimek. Robustnost má své náklady.

Během testování komponenty by měly být zkontrolovány i nefunkční charakteristiky jako je účinnost a udržitelnost.

Účinnost uvádí, jak účinně jednotka používá počítačové zdroje. Zde máme různé aspekty, jako je využití paměti, výpočetní čas, doba přístupu k disku nebo síti a doba potřebná k provedení funkcí a algoritmů komponenty. Testy účinnosti se zřídka kdy provádějí pro všechny součásti systému. Účinnost musí být ověřena pouze v kriticky účinných částech systému nebo pokud požadavky na účinnost jsou specifikovány.

Udržitelnost představuje všechny vlastnosti programu, které mají vliv na to, jak snadné nebo jak obtížné je změnit program nebo jej dále rozvíjet. Zde je rozhodující, kolik úsilí musí vývojář vynaložit, aby pochopil program a jeho kontext. To platí pro vývojáře původního programu, který je po několika měsících nebo letech požádán o pokračování vývoje, a také pro programátora, který přebírá odpovědnost za kód, který napsal někdo jiný. Následující

aspekty jsou proto nejdůležitější pro testování udržitelnosti: struktura kódu, modularita a kvalita komentářů v kódu, dodržování norem, srozumitelnost a změna dokumentace. [1, s.43]

Testovací strategie

Tester má zpravidla přístup ke zdrojovému kódu, který provádí testování jednotek domény, testování „bílé skříňky“ (viz Dynamické testování). Tester může navrhnout test pomocí svých znalostí o programových strukturách jednotky, jejich funkcích a proměnných. Pomoci speciálních nástrojů – debugger, je možné sledovat programové proměnné během provádění testu.

Ve skutečnosti se však test komponenty často provádí jako čistý test „černé skříňky“, což znamená, že vnitřní struktura kódu se nepoužívá k návrhu testovacích případů.

Moderní přístup v testování komponent, který je populární při přírůstkovém vývoji, spočívá v přípravě a automatizaci testovacích případů před kódováním. Toto se nazývá testem řízený vývoj. Tento přístup je vysoce iterativní. Kousky kódu jsou testovány a poté vylepšeny dokud komponenta neprojde všemi testy. [1, s.44]

1.8.2 Integrační test

Po jednotkovém testu je druhou úrovní integrační test. Testování integrace předpokládá, že testované objekty, které jsou předmětem testu (tj. jednotky), již byly testovány. Vady by měly být pokud možno již opraveny. Skupiny těchto jednotek se skládají tak, že vytvářejí větší strukturální jednotky a subsystémy. Cílem integračního testu je ověření správné spolupráce mezi všemi dílčími jednotkami a odhalit závady v rozhraní nebo interakci mezi integrovanými jednotkami. Je úkolem nalézt problémy spolupráce, vnitřně operativní problémy a izolovat jejich příčiny. [1, s.45]

Objekty testování

V průběhu integrace jsou jednotlivé jednotky sestaveny krok za krokem do větších jednotek. V ideálním případě by se po každém z těchto kroků měl provést integrační test. Každý takto generovaný subsystém může být základem pro integraci ještě větších jednotek. Takové složené jednotky by samy mohly být opět objekty integračního testu. [1, s.48]

Testovací prostředí

Stejně jako u testování jednotek, jsou v integračním testu potřebné testovací softwarové ovladače. Odesílají zkušební data testovaným objektům a přijímají a zaznamenávají

výsledky. Protože testované objekty jsou tvořeny jednotkami, jež mají stejné rozhraní jako jednotky ze kterých jsou tvořeny, je zřejmé a rozumné znovu použít testovací ovladače, které již byly použity pro jednotkový test. Během integračního testování jsou vyžadovány další nástroje, tzv. monitory. Monitory jsou programy, které čtou a zaznamenávají datovou komunikaci mezi jednotkami. Monitory pro standardní protokoly (např. síťové protokoly) jsou komerčně dostupné. Pro pozorování rozhraní specifických pro jednotlivé projekty musí být vyvinuty speciální monitory. [1, s.48]

Testovací cíle

Cíl Integračního testu je zřejmý: odhalit problémy s rozhraním a spolupráci, jakož i konflikty mezi integrovanými jednotkami. Problémy již mohou nastat, když se pokusíme integrovat dvě jednotlivé jednotky. Spojení mezi nimi nemusí fungovat, protože například jejich formáty rozhraní nejsou vzájemně kompatibilní, nebo proto, že některé soubory chybí, nebo protože vývojáři rozdělili systém na zcela jiné komponenty, než byly navrženy (viz Statické testování) Následující typy závad lze hrubě rozlišit:

- Jednotka vysílá syntakticky nesprávná nebo žádná data.
- Přijímající jednotka nemůže pracovat nebo spadne (funkční porucha v jednotkách, nekompatibilní formáty rozhraní, poruchy protokolů).
- Komunikace funguje, ale příslušné komponenty interpretují přijatá data jiným způsobem (funkční porucha jednotky, rozporuplná nebo nesprávně vyložená specifikace).
- Data jsou přenášena správně, ale jsou přenášena v nesprávném čase nebo pozdě (časový problém) nebo jsou intervaly mezi převody příliš krátké (průchod, zatížení nebo kapacitní problém).
- Kromě funkčnosti mohou být v integračním testu provedeny také nefunkční testy. Ty mohou zahrnovat testování výkonu a rozhraní. [1, s.49]

Integrační strategie

Při vypracování plánů může tester následovat tyto obecné strategie integrace:

- **Integrace shora dolů:** Test začíná částí z nejvyšší úrovní systému, která volá jiné součásti, ale není volána sama (s výjimkou volání z operačního systému). „Stuby“ nahrazují všechny podřízené součásti. Úspěšně probíhá integrace s jednotkami nižší úrovně. Vyšší úroveň, která již byla testována, slouží jako testovací stroj.

Výhoda: Testovací softwarové ovladače nejsou potřeba nebo jsou požadovány pouze jednoduché, protože části vyšší úrovně, které již byly testovány, slouží jako hlavní část testovacího prostředí.
Nevýhoda: Části vyšší úrovně musí být simulovány testovacími ovladači.

- **Integrace zdola nahoru:** Test začíná elementárními jednotkami systému, které nevyzývají další součásti, s výjimkou funkcí operačního systému. Větší subsystémy jsou sestaveny z testovaných jednotek a poté jsou tyto integrované jednotky testovány.

Výhoda: Nejsou potřeba stubs (poskytují konzistentní odpovědi na volání během testu, obvykle neodpovídají na vše, co je naprogramováno v testu).
Nevýhoda: Části vyšší úrovně musí být simulovány testovacími ovladači.

- **Integrace ad-hoc:** Jednotky jsou integrovány do (náhodného) pořadí, ve kterém jsou dokončeny.

Výhoda: Šetří čas, protože každá součást je integrována co nejdříve do svého prostředí.
Nevýhoda: Jsou vyžadovány stubs stejně jako testovací ovladače.

- **Páteřní integrační strategie:** K dispozici je kostra nebo páteř, do kterých jsou jednotky postupně integrovány.

Výhoda: Jednotky lze integrovat v libovolném pořadí.
Nevýhoda: Je vyžadována pracovní kostra nebo páteř.
Integrace shora dolů nebo zdola nahoru v čisté podobě je aplikovatelná pouze na programové systémy strukturované striktně hierarchickým způsobem. Ve skutečnosti se to nestává často. Proto je ve skutečnosti zvolena více či méně individuální směs výše uvedených strategií integrace. [1, s.52]

1.8.3 Systémový test

Testování systému kontroluje, zda integrovaný produkt splňuje stanovené požadavky. U nižších testovacích úrovní bylo testování provedeno s ohledem na technické specifikace, tj. z technického hlediska výrobce softwaru. Systémový test se však podívá na systém z pohledu zákazníka a budoucího uživatele. Testeři ověřují, zda jsou požadavky splněny zcela a přiměřeně. Mnoho funkcí a systémových charakteristik je výsledkem interakce všech jednotek systému, a proto jsou viditelné pouze na úrovni celého systému a lze je zde pozorovat. [1, s.53]

Testovaný objekt a testovací prostředí

Po dokončení integračního testu je softwarový systém kompletně sestaven a systémový test se podívá na systém jako celek. To se provádí v prostředí, které je co možná nejvíce podobné pozdějšímu provoznímu prostředí. Namísto testovacích ovladačů a hardwarových nástrojů by měly být hardwarové a softwarové produkty, které jsou používány později, instalovány na zkušební platformě (hardware, systémový software, zařízení ovladačů softwaru, sítě, externí systémy). Snaha adekvátního systémového testu nesmí být podceňována. Zejména kvůli komplexnímu testovacímu prostředí. [1, s.54]

Testovací cíle

Jak je popsáno výše, cílem testu systému je ověřit, zda celý systém splňuje stanovené funkční a nefunkční požadavky a jak dobře to dělá. Měly by být zjištěny nedostatky v nesprávné, neúplné nebo nedůsledné implementaci požadavků. A požadavky, které jsou nedokumentovány nebo byly zapomenuty, by měly být identifikovány. [1, s.55]

Problémy v testování systému

V mnoha projektech je písemná dokumentace požadavků velmi neúplná nebo vůbec neexistuje. Poté testeři čelí problému, že není jasné, co představuje správné chování systému. A je těžké najít chyby. [1, s.55]

1.8.4 Akceptační test

Jde o pozornost zákazníka a jeho úsudek. Akceptační test je jediným testem, na kterém se zákazník skutečně podílí a kterému rozumí. Zákazník může být zcela zodpovědný za akceptační test.

Typické formy akceptačního testování zahrnují:

1. Testování s cílem určit, zda je smlouva splněna. Testovací kritéria jsou akceptační kritéria stanovená ve smlouvě o vývoji. Proto musí být tato kritéria formulována jasně a výslovně. V praxi si výrobce softwaru zkontroluje tato kritéria v rámci svého vlastního systémového testu. Pro akceptační test je pak dostačující znovu ukázat testovací případy, které jsou důležité pro přijetí zákazníkovi. A potvrdit, že jsou splněna kritéria smlouvy.
2. Testování akceptování uživateli
3. Provozní akceptační testování
4. Testy alfa a beta [1, s.56,57]

1.8.5 Testování nové verze produktu

Až doposud se předpokládalo, že vývoj softwarového produktu je ukončen, projitím akceptačním testem a nasazením nového produktu. Realita vypadá velmi odlišně. Jakmile je instalován, bude často používán roky nebo desetiletí a mnohokrát se mění, aktualizuje a rozšiřuje. Pokaždé, když k tomu dojde, je vytvořena nová verze původního produktu. [1, s.59]

Údržba softwaru

Mluvíme o údržbě softwaru, když je produkt přizpůsoben novým provozním podmínkám (adaptivní údržbě) nebo při odstranění závad (nápravná údržba). Testování, zda tyto změny fungují může být velmi obtížné, protože specifikace systémů jsou často zastaralé nebo chybějící.

Typické problémy, které se objeví i v nejvhodnějším softwarovém systému jsou např.: Systém je provozován za nových provozních podmínek, které nebyly předvídatelné a nebyly plánovány. Zákazníci vyjadřují nové přání. Zaznamenávají se havárie, které se vyskytují zřídka nebo jen po velmi dlouhém uplynutí času. Tyto havárie jsou způsobeny vnějšími vlivy.

Kromě údržbářských prací nezbytných z důvodu selhání existují změny a rozšíření produktu, které vedení projektu zamýšlí od začátku.

1. Plánovaná změna sousedního systému.
2. Funkčnost, která byla naplánována od počátku, ale nemohla být implementována co nejdříve.
3. Rozšíření, která jsou nezbytná v průběhu plánované expanze trhu.

Proto se softwarový produkt neukončil vydáním první verze. Vylepšená verze produktu bude dodávána v určitých časových okamžicích, např. jednou za rok. Po každém vydání projekt prakticky začíná znovu procházet všemi fázemi projektu. Proto se toto nazývá iterativní vývoj softwaru. V dnešní době je to obvyklý způsob vývoje softwaru. [1, s.60,61]

1.8.6 Generické typy testování

Zaměření a cíle se mění při testování na těchto různých úrovních popsaných v předchozích kapitolách. A různé typy testů jsou relevantní pro každou úroveň testu.

Následující typy testů lze rozlišit:

- funkční tetování;
- nefunkční testování;
- testování softwarové struktury;
- testování související se změnami. [1, s.63,64]

1.9 Analýza pokrytí kódu

Hlavní technikou testování Bílé skříňky je analýza pokrytí kódu (code coverage analysis). Analýza pokrytí kódu eliminuje díry v sadě testovacích případů. Identifikuje oblasti programu, které nejsou testovány. Jakmile jsou zjištěny, vytvoří se také testy k ověření netestovaných částí kódu. Tím se zvyšuje kvalita softwarového produktu. K provádění analýzy pokrytí kódem se nejčastěji používají techniky:

Statement coverage: technika vyžaduje vykonání všech příkazů ve zdrojovém kódu alespoň jednou. Míra pokrytí měří, zda byl příkaz alespoň jednou testován.

Branch coverage: technika kontroluje každou možnou cestu (if-else, podmíněné smyčky, atd.) softwarové aplikace.

Mezi další techniky patří **Condition coverage**, **Multiple Condition coverage**, **Path coverage, Function coverage**. Použitím Statement a Branch coverage se většinou dosáhne pokrytí 80-90%, což je dostatečné. [18]

1.10 Normy testování

Účelem řady norem ISO/IEC/IEEE 29119 testování softwaru je definovat sadu mezinárodních odsouhlasených norem testování softwaru, které mohou být použity kteroukoliv organizací při provádění jakékoliv formy testování softwaru.

ISO/IEC/IEEE 29119-1 je informativní - poskytuje výchozí bod, kontext a odborné vedení pro další části.

ISO/IEC/IEEE 29119-2 obsahuje popisy testovacích procesu, které definují procesy testování softwaru na úrovni organizace, úrovni řízení testování a úrovních dynamického testování. Podporuje dynamické testování, funkční a nefunkční testování, manuální a automatizované testování, a skriptované a neskriptované testování. Procesy definované v ISO/IEC/IEEE 29119-2 mohou být použity ve spojení s jakýmkoliv modelem životního cyklu vývoje softwaru.

ISO/IEC/IEEE 29119-3 zahrnuje šablony a příklady dokumentace testování. Šablony jsou uspořádány do kapitol odrážejících strukturu popisu celého testovacího procesu v ISO/IEC/IEEE 29119-2, tj. podle testovacího procesu, v kterém vznikají. ISO/IEC/IEEE 29119-3 podporuje dynamické testování, funkční a nefunkční testování, manuální a automatizované testování a skriptované a neskriptované testování. Šablony dokumentace definované v ISO/IEC/IEEE 29119-3 mohou být použity ve spojení s jakýmkoliv modelem životního cyklu softwarového vývoje.[19]

2 KOMPARACE NÁSTROJŮ PRO TESTOVÁNÍ SOFTWARU

Když mluvíme o testovacích nástrojích obecně, máme většinou na mysli nástroje pro automatizaci dynamických testů. Nástroje dodávají testovanému objektu testovací data, zaznamenávají reakce testovaného objektu a zaznamenávají provedení testu. [1, s.194]

Existují různé kategorie nástrojů, které podporují různé fáze a činnosti testovacích procesů. Jsou klasifikovány podle fází nebo aktivit, které podporují (např. nástroje pro testování výkonu, specializované na webové aplikace). Všechny dostupné kategorie testovacích nástrojů jsou v projektu použity jen ve velmi málo případech. [1, s.189]

2.1 Přehled nástrojů

NodeJS

NodeJS je platforma pro vytváření škálovatelných, vysoce výkonných síťových webových aplikací pomocí jazyka JavaScript:

- Aplikace v reálném čase, jako komunikační servery,
- aplikace pro spolupráci více uživatelů,
- Online hry pro více hráčů v reálném čase,
- aplikace, které vyžadují rozsáhlé Vstupy/Výstupy,
- aplikace, které vyžadují vysoký stupeň škálovatelnosti,
- aplikace, které vyžadují nepřetržité připojení k serveru.

S aplikací NodeJS je vytvořena skupina malých aplikací namísto jedné velké aplikace, což umožňuje provádět změnu nebo přidávat nové funkce, aniž by bylo zapotřebí provádět změny hluboko uvnitř celé kódové základny.[2]

Mocha

Mocha je rozsáhlý JavaScript testovací framework (rámec) fungující na NodeJS a v prohlížeči, takže asynchronní testování je jednoduché. Mocha testy probíhají sekvenčně a umožňují flexibilní a přesné hlášení, přičemž mapují nezachycené výjimky ze správných testovacích případů. Mocha zahrnuje spoustu skvělých funkcí např.: timeout podpora asynchronního testu, metody pro inicializaci a čištění prostředí (before(), after(), before each(), after each()).[3]

Chai

Externí knihovna Chai se instaluje do NodeJS, prohlížeče a jiného prostředí.

Assert modul se používá k psaní jednotkových testů testovaných aplikací (viz. praktická část) K jednotkám přistupujeme s požadavkem assert (assert – angl. sloveso – uvést s jistotou, potvrdit) Modul má několik metod např.: (assert.equal(), assert.deepEqual(), assert.throws()). Poskytuje účinný způsob detekce a opravy chyb při programování.
Styl assert v knihovně Chai je velmi podobný modulu assert v NodeJS. Chai zahrnuje dva druhy BDD stylů (BDD vývoj řízený chováním používaný u automatického testování). Rozhraní API je natolik flexibilní, že všechny synchronní úkoly mohou být snadno zapouzdřeny v rámci jediného assertu a znovu použity během testů. [4]

PhantomJS

Je prohlížeč postavený na WebKit jádře, který běží v headless modu a lze ovládat pomocí JavaScript. Má rychlou a přirozenou podporu pro různé webové standardy: DOM (objektový model dokumentu), CSS (jazyk pro popis způsobu zobrazení elementů), JSON (textový, na jazyce zcela nezávislý formát), Canvas (element HTML používaný k vykreslení grafiky pomocí scriptování v jazyce JavaScript v reálném čase) a SVG (škálovatelná vektorová grafika – popisuje dvourozměrnou grafiku pomocí XML). Zjednodušeně PhantomJs je webový prohlížeč bez grafického rozhraní. To vyvolává otázku: Jaké je použití prohlížeče bez grafického uživatelského rozhraní? Vzhledem k tomu, že PhantomJS není použitelný, pokud jde o procházení internetu, má celou řadu funkcí, které vývojáři používají pro mnoho účelů. Snímání obrazovky, automatizace stránky, sledování sítě, testování, atd. [5]

Selenium

Je sada nástrojů pro automatizaci webových prohlížečů na mnoha platformách. Primárně jde o automatizaci webových aplikací pro účely testování, ale určitě se to nijak neomezuje. Nudné úkoly, zprávy na webu mohou (a měli by být) také automatizované.

Jasmine

Je framework pro testování JavaScriptu. Neopírá se o prohlížeče, DOM nebo jakýkoliv framework JavaScriptu. Je tedy vhodný pro webové stránky, projekty Node.js nebo kdekoli, kde může být spuštěn JavaScript. Abychom mohli psát testy v TypeScriptu s frameworkem Jasmine, musíme nastavit projektové prostředí s některými specifikacemi. [8]

Sinon.JS

Je knihovna, která poskytuje falešné objekty. Pokud se vyvíjí bohatá a složitá aplikace, pak jsou falešné objekty velmi užitečnou součástí testovací sady nástrojů. Sinon.JS poskytuje tři typy falešných objektů.[7]

- Stubs - poskytují konzistentní odpovědi na volání během testu, obvykle neodpovídají na vše, co je naprogramováno v testu.
- Spies - jsou stuby , které také zaznamenávají nějaké informace založené na tom, jak byly volány. Jednou z těchto možností může být e-mailová služba, která zaznamenává, kolik zpráv bylo odesláno.
- Mocks - objekty předem naprogramované, které tvoří specifikaci volání, které se očekává.)

JSDOM

Je prohlížeč, který běží v headless modu (bez GUI), lze použít k vytvoření reálného testovacího prostředí. JSDOM je JavaScript implementace mnoha webových standardů pro použití s Node.js. Cílem projektu je obecně napodobit dostatečně velkou podmnožinu webového prohlížeče, který je užitečný pro testování webových aplikací v reálném světě. To znamená, že můžeme testovat v prostředích bez prohlížečů, jako v prostředí uzlů nebo v kontinuálním integračním prostředí. [9]

Intern

Je kompletní testovací systém pro JavaScript, který pomáhá při psaní testovacích případů a provádí konzistentní, vysoce kvalitní testovací případy. Intern je minimálně normativní a vynucuje pouze základní sadu osvědčených postupů navržených tak, aby zajistily, že testy zůstanou udržitelné v průběhu času. Jeho rozšiřitelná architektura umožňuje psát vlastní testovací rozhraní. Je také vynikající volbou pro testování v TypeScript. Podpora zdrojové mapy Internu usnadňuje reportování chyb zpět do zdrojových souborů v TypeScriptu.[10]

2.2 Výběr nástrojů

Samotný výběr (hodnocení) nástroje začíná, jakmile je objasněno, jakou testovací úlohu musí nástroj podporovat. Investice může být velmi velká. Proto je vhodné postupovat

pečlivě a dobře plánovaným způsobem. Výběrový proces se skládá z následujících pěti kroků:

1. Specifikace požadavků, které požadujeme po aplikaci(nástroji).
2. Průzkum trhu (vytvoření přehledu možných kandidátů).
3. Předvedení nástrojů a vytvoření krátkého seznamu.
4. Vyhodnocení nástrojů v krátkém seznamu.
5. Kontrola výsledků a výběr nástroje.

Pro automatizaci testů je jako nejvhodnější nástroj vybráno Selenium. S ohledem na další automatizační nástroje má Selenium možnost pracovat téměř na všech operačních systémech. Selenium podporuje více programovacích jazyků, všechny základní prohlížeče, má řadu robustních metod pro lokalizaci prvků jako CSS, Xpath, DOM atd., poskytuje podporu pro integraci open source framework. S využitím Selenia je možné souběžně spouštět testy využívající různé prohlížeče na různých strojích.

Pro spouštění testů je ideální Intern, který zahrnuje jednotkové testování, funkční testování, TypeScript, analýzu pokrytí kódu, spouští testy paralelně pro lepší výkon. Testuje kód Node.js, může být použit s libovolnou knihovnou assert funkcí, umožňuje napsat vlastní testovací rozhraní, používá Promis pro asynchronní testování, automaticky zpracovává chyby serveru Selenium.[10]

Chai je knihovna pro prohlížeč, která může být spárována s javascriptovým testovacím frameworkem.

Jako virtuální prohlížeč byl zvolen JSDOM. Je to čistý JavaScript a běží kdekoli běží Node.js. Má experimentální podporu pro běh v prohlížečích, což dává možnost vytvořit celý dokument DOM uvnitř webu. Jedním z důvodů, proč se JSDOM hodí pro testování, je, že vytváření nové instance dokumentu má v JSDOM velmi malou režii, takže běh mnoha malých testů může trvat jen pár sekund.[13]

3 WUI FRAMEWORK

WUI Framework neboli Web-based User Interface Framework je softwarová platforma pro tvorbu hybridních webových aplikací. Hlavním cílem hybridních webových technologií je uplatnění programovacích jazyků původně určených pro vývoj webových prezentací do oblastí ryze desktopového či mobilního prostředí bez nutnosti trvalého připojení k internetu. Vývoj WUI Framework byl zahájen jako interní projekt ve společnosti Freescale Semiconductor (nyní NXP Semiconductors) s uzavřenou licencí. Rozvoj vnitropodnikového softwaru byl zahájen, protože v té době neexistovalo dostatečně robustní řešení, které by plně vyhovovalo specifickým nárokům na tvorbu multiplatformních aplikací ve Freescale. Dostupná řešení byla typická použitím různorodých technologií dle cílové platformy, ale tento přístup byl pro aplikace vyvíjené ve Freescale prostředí nevhodný, protože vedl na značnou duplicitu kódu. Inovovaný koncept hybridních webových technologií se naproti tomu jevil jako jedna z možných cest, jak uspokojit většinu požadavků na vývoj multiplatformních aplikací a současně redukovat duplicitu kódu. Postupem času bylo čím dál jasnější, že WUI Framework může být použit jako univerzální softwarová platforma pro vývoj multiplatformních aplikací i za hranicemi společnosti Freescale a proto došlo k rozhodnutí o změně licence z uzavřené na volně šiřitelnou BSD-3-Clause.

Za dobu vývoje WUI Frameworku se ve světe objevila celá řada konkurenčních řešení, ale žádné z nich nemůže plně obsáhnout veškeré vnitropodnikové požadavky. Například maximální možná automatizace vývojového a testovacího procesu, splňující vnitropodnikové nároky na kontinuální integraci a dodávky je naprosto nutná, ale nelze ji dosáhnout bez značných úprav konkurenčních řešení, a proto se ve vývoji WUI Frameworku dále pokračuje. WUI Framework se od konkurenčních řešení liší i celou řadou přístupů jako je například čistě objektově orientované programování prezenční vrstvy. Striktně oddělená prezentační a výkonná vrstva umožňující testování a spuštění v headless módu. Možnost nasazení produktu na širokou škálu prostředí zahrnující i rozšíření pro IDE třetích stran. Kladou se vysoké nároky na úroveň zabezpečení a míru opětovného využití. WUI Framework je typický možností bezproblémového chodu i z lokálního souborového systému při absenci serveru a vysokou kompatibilitou s prohlížeči.

Hlavními programovacími jazyky jsou TypeScript (silně typová alternativa pro JavaScript) a SASS (robustní nadstavba nad CSS), ale části WUI Frameworku se vyvíjejí i v C++, JAVA

či PHP. Široká jazyková základna se řeší specifickou cross-language API a jednotnou strukturou projektů což umožnuje bezproblémovou kombinaci jednotlivých částí. [15]

II. PRAKTICKÁ ČÁST

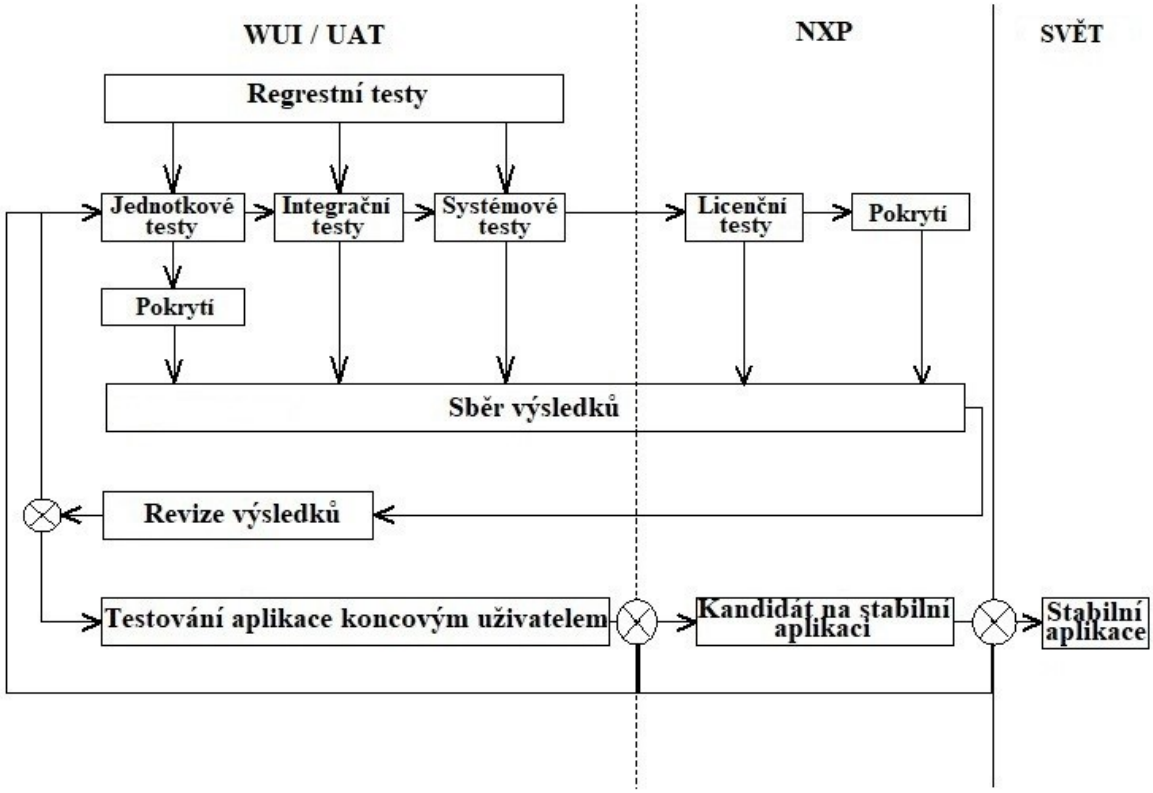
4 NÁVRH STRUKTURY A METODIKY TESTOVÁNÍ

Volba správné metodiky testování softwaru spolu s návrhem optimální struktury testování je nutná podmínka pro zajištění efektivního testovacího procesu pro danou aplikaci.

4.1 Struktura testování

Struktura testování WUI Frameworku obsahuje všechny základní úrovně testování a její podoba je ilustrována na blokovém schématu Obr. 3. Toto schéma poukazuje na zařazení jednotlivých úrovní testování v komplexní metodice testování softwaru jako celku.

Tato diplomová práce se majoritně soustředí na testování softwaru prostřednictvím jednotkových testů ve spolupráci s analýzou pokrytí kódu.



Obr. 2 Struktura testování

4.2 Struktura jednotkového testu

Při psaní testu se může vyskytnout nějaké nastavení, které je potřeba provést před spuštěním testu nebo po spuštění testu.

setUp(), tearDown(), before(), after()

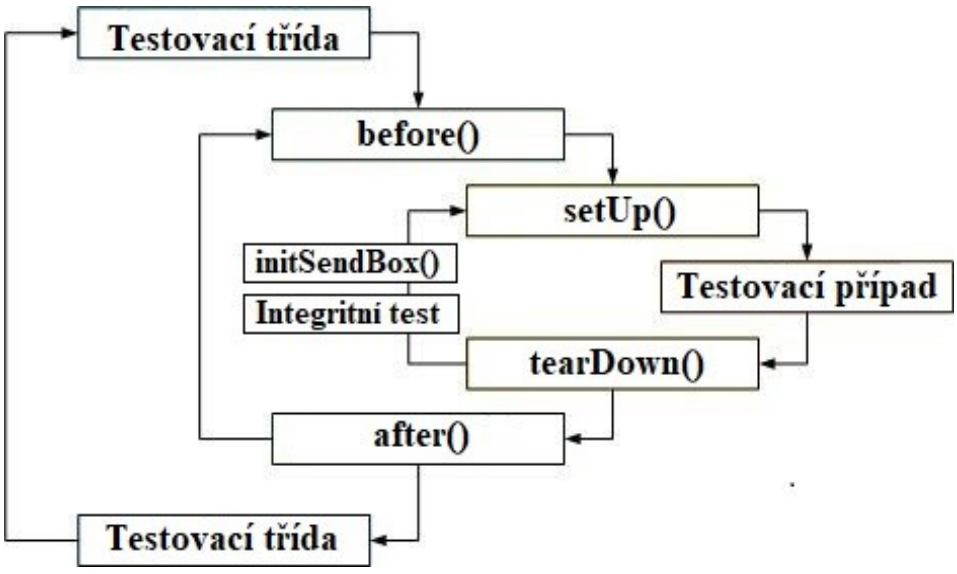
Pokud se vyskytne taková úloha, která je potřeba provést opakovaně pro mnoho testovacích případů, může se použít funkce **setUp()** a **tearDown().** V některých případech stačí provést nastavení pouze jednou, na začátku souboru. K tomu lze použít funkce **before(), after() .**

initSendBox()

Prostor, který zajišťuje kontrolovaný běh testu. Zajišťuje, aby nedocházelo k ovlivňování mezi jednotlivými integračními testy. Sandbox je zde v roli jakéhosi bezpečného prostoru, který zjednodušuje psaní testu a minimalizuje rozdíl mezi single jednotkovým testem a celkovým testem. V tomto případě se nejedná o virtualizaci ani omezení přístupu ke zdrojům.

Integritní test

Sledování stavu systému a vyhodnocování změny souborů, systémových oblastí a obsahu adresářů. Primárním účelem je kontrola, že v systému nezůstaly opomenuty nastavení, které byly provedeny během testu. Za tímto účelem je prováděna tato metoda jako nadstavba testovacího lůžka. Vznik integritního testu byl podmíněn nutností docílit vyšší stability při vývoji testu bílé skříňky v rámci jednotlivých architektonických bloků WUI Frameworku, kde přímá kontradikce není předem známá a objeví se až při spuštění testů ve vyšším hierarchickém bloku.

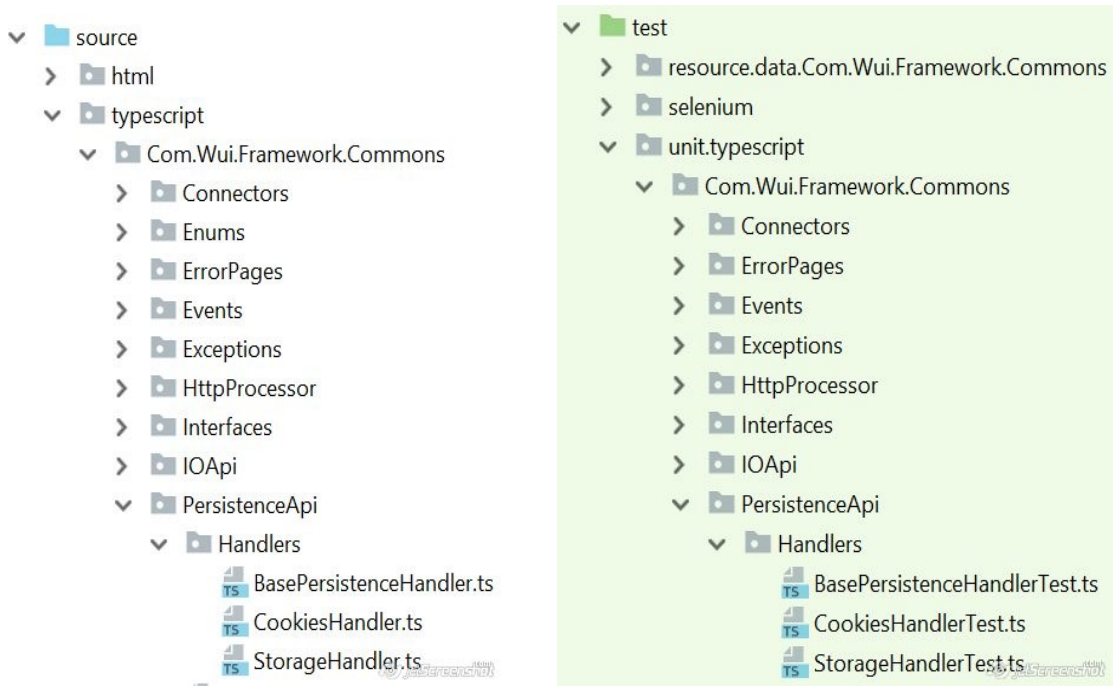


Obr. 3 Struktura testu

4.3 Struktura projektu

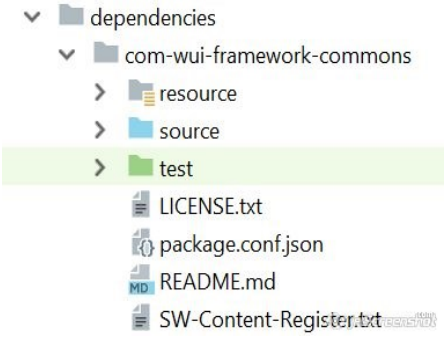
Všechny projekty vycházející nebo používající WUI Framework dodržují totožnou strukturu projektu znázorněnou na Obr. 4. Základní struktura projektu obsahuje adresář resouce (zdroje dat např. obrázky použité v projektu), klíčový adresář source s podadresáři se zdrojovými soubory projektu, adresář test zahrnující podadresáře se soubory testů.

Adresáře source a test mají stejnou strukturu, což velice usnadňuje orientaci v projektu. Testovací soubory se od zdrojových souborů liší klíčovým slovem „Test“ obsaženým v názvech souborů.



Obr. 4 Struktura projektu

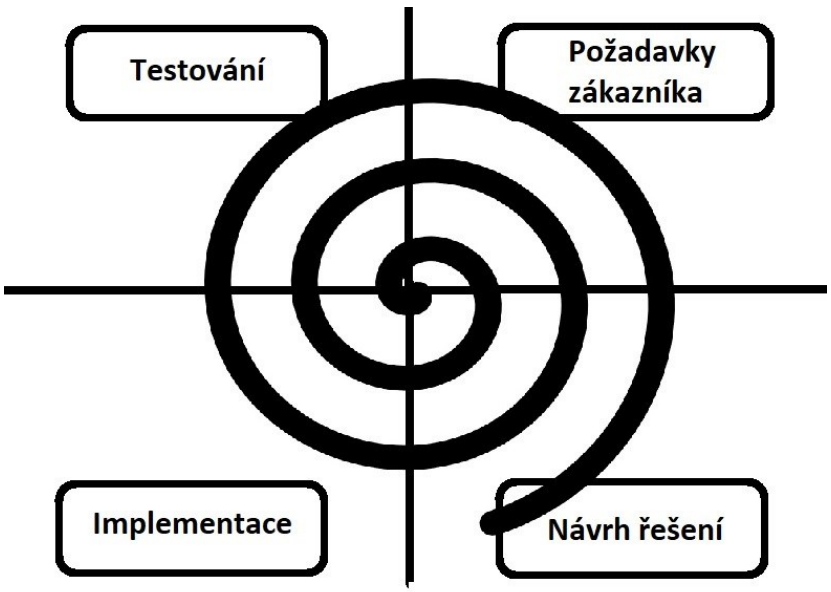
Závislý projekt je automaticky umístěn do složky „dependencies“ v kořenovém adresáři projektu, jak je patrné na Obr. 5.



Obr. 5 Struktura závislého projektu

4.4 Iterativně inkrementální model

Životní cyklus vyvíjeného softwaru jak je vidět na Obr.7 se pohybuje jakoby po spirále. Jedna iterace znamená přechod přes více aktivit (požadavky, řešení, programování, testování, atd.). Na začátku každé iterace může zákazník upřesnit svoje požadavky a výstupem z iterace bude funkční aplikace nebo její část. To, že tým je v přímém kontaktu s klientem, a zároveň prostředím, ve kterém bude software pracovat je velkou výhodou pro všechny zúčastněné. Klient má přehled v jakém stádiu je jeho produkt, vývojáři mohou lépe pochopit, jak se software bude používat a testeři mohou napsat kvalitnější testovací případy.



Obr. 6 Iterativně inkrementální model

Tento model zahrnuje hlavně automatizované testování a to z důvodu, že projekt se neustále mění a neustále přibývá nová funkcionalita. Přibývá nutnost psát regresní testy, proto se tester v takovém projektu musí angažovat od začátku. [14, s.30]

5 IMPLEMENTACE TESTŮ PRO OTESTOVÁNÍ WUI FRAMEWORK

Na začátku této aktivity se analyzuje testovací základ, aby se zjistilo, co musí být testováno. Například test by měl ukázat, že časová omezení jsou splněna, nebo některé transakce „operace“ správně provedeny. Určujeme testovací cíle pro prokázání splnění požadavků. Riziko selhání by se mělo pečlivě hlídat. Jsou stanoveny nezbytné předpoklady a podmínky testu.

Při testování je použitý TypeScript, což je jazyk a sada nástrojů pro generování kódu JavaScript. Jedná se o nadstavbu nad jazykem JavaScript, která jej rozšiřuje o statické typování a další atributy, které známe z objektově orientovaného programování.

Statické testování je zajištěno Lint testem kontrolujícím strukturu kódu prostřednictvím knihovny TSLint. Pomáhá zachytit chyby, vyžaduje jednotný styl kódu, zabraňuje příliš složitému kódu.

Všechny jednotkové testy (třídy testů) extendují (dědí) testovací script UnitTestRunner. Důvodem je sdílení přetížených testovacích metod a zajištění reportů (zpráv výsledů testů). Třída UnitTestRunner je rozšíření testovacího lůžka tak, aby prostředí, které nabízí Intern lépe odpovídalo potřebám WUI Frameworku. Přičemž je tato třída generalizovaná třída zrcadlena i do dalších částí WUI Frameworku, které používají jiné programovacích jazyky jako jsou JAVA, PHP a C++.

5.1 Moduly assert používané k testování

assert.equal(actual,expected)

Testuje mezi skutečnými (actual) a očekávanými (expected) parametry pomocí porovnání abstraktní rovností (==) (the Abstract Equality Comparison).

```
1. public testRemove() : void {
2.     assert.equal(String.Remove("StRiNg", "Ri"), "StNg");
3.     assert.equal(String.Remove("StRiNg"), "StRiNg");
4.     assert.equal(String.Remove("StRiNg", "StRiNg"), "");
5.     assert.equal(String.Remove("", "StRiNg"), "");
6.     assert.equal(String.Remove("StRiNg", "pr"), "StRiNg");
7.     assert.equal(String.Remove("StRiNg", "pr", "Ng"), "StRi");
8. }
```

Porovnání řetězců s použitím assert.equal.

```
1. public testResult() : void {
2.  const async : AsyncRequestEventArgs = new AsyncRequestEventArgs
3. ("http://localhost:8888/InternEnvironmentHelper.js/");
4.     async.Result(true);
5.     assert.equal(async.Result(), true);
6.     async.Result(false);
7.     assert.equal(async.Result(), false);
8. }
```

Porovnání hodnot typu boolean.

```
1. public testNativeEventArgs() : void {
2.     const event : MessageEventArgs = new MessageEventArgs();
3.     const messageEvent : any = {origin: "test"};
4.     event.NativeEventArgs(messageEvent);
5.     assert.equal(event.NativeEventArgs(), messageEvent);
6.
```

Porovnání objektů. Assert.equal() neporovnává objekt hlouběji, tj. privátní proměnné, ale srovnává paměťové bloky objektů. Paměťové bloky jsou stejné, protože jde o stejné třídy.

```
1. public testClear() : void {
2.     const manager : EventsManager = new EventsManager();
3.     const args : EventArgs = new EventArgs();
4.     const handler : any = ($eventArgs : EventArgs) : void => {
5.         args.Owner(GeneralEventOwner.WINDOW);
6.         args.Type(EventType.ON_START);
7.         assert.equal(GeneralEventOwner.WINDOW, "window");
8.         assert.equal(EventType.ON_START, "onstart");
9.     };
10.
11.     manager.setEvent("test1", "type1", handler);
12.     manager.setEvent("test1", "type2", handler);
13.     manager.setEvent("test1", "type3", handler);
14.     manager.setEvent("test2", "type1", handler);
15.     manager.setEvent("test3", "type1", handler);
16.     manager.setEvent("45", "type", handler);
17.
18.   assert.equal(manager.getAll().getItem("test1").Length(), 3);
19.   manager.Clear("test1", "type1");
20.   assert.equal(manager.getAll().getItem("test1").Length(), 2);
21.   manager.Clear("test1");
22.   assert.equal(manager.getAll().KeyExists("test1"), false);
23.   manager.Clear(45);
24.   assert.equal(manager.getAll().KeyExists(45), false);
25.   manager.Clear();
26.   assert.equal(manager.getAll().Length(), 0);
27. }
```

Porovnání výčtového typu enum a čísel. Test zajišťuje neasynchronní ověření, že po nastavení událostí, jsou všechny správně uloženy v manažerovi událostí. A po zavolání funkce Clear() také všechny odstraněny.

Programové rozhraní IUnitTestRunnerPromise

Upravené testovací lůžko UnitTestRunner poskytuje programové rozhraní IUniTestRunnerPromise vhodné pro implementaci **asynchronního testování**.

```
1. public testIsOnline2() : IUnitTestRunnerPromise {
2.     return ($done : () => void) : void => {
3.         (<any>window.navigator).__defineGetter__("onLine",
4. () : boolean => {
5.             return false;
6.         });
7.         HttpManager.IsOnline(($status : boolean) : void => {
8.         (<any>window.navigator).__defineGetter__("onLine",
9. () : boolean => {
10.             return true;
11.              });
12.              assert.equal($status, false);
13.              this.initSendBox();
14.              $done();
15.          });
16.      };
17.  }
```

Test, který používá IUnitTestRunnerPromise je charakteristicky návratovou hodnotou v podobě struktury řídicí vykonávání asynchronního běhu. Asynchronní běh je takový, který po provedení testovací metody neskončil synchronně s koncem testovací metody, ale který dál pokračuje na pozadí. Testovacímu prostředí se tedy musí dát najevo, kdy lze vykonat asserty a řádně ukončit běh testované metody. Signál pro řádné ukončení testu se provádí pomocí takzvaného zpětného volání (callback) jež je přístupný z návratové struktury IUnitTestRunnerPromise jako funkce $done(). V případě, že se $done() nikdy nevykoná, například kvůli odhalené chybě v kódu, bude úloha po 10 s automaticky ukončena v důsledku selhání nastaveného časovače (failure Timeout) [16]

```
protected setUp() : void {
    this.tmpHref = window.location.href;
    this.setUrl("http://localhost.wuiframework.com/index.html#/
                                projectname/unit/HttpManagerTest");
    const manager : HttpManager = new HttpManager(
            new HttpRequestParser("project-name"));
}

protected after() : void {
    this.setUrl(this.tmpHref);
    const manager : HttpManager = new HttpManager(
                                        new HttpRequestParser());
}

protected tearDown() : void {
    (<any>HttpManager).request = null;
}
```

Pro správné fungování předchozího testovacího případu, je nutné zajistit požadavek i jeho zrušení.

assert.deepEqual(actual,expected)

Testy pro hlubší rovnost mezi skutečnými (actual) a očekávanými (expected) parametry. Základní hodnoty jsou porovnány abstraktní rovností (==). Jsou brány v úvahu pouze vyčíslitelné „vlastní“ vlastnosti.

```
public testAdd() : void {
    const array3 : ArrayList<string | number> =
                                    new ArrayList<string | nuber>();
    array3.Add("test1", "key1");
    array3.Add("test2", "key2");
    array3.Add("test3", "key3");
   assert.deepEqual(array3.getKeys(), ["key1", "key2", "key3"]);
   assert.deepEqual(array3.Length(), 3);
}
```

Porovnání hodnot uvnitř objektu typu string i number.

```
public testRequest() : void {
    (<any>HttpManager).request = null;
    assert.deepEqual(HttpManager.Request(),
                                HttpRequesParser.getInstance());

    const parser : HttpRequestParser = new HttpRequestParser();
    const manager : HttpManager = new HttpManager(parser);
    assert.deepEqual(HttpManager.Request(), parser);

     (<any>HttpManager).request = null;
     HttpManager.Request();
}
```

Assert deepEqual() slouží taktéž k porovnání instancí.

```
1. public testgetActive() : void {
2.     const manager : GuiObjectManager = GuiObjectManager
3.                                         .getInstanceSingleton();
4.     const gui : GuiCommons = new MockGuiCommons("id10");
5.     const gui2 : GuiCommons = new MockGuiCommons("id11");
6.     manager.Add(gui);
7.     manager.Add(gui2);
8.     manager.setActive(gui, true);
9.     manager.setActive(gui2, false);
10.      assert.deepEqual(manager.IsActive(<any>gui), true);
11.      assert.deepEqual(manager.IsActive(MockGuiCommons), true);
12.      assert.equal(manager.getActive(gui).ToString("", false),
13.          "Com.Wui.Framework.Commons.Primitives.ArrayList
14.  object\r\nData object EMPTY");
15.    }
```

Porovnání hodnot typu boolean. Testujeme, zda funkce getActive(), vrátí seznam aktivních elementů prvku, který jí byl předán.

```
1. public testSplit() : void {
2. assert.deepEqual(String.Split("St*Ri*Ng", "*"), ["St", "Ri","Ng"]);
3. assert.deepEqual(String.Split("name1:value1;name2:value2", ";",
4. ":"), ["name1", "value1", "name2", "value2"]);
5. assert.deepEqual(String.Split("*string*", "*"), ["", "string", ""]);
6. assert.deepEqual(String.Split("string"), ["string"]);
   }
```

Assert.deepEqual pracuje uvnitř řetězce.

```
1. class MockGuiObject extends DrawGuiObject {
2. public testcontextMenu() : string {
3. const object : DrawGuiObject = new MockGuiObject();
4.          return this.contextMenu();
5.      }
6. }
7. public testMenu() : void {
8.     const drawGui : MockGuiObject = new MockGuiObject();
9.     assert.deepEqual(drawGui.testcontextMenu(),
10.  "<div guiType=\"DeveloperCorner\"
11.                                 class=\"DeveloperCorner\">\r\n" +
12.  "<div id=\"DeveloperCorner_ContextMenu\"
13.                                 class=\"ContextMenu\">\r\n" +
14.  "<div id=\"DeveloperCorner_ContextMenu_ViewCode\"
15.                       class=\"Link\">View HTML Code</div>\r\n" +
16.  "<div id=\"DeveloperCorner_ContextMenu_GenerateCache\"
17.                       class=\"Link\">Generate Cache</div>\r\n" +
18.  "<div id=\"DeveloperCorner_ContextMenu_BackToIndex\"
19.                        class=\"Link\">Back to index</div>\r\n" +
20.  "</div>\r\n</div>");
21.     this.initSendBox();
22.  }
```

Vzhledem k tomu, že je testována třída DrawGuiObject, která je abstraktní. Musí být nahrazena Mock objektem, jak je vidět na příkladu.

Mock objekt nahrazuje původní objekt za jeho testovací imitaci, která neprovádí žádnou funkcionalitu a jen se tváří jako původní objekt. Mock objekt předstírá chování, které potřebujeme kvůli testování. A aby bylo možné mock předávat metodám, je potřeba, aby třída mocku dědila od třídy, kterou nahrazuje. V testech WUI Frameworku, je utvářen za účelem nahrazení abstraktní třídy nebo k otestování protected metod.

```
1. public testgetMethods() : void {
2.     const object : BaseObject = new MockBaseObject();
3.     assert.deepEqual(object.getMethods(), [
4.          "getUID",
5.          "getClassName",
6.          "getNamespaceName",
7.          "getClassNameWithoutNamespace",
8.          "getProperties",
9.          "getMethods",
10.         "IsTypeOf",
11.         "IsMemberOf",
12.         "Implements",
13.         "SerializationData",
14.         "ToString",
15.         "getHash",
16.         "excludeSerializationData",
17.         "excludeIdentityHashData"
18.     ]);
19. }
```

Porovnává pole a vrací všechny metody instance.

assert.patternEqual(actual,expected)

Test pro porovnání výpisu objektu. Porovnává řetězce a umožňuje zavést určitý stupeň volnosti. Je uplatňován při testování objektu ve spojení s funkcemi pro jeho výpis do HTML formátu nebo řetězce. Podporuje hvězdičkovou notaci. Zápis je jednodušší a přehlednější než RegExp (Regulární výraz je textový vzor sestávající z kombinace alfanumerických znaků a speciálních znaků známých jako metaznaky).

```
1.  public testgetAll() : void {
2.      const manager : GuiObjectManager = new GuiObjectManager();
3.      const gui : GuiCommons = new MockGuiCommons("id4");
4.      const gui2 : GuiCommons = new MockGuiCommons("id5");
5.      const gui3 : GuiCommons = new MockGuiCommons("id6");
6.      manager.Add(gui);
7.      manager.Add(gui2);
8.      manager.Add(gui3);
9.      this.resetCounters();
10.     assert.patternEqual(manager.getAll().toString(),
11. "<i>Com.Wui.Framework.Commons.Primitives.ArrayList object</i> " +
12. "<span onclick=\"" +
13.         "document.getElementById('ContentBlock_0').style.display=" +
14. "docment.getElementById('ContentBlock_0').style.display===" +
15. "'block'?'none':'block';\" " +
16.         "style=\"CURSOR: pointer; FONT-SIZE: 10px; COLOR: #BE0000; FONT-FAMILY:
17.  Verdana, sans-serif;\">Open/Close</span><br/><span id=\"ContentBlock_0\"
18.   style=\"border: 0 solid black; display:none;\">[ \"id4\"
19. ]    Com.Wui.Framework.Gui.Primitives.GuiCommons (id4)<br/>"
20. + "[ \"id5\"
21. ]    Com.Wui.Framework.Gui.Primitives.GuiCommons (id5)<br/>"
22. + " [ \"id6\"
23. ]    Com.Wui.Framework.Gui.Primitives.GuiCommons
24. (id6)<br/></span>");
25. }
```

Testovací případ pro ověření obsahu instance GuiObjectManager. Po výpisu obsahuje všechny své Gui objekty, které do ní byli vloženy.

assert.doesNotThrow(block,[error])

Tvrdí, že funkční blok nevyhodí chybu. Když je volán assert.doesNotThrow(), okamžitě zavolá funkci blokování. Pokud dojde k chybě a je to stejný typ chyby, jako je zadán v parametru chyby, pak je chyba vyhozena. Pokud je chyba jiného typu nebo parametr chyby je nedefinovaný, je chyba vrácena zpět volajícímu.

```
1. public testPrintCode() : void {
2.     Echo.ClearAll();
3.     assert.doesNotThrow(() : void => {
4.         Echo.PrintCode("<span>test code</span>");
5.     });
6.     assert.equal(this.getEchoOutput(),
7.      "<span gutype=\"HtmlAppender\"><br><pre>&lt;span&gt;test
8.  code&lt;/span&gt;</pre></span>");
9. }
```

Test case kontroluje, že funkce PrintCode() nevyhodí výjimku pokud je volaná.

```
1. public testHandleExceptionWithMoreInfo() : void {
2.     try {
3.         ExceptionsManager.Throw("test", "message", 58,
4.                                         "ExceptionManagerTest", 120);
5.     } catch (ex) {
6.         assert.doesNotThrow(() : void => {
7.             ExceptionsManager.HandleException(ex);
8.         });
9.     }
10. }
```

Kód v try-catch bloku se snaží zachytit výjimky a zachycené výjimky jsou zpracovány metodou HandleException(). Třída ExceptionManager zpracovává výjimky, ale sama o sobě nesmí vyvolat výjimku.

```
1. public testPreventDefaultWithoutRequiredAPI() : void {
2.     const args : EventArgs = new EventArgs();
3.     const nativeEventArgs : any = {
4.         type: "MouseEvent"
5.     };
6.     args.NativeEventArgs(nativeEventArgs);
7.     assert.equal(args.NativeEventArgs(), nativeEventArgs);
8.     assert.doesNotThrow(() : void => {
9.         args.PreventDefault();
10.      });
11.      assert.equal(nativeEventArgs.returnValue, false);
12. }
```

```
1. public testConstructor() : void {
2.     assert.doesNotThrow(() : void => {
3.         const instance : Exception = new Exception();
4.     });
5.     const exception : Exception = new Exception("message");
6.     assert.equal(exception.Message(), "message");
7. }
```

Testuje se třída jako systém frameworku, záměrem je, že instance se dá vytvořit, aniž by došlo k selhání.

assert.onRedirect(block,block,block,[Page]

Existuje snadný způsob, jak v jednotkovém testu ověřit, že akce byla skutečně přesměrována na určitou stránku?

```
1.  public testReturn404NotFound() : IUnitTestRunnerPromise {
2.     return ($done : () => void) : void => {
3.        assert.onRedirect(
4.          () : void => {
5.             assert.doesNotThrow(() : void => {
6.
7.    HttpManager.Return404NotFound("http://localhost:8888/required/path/location");
8.                 });
9.            },
10.         ($eventArgs : AsyncRequestEventArgs) : void => {
11.       assert.equal($eventArgs.Url(), "/unit/ServerError/Http/NotFound");
12.       assert.ok($eventArgs.POST().KeyExists(HttpRequestConstants.HTTP404_FILE_PATH));
13.       assert.equal($eventArgs.POST().getItem(HttpRequestConstants.HTTP404_FILE_PATH),
14.               "http://localhost:8888/required/path/location");
15.            },
16.          () : void => {
17.             this.initSendBox();
18.               $done();
19.            },
20.            "unit");
21.     };
22. }
```

Tento testovací případ je asynchronní a proto použijeme rozhraní IUnitTestRunnerPromise, ve kterém musíme všechny pro test specifické metody volat v jeho zpětném volání tzv. Callback.

První parametr je callback pro přesměrování na chybovou stránku, má jako parametr cestu, která bude požadována při reportování chybové stránky. Druhým parametrem je handler, kterému předáme asynchronní argumenty událostí požadavků a v bloku se ujišťujeme, že volba přesměrování byla přesně shodná s přesměrováním, které jsme zvolili v předchozí akci. Ve třetím parametru je v handleru odebráno vlákno a funkce $done() potvrzuje, že kód byl spuštěn a skutečně se vrátil s výsledkem. Čtvrtým parametrem je základní URL adresa.

```
1. public testReloadTo2() : IUnitTestRunnerPromise {
2.    return ($done : () => void) : void => {
3.    (<any>HttpManager).request = HttpRequesParser.getInstance("unit");
4.          const list : ArrayList<any> = new ArrayList<any>();
5.          list.Add("test", HttpRequestConstants.ASYNC_SESSION);
6.          assert.onRedirect(
7. () : void => {
8.           HttpManager.ReloadTo(
9. "http://localhost.wuiframework.com/index.html#/test", list, true);
10.              },
11.          ($eventArgs : AsyncRequestEventArgs) : void => {
12.           assert.equal(window.location.href,
13.  "http://localhost.wuiframework.com/index.html#/projectname/
14.                                                  unit/HttpManagerTest");
15.    },
16.  () : void => {
17.            this.initSendBox();
18.              $done();
19.               });
20.       };
21.  }
```

Testovací případ ověřuje, že funkce ReloadTo() přesměruje a otevře odkaz v novém okně.

assert.throws(block,[error])

block <Function>

error <RegExp> | <Function>

```
1. public testLoadException() : IUnitTestRunnerPromise {
2.     return ($done : () => void) : void => {
3.         assert.throws(() : void => {
4.             JsonpFileReader.Load("", null);
5.             throw new Error("Not Found File Path.");
6.         }, /Not Found File Path./);
7.         $done();
8.     };
9. }
```

Očekává, že funkční blok vyhodí chybu. Pokud je zadána, může být chyba konstruktor, RegExp nebo ověřovací funkce.

Ověření (validace) chybové zprávy pomocí RegExp.

assert.notEqual(actual,expected)

actual <any>

expected <any>

Testuje pomocí porovnání abstraktní rovností mezi (actual) a očekávanými (expected) parametry (!=) (the Abstract Equality Comparison).

```
1. private timeout : TimeoutManager;
2.
3. public testgetId() : void {
4.     const timeout : TimeoutManager = new TimeoutManager();
5.     assert.notEqual(this.timeout.getId(), timeout.getId());
6. }
```

assert.resolverEqual([className], expected, [eventArgs]) : BaseHttpResolver

className **<any>**

expected <string>

eventArgs <any>

Modul assert.resolverEqual testuje objekt poskytující základní Http request (požadavek) resolver. Slouží k synchronnímu otestování. Vytiskne obsah třídy do prohlížeče, slouží k otestování očekávaného generovaného obsahu po redirect (přesměrování). Podporuje hvězdičkovou notaci.

```
1. class MockBaseHttpResolverClass extends BaseHttpResolver {
2. }
3.
4. export class BaseHttpResolverTest extends UnitTestRunner {
5.
6. public testConstructor() : void {
7.   assert.resolveEqual(MockBaseHttpResolverClass,
8.    "<br/>This is abstract BaseHttpResolver, " +
9.    "which has been executed at: InternTestLoader");
10.  }
```

Testovaná třída BaseHttpResolver je abstraktní, proto musí být nahrazena Mock objektem. Ověřujeme, zda po vytisknutí do prohlížeče je obsah třídy očekávaný.

```
1. public testgetPageBody2() : void {
2.     const data : ArrayList<any> = new ArrayList<any>();
3.     data.Add("fileMoved/301_newLink",
4.                                HttpRequestConstants.HTTP301_LINK);
5.     data.Add(true, HttpRequestConstants.ASYNC_SESSION);
6.     const args : AsyncRequestEventArgs =
7. new AsyncRequestEventArgs(
8.                     HttpManager.Request().getScriptPath(),data);
9.
10.  assert.equal(args.POST().getItem(
11.   HttpRequestConstants.HTTP301_LINK), "fileMoved/301_newLink");
12.    assert.resolveEqual(Http301MovedPage, "" +
13.       "<head></head>" +
14.       "<body>" +
15.       "<div id=\"Content\"><span guitype=\"HtmlAppender\">" +
16.       "<br><h1>HTTP status 301:</h1>\n" +
17.       "<h2>File has been moved. New address is:</h2>\n" +
18.   "<a href=\"#fileMoved/301_newLink\">http://localhost:8888/" +
19.  "InternEnvironmentHelper.js#fileMoved/301_newLink</a>" +
20.          "</span></div>" +
21.          "</body>", args);
22.  }
23.
24.  protected setUp() : void {
25.  this.setUrl(http://localhost:8888/InternEnvironmentHelper.js
26.  );
   }
```

Testovací případ testuje, zda je obsah stránky po přesměrování očekávaný

assert.ok(value,[message])

value <any>

message <any>

Testuje, zda je hodnota pravdivá. Viz následující dva testovací případy. Pokud hodnota není pravdivá AssertionError vyhodí zprávu rovnající se hodnotě parametru zprávy (message). Není-li parametr zprávy definován, je přiřazena výchozí chybová zpráva.

```
1. public testgetBrowserVersion() : void {
2.     const request : HttpRequestParser = new HttpRequestParser();
3.     assert.ok(request.getBrowserVersion() >= -1);
4. }
```

```
1. public testRefreshWithoutReload() : void {
2.     Loader.getHttpManager().RefreshWithoutReload();
3.     assert.ok(Loader.getHttpManager().getProcessTime() < 5);
4.     assert.deepEqual(Loader.getHttpManager().getRequest(),
5.     Loader.getHttpResolver().CreateRequest());
6. }
```

assert.onGuiComplete([instance], block, block, [owner]) : void

Asynchronní test pro simulaci interakce s prvkem. Účelem testu je dostat testovaný prvek do aktivního stavu. Metody jsou volány ve zpětném volání. Důvodem je simulovat skutečné chování prvku v prohlížeči a musíme tedy počkat na to, až se prvek vykreslí a zinicializuje pro interakci s uživatelem. Asynchronní test je tedy nutný, protože inicializace prvků jsou obecně provedené ve WUI Frameworku asynchronně.

```
1.  public testConfigurationNext() : IUnitTestRunnerPromise {
2.      return ($done : () => void) : void => {
3.          const groupsargs : BaseGuiGroupObjectArgs = new MockBaseGuiGroupObjectArgs();
4.          groupsargs.Visible(false);
5.          groupsargs.TitleText("text");
6.          groupsargs.Value("testValue");
7.          const guigroup : BaseGuiGroupObject = new MockBaseGuiGroupObjectSecond(groupsargs,
8.                                                                                 "454");
9.          guigroup.InstanceOwner(new MockBaseViewer());
10.         StaticPageContentManager.BodyAppend(guigroup.Draw());
11.         StaticPageContentManager.Draw();
12.         guigroup.Enabled(true);
13.         guigroup.Visible(true);
14.         guigroup.Value("VALUE");
15.         guigroup.Title().Text();
16.         guigroup.Changed();
17.         guigroup.Height(100);
18.         guigroup.DisableAsynchronousDraw();
19.
20.         assert.onGuiComplete(guigroup,
21. () : void => {
22.                 assert.patternEqual(guigroup.Draw(),
23.                     "\r\n<div class=\"ComWuiFrameworkGuiPrimitives\">\r\n" +
24.                     "   <div id=\"454_GuiWrapper\" guiType=\"GuiWrapper\">\r\n" +
25.                     "      <div id=\"454\" class=\"BaseGuiGroupObject\" style=\"display:" +
26.                       "block;\">\r\n" +
27.                     "         <div id=\"454_Envelop\" class=\"BaseGuiGroupObject\">\r\n" +
28.                     "            <div class=\"ComWuiFrameworkGuiPrimitives\">\r\n" +
29.                     "               <div id=\"id51_GuiWrapper\" guiType=\"GuiWrapper\"
30.                                         class=\"UserControlInput\">\r\n" +
31.                     "                  <div id=\"id51\" class=\"FormsObject\" style=" +
32.                                           "\"display:
33.                                           none;\"></div>\r\n" +
34.                     "               </div>\r\n" +
35.                     "            </div>\r\n" +
36.                     "         </div>\r\n" +
37.                     "      </div>\r\n" +
38.                     "   </div>\r\n" +
39.                     "</div>"
40.                 );
41.             },
42.             () : void => {
43.                 assert.equal(guigroup.Configuration(), groupsargs);
44.                 this.initSendBox();
45.                 $done();
46.             });
47.     };
48. }
```

První parametr je instance objektu, který chceme interaktivně testovat. Druhý parametr je zpětné volání ve kterém objekt nastavíme do požadovaného stavu. Ve třetím parametru je objekt testován.

```
1. public testsetDimensionSecond() : IUnitTestRunnerPromise {
2.     return ($done : () => void) : void => {
3.      const gui : GuiCommons = new MockGuiCommons("id6");
4.      const cropbox : CropBox = new CroBox(CropBoxType.GENERAL,
5.      gui, "id8");
6.         cropbox.Visible(true);
7.         cropbox.Enabled(true);
8.         const viewer : BaseViewer = new BaseViewer();
9.         assert.onGuiComplete(cropbox,
10.              () : void => {
11.                  cropbox.InstanceOwner(viewer);
12.                  cropbox.setDimensions(1, 100, 100, 1);
13.                  cropbox.setDimensions(1, 200, 200, 1);
14.          assert.equal((<any>cropbox).topOffset.Top(), 200);
15.          assert.equal((<any>cropbox).topOffset.Left(), 1);
16.          assert.equal((<any>cropbox).bottomOffset.Top(), 1);
17.          assert.equal((<any>cropbox).bottomOffset.Left(), 200);
18.              }, $done, viewer);
19.      };
20. }
21.
22. protected tearDown() : void {
23.     this.initSendBox();
24. }
```

Asynchronní test se provádí pomocí modulu assert.onGuiComplete(), kterým jsou testovány funkce komponenty cropbox.

5.2 Testování třídy Reflexe

Schopnost programu zjišťovat informace o objektech a jejich třídách za běhu. Technika se používá zřídka při psaní běžného programu, ale má velkou roli v rodičovských třídách, které potřebují k objektům přistupovat s vysokou abstrakcí. Reflexe je schopnost jazyka kontrolovat a dynamicky volat třídy, metody, atributy atd. za běhu. Reflexe je důležitá, protože umožňuje psát programy, které nemusejí "znát" vše v době kompilace, což je činí dynamičtějšími, protože mohou být spojeny za běhu. Kód lze zapisovat na známé rozhraní, ale skutečné třídy, které je třeba použít, mohou být distancovány pomocí reflexe programově za běhu. Reflexe se používá k deserializaci ze silně typové serializace. Využití silně typové deserializace je raritní pro JavaScript, protože se obvykle řeší na serveru. WUI Framework má možnost řešit silně typovou deserializaci přímo v prohlížeči (na klientské straně) protože pro svůj běh nevyžaduje server. Jednotka ReflexeTest obsahuje testovací případy viz příloha P I:

- **testIsMemberOf() -** metoda IsMemberOf vrací typ boolean a ověřuje na základě parametrů, že instance je instancí konkrétní třídy nebo její dítě. V testovacím případě

jsou testovány všechny kombinace parametrů, které mohou nastat na základě abstraktní rovnosti, abychom se ujistili, že metoda funguje správně. Používá se kombinace pozitivních i negativních testů, pro plné pokrytí kódu.

- **testToString()** - Reflexe má vlastní přepsanou metodu ToString(). Je použito více testovacích případů. Jsou navrženy tak, aby vrátili testovaný objekt vypsaný v HTML formátu, i jako plaintext (čistý text) formát. Výpisy slouží pro ladící účely. Před vykonáním testu je volána metoda resetCounters(), která zajišťuje ignorování počítadla.
- **testInstanceOf() -** testuje, že instance je přímo instancí třídy a ne její dítě.
- **testImplements()** - Implements() je dostupná ve vyšších programovacích jazycích a není přímo podpořena JavaScriptem. Tuto funkcionalitu dodává, WUI Framework pomocí metadat získaných parsováním TypeScriptových zdrojových souborů. Oproti InstanceOf() a MemberOf() nevyžaduje, aby cílový objekt pro porovnání existoval, k porovnání stačí pouze znalost rozhraní, které cílový objekt musí implementovat.
- **testgetProperties() -** testovací případ pro vypsání všech proměnných, které třída obsahuje do pole. Porovnání na základě hloubkové abstraktní rovnosti.
- **testgetMethods() -** funkce getMethods() vrátí všechny metody použité ve třídě rovněž jako obsah pole.

5.3 Testování třídy ObjectDecoder

Jedná se o integrační test, ve kterém se testuje spolupráce několika tříd, jednou z nich je výše zmíněná Reflexe. Testování reflexe, deserializace a serializace na reálných objektech. Simulace komunikace se serverem. Reálně se nekomunikuje, jen se simulují data, které by ze serveru přišli. Testovací případy viz příloha P II:

- **testUnserialize() -** testovací případ se zaměřuje na jednotlivé formáty, které mohou ze serveru přijít. Test kombinuje jak synchronní, tak asynchronní přístup. Kombinace těchto metod se používá zejména pro zvýšení efektivity psaní testovacích případů. Bez ohledu, jestli je testování synchronní nebo asynchronní, musí po deserializaci vyjít objekt stejného typu. Pro eliminaci chyby při zápisu testů, je použit do assert modulu stejný objekt, který je ale implementovaný pouze jednou. Asynchronní přístup se používá při zpracování enormně velkých dat. Test validuje,

že je deserializace schopna převádět data po částech se stejným výsledkem jako u synchronní deserializace.

- **testUtf8() -** testování správného zpracování různorodých jazykových sad. Testuje se správná deserializace do UTF-8. Předpokladem je, že soubor testovací jednotky je uložen ve formátu UTF-8. A proto není nutný explicitní převod expected hodnoty do UTF-8.
- **testUrl() -** správné deserializování formátu URL
- **testBase64()** - například i když jsou zprávy v JSON formátu, jsou ještě kódovány do Base64, protože mohou obsahovat hodnoty, které nejsou vhodné pro přenos přes HTTP protokol.

5.4 Testování třídy BaseObject

Abstraktní třídy jsou základní třídy, ze kterých mohou být odvozeny jiné třídy. Nemusí být vytvořeny přímo. Metody v rámci abstraktní třídy, které jsou označeny jako abstraktní neobsahují implementace a musí být implementována v odvozených třídách. Jednotka BaseObjectTest je testována pomocí MockObjektu který dědí (je odvozen) z abstraktní třídy. BaseObjectTest (viz příloha P III) obsahuje zajímavé testovací případy jako je:

- **testSerializationData() -** ve kterém se testuje proces převodu stavu objektu do formátu JSON, který může být použit mimo jiné pro přenos po síti. Testuje se hloubkovou abstraktní rovností. SerializationData respektuje seznam, který má vyloučit prvky pro serializaci. Testovacím případem se testuje hlavně schopnost vyrovnat se s rekursivní serializací.
- **testUID() - t**estuje, že každé volání UID() vrací unikátní řetězec.
- **testgetHash() -** testuje vrácení CRC (kontrolního součtu) metodou getHash() vypočítanou z dat, které představují aktuální objekt. Testovací případ zahrnuje i prázdný objekt, který musí vrátit 0. Pro ujištění se, že dva stejné objekty mají stejné hash kódy je tentýž test použit dvakrát.

5.5 Testování třídy PersistenceFactory

Session (relace) persistence odkazuje na směrování požadavků klienta na stejný webový server nebo na aplikaci serveru po dobu trvání „session" nebo času potřebného k dokončení úkolu nebo transakce. Typickou relací je přihlášení na webovou stránku, dokud je uživatel přihlášen, zaměřuje se na informace o přihlášení. Persistence obsahuje testovací případy viz příloha P IV:

- **testgetPersitence()** - který testuje, že při zavolání metody getPeristence() s předáním parametrů typu persistence a vlastníka persistence, vrátí instanci správného handleru. Což umožňuje odesílat a zpracovávat zprávy a spouštěné objekty spojené s frontou zpráv vlákna.
- **testgetPersistenceById() -** testuje, zda nám funkce getPersistenceById() vrátí instanci odpovídajícího handleru na základě „sessionId()."
- **testgetHttpSessionId() -** zjišťuje, jestli je schopen získat a naparsovat (převod z textové podoby na nějaký specifický typ, např. číslo) SessionId z URI.
- **testgetPesistenceType() -** testuje abstraktní rovností zda funkce vrací na základě vytvořené instance handleru typ persistence. Testovací případ testuje všechny možné instance.
- **testDestroyAll()** - v testovacím případu je použit předdefinovaný getter. Testuje se v headless modu (virtualizace prohlížeče na serveru pomocí JSDOM). Používá se defaultně storage. To, co chceme otestovat je situace s cookies. Situaci obejdeme tím způsobem, že se vzala reflexe z JavaScriptu a zneužila se pro podstrčení zamýšlené hodnoty. Dále pak na utvořenou persistenci voláme funkci destroyAll() a ověřujeme, že veškerá persistence daného vlastníka, jehož jsme předali parametrem je smazána. Nakonec smažeme privátní proměnnou persistence a ověřujeme, že zavolání funkce destroyAll() nevyhodí výjimku.

5.6 Testování třídy ExceptionManager

Třída ExceptionManager slouží k manipulaci s výjimkami. Výjimka je událost chyby, ke které může dojít během provádění programu a narušuje jeho normální tok. Kdykoliv dojde k chybě při provádění příkazu, vytvoří se objekt výjimky. Objekt výjimky obsahuje

velké množství ladicích informací, jako jsou metody hierarchie (stacktrace), řádek, kde došlo k výjimce, soubor, typ výjimky atd. Testovací případy viz příloha P V:

- **testThrowMessage(), testThrowError(), testThrowExit() -** ExceptionManager umožňuje vytvořit uživatelské výjimky, které nejsou přímo podpořeny ve starších verzích JavaScriptu. Jednotlivé testovací případy validují, zda lze vyhodit výjimku požadovaného typu.
- **testIsNativeException() -** validuje, jestli se jedná o nativní výjimku nebo uživatelskou.

InitSandBox() se zpracovává automaticky. V try-catch bloku se řeší, jestli vše ExceptionManager správně zaregistroval a zpracoval.

5.7 Testování třídy Http404NotFoundPage

Chyba 404 je stavový kód protokolu HTTP, což znamená, že stránka, kterou jste se pokoušeli oslovit na webu, nebyla na serveru nalezena. Testování této třídy zahrnují dva testovací případy viz příloha P VI

- **testgetPageBody()** - synchronní testování, obsah třídy se vytiskne do prohlížeče. Porovnání na základě řetězce.
- **testgetPageBody2()** - testuje, že po přesměrování a vytisknutí do prohlížeče se obsah elementu "body" liší a uvádí očekávanou URL adresu.

5.8 Testování třídy ThreadPool

Vlákna, která se starají o vykonání úkolů. Smyslem je odstranit náklady za neustálou inicializaci vláken. Testovací případy viz příloha P VII:

- **testAddThread()** - tento testovací případ testuje synchronně, zda nenastavené vlákno skutečně neběží. Poté je vlákno přidáno a testujeme, že bylo spuštěné a běží.
- **testRemoveThread()** - funkce RemoveThread() zajišťuje ukončení úkolu a vrácení vlákna. Pomocí funkce IsRunning(), zjišťujeme, že vlákno neběží.
- **testIsRunning()** - asynchronní testování, že aplikace je schopna otestovat běh vláken v průběhu jejich vykonávání. Vykonání metody $done() ověřuje správnost assertu.

After() - po otestování celé jednotky musíme zajistit, že vlákna byla vyjmutá, aby byla zajištěná integrita ostatních testů.

5.9 Implementace systémových testů

Systémové testování zatím není plně automatizované. Pouze v některých částech projektu. Implementaci systémových testů provádí externí testovací team, vzhledem ke komplexitě a silné integraci do NXP prostředí, se průběžně vyvíjejí ukázkové aplikace, které tuto komplexitu redukují a současně jsou používány pro manuální testování. Kde se funkčnost aplikací testuje v prohlížeči jejich používáním a klikáním na jednotlivé komponenty (např. u komponenty Link se ověřuje, zda došlo k přesměrování po kliknutí na odkaz).

6 ZHODNOCENÍ VÝSLEDKŮ

Code coverage (pokrytí kódu testy), tato analýza vypočítá procento pokrytí, které slouží jako nepřímé měření kvality testů. Na základě těchto měření, pak můžeme vytvořit další testovací případy pro zvýšení pokrytí kódu.

all files Commons/Utils/

99.56% Statements 1344/1350 97.24% Branches 845/869 95.39% Functions 207/217 99.53% Lines 1258/1264

File		Statements		Branches		Functions		Lines	
Convert.ts		100%	208/208	99.35%	152/153	95.83%	23/24	100%	194/194
Counters.ts		100%	23/23	84.62%	11/13	88.89%	8/9	100%	17/17
Echo.ts		98.33%	59/60	96%	24/25	94.44%	17/18	98%	49/50
FileSystemFilter.ts		100%	21/21	100%	7/7	87.5%	7/8	100%	13/13
JSON.ts		99.1%	110/111	94.12%	80/85	92.86%	13/14	99.05%	104/105
LogIt.ts		100%	67/67	100%	43/43	94.44%	17/18	100%	58/58
ObjectDecoder.ts		99.55%	223/224	100%	107/107	95.65%	22/23	99.54%	217/218
ObjectEncoder.ts		100%	213/213	96.52%	111/115	96.15%	25/26	100%	206/206
ObjectValidator.ts		100%	56/56	100%	49/49	95%	19/20	100%	50/50
Property.ts		99.3%	142/143	98.5%	131/133	93.75%	15/16	99.27%	136/137
Reflection.ts		99.11%	222/224	93.53%	130/139	100%	41/41	99.07%	214/216

Obr. 7 Zpráva pokrytí kódu

Tento přehled je automaticky generován a odeslán vždy, když je spuštěn testovací kód, a poskytuje úplné rozčlenění několika metrik pokrytí, které jsou považovány za důležité při měření pokrytí.

Statements coverage se týká nejmenších možných jednotek, které má programátor k dispozici, například přiřazení, kontrolní výrazy, návratové příkazy, smyčky a podmínky.

Branch coverage je vyhodnocení počtu logických větví v kódu. Například v bloku if-else jsou obě podmínky považovány za branch (větev).

Functions coverage se vztahuje k počtu testovaných funkcí v kódu.

```
/**
 * @method StringToBoolean
 * @memberOf Com.Wui.Framework.Commons.Utils.Convert
 * @param {string} $input Input string for conversion.
 * @return {boolean} Returns boolean representation of input string.
 */
public static StringToBoolean($input : string) : boolean {
    return WuiString.ToBoolean($input);
}

/**
 * @method BooleanToString
 * @memberOf Com.Wui.Framework.Commons.Utils.Convert
 * @param {boolean} $input Input boolean for conversion.
 * @return {string} Returns string representation of boolean input.
 */
public static BooleanToString($input : boolean) : string {
    E if (!ObjectValidator.IsEmptyOrNull($input) && ObjectValidator.IsBoolean($input)) {
        if ($input) {
            return "true";
        } else {
            return "false";
        }
    }
    return "";
}

/**
 * @method FunctionToString
 * @memberOf Com.Wui.Framework.Commons.Utils.Convert
 * @param {Function} $input Input function for conversion.
 * @param {boolean} [$multiline=false] Specify, if new lines in function declaration should be preserved.
 * @return {string} Returns string representation of function input.
 */
public static FunctionToString($input : any, $multiline : boolean = false) : string {
    let output : string = "";
    if (ObjectValidator.IsFunction($input)) {
        if (ObjectValidator.IsSet($input.toSource)) {
            output = $input.toSource();
        }

        if (WuiString.IsEmpty(output) || WuiString.Contains(output, "\n")) {
```

Obr. 8 Zpráva pokrytí kódu jednoho souboru

Lines coverage je přímočaré, je to počet řádků kódu, které testy vyhodnotily.

Z Obr. 8 vidíme vysoká procenta pokrytí. Zlepšováním práce a opakovaným testováním jsme dospěli k závěru, že právě jeden testovací případ pro každou funkci je tajemství k dosažení vysokého procentuálního výsledku. Je dobré se vyhnout dlouhým testovacím případům, které se pokouší pokrýt značnou část programu. Dobré pokrytí dává vývojářům jistotu, že celý jejich projekt je dobře rozvinutý a udržovaný, takže je snadné určit s vysokou mírou jistoty, zda nedávný refactoring (disciplinovaný proces provádění změn v softwarovém systému takovým způsobem, že nemají vliv na vnější chování kódu, ale vylepšují jeho vnitřní strukturu s minimálním rizikem vnášení chyb) nebo doplnění zdrojového kódu něco poškodilo.

ZÁVĚR

Hlavním cílem této práce bylo přiblížit čtenáři problematiku testování softwaru, především návrhu jednotkových testů a jejich implementace. Tohoto cíle bylo dosaženo prostřednictvím rešerše dostupných knižních a internetových zdrojů věnovaných problematice testování a využitím vhodných testovacích nástrojů a implementace testů, přičemž tyto informace byly doplněny o zkušenosti z vlastní testovací praxe.

Prvním z cílů bylo vymezení základních pojmů z oblasti testování, což bylo důležité pro pochopení dalších částí práce. Byli popsány pojmy jako testování softwaru, jednotkový test, terminologie chyby, testovací script.

Druhým dílčím cílem bylo popsání nástrojů využívaných při testování. Kapitola věnovaná testovacím nástrojům zahrnuje popis konkrétních nástrojů z různých hledisek a přiblížení některých z nich.

Dalším z cílů bylo představit a přiblížit testovaný software WUI Framework. Jsou popsány jeho vlastnosti, kvality a možnosti využití.

Dalšími cíli byly návrh struktury a metodiky pro otestování WUI Frameworku. Prostřednictvím schémat je vysvětlena struktura jednotkového testu i celé oblasti testování. Obecněji je popsána metodika testování.

Praktická část zahrnuje testovací případy. Test by měl testovat chování kódu jak za standardních situací, tak v situacích mimořádných. Je věnována pozornost ukázkovým případům a jejich použití, jelikož je na nich závislý úspěch celého testování.

Posledním cílem pak byla analýza pokrytí kódu. Kde je prezentován přehled výsledů a jejich zhodnocení.

Výsledkem diplomové práce je zvýšení pokrytí kódu z méně než 10 % na více jak 85 % na celém frameworku. Nižší skóre je způsobené chybějícími systémovými testy. Bylo napsáno více jak 2 000 testovacích případů a více než 10 000 assert modulů, které výrazně pomáhají při refactoringu. Úsilí při psaní testu se projevilo i ve formě a uhlazenosti kódu testů, protože se jedná o veřejně dostupný projekt, není vhodné publikovat nevzhledný kód.

Výsledky diplomové práce nejsou jen v akademické sféře, ale jsou denně používány v průběhu vývoje projektu.

SEZNAM POUŽITÉ LITERATURY

[1] SPILLNER, Andreas, Tilo LINZ a Hans SCHAEFER. Software Testing Foundations. O´Reilly Media, 2006. ISBN 978-3898643634.

[2] BHATNAGAR, Sachin. *All about NodeJS* [online]. 2018 [cit. 2018-02-15]. Dostupné z: https://www.udemy.com/all-about-nodejs/

[3] *Mocha* [online]. [cit. 2018-02-15]. Dostupné z: https://mochajs.org/

[4] *Chai Assertion Library* [online]. [cit. 2018-02-15]. Dostupné z: http://chaijs.com/

[5] *What is PhantomJS* [online]. [cit. 2018-02-15]. Dostupné z: https://scotch.io/tutorials/what-is-phantomjs-and-how-is-it-used

[6] *Tobias Markert* [online]. [cit. 2018-02-15]. Dostupné z: https://www.tobiasmarkert.com/using-gulp-js-for-typescript-sass-and-copying-assets/

[7] *Testing Backbone applications with Jasmine and Sinon* [online]. [cit. 2018-02-15]. Dostupné z: https://tinnedfruit.com/articles/testing-backbone-apps-with-jasmine-sinon.html

[8] *Jasmine* [online]. [cit. 2018-02-15]. Dostupné z: https://jasmine.github.io/pages/

[9] [online]. [cit. 2018-02-18]. Dostupné z: https://www.npmjs.com/package/jsdom

[10] *Intern* [online]. [cit. 2018-02-19]. Dostupné z: https://theintern.io/docs.html#Intern/4/docs/README.md

[11] *Node.js* [online]. [cit. 2018-02-23]. Dostupné z: https://nodejs.org/api/assert.html

[12] *Software testing fundamentals* [online]. [cit. 2018-03-01]. Dostupné z: http://softwaretestingfundamentals.com/test-case/

[13] DENICOLA, Domenic. [online]. [cit. 2018-03-08]. Dostupné z: https://github.com/jsdom/jsdom/wiki/jsdom-vs.-PhantomJS

[14] PATTON, Ron. *Testování softwaru*. Praha. Computer Press, 2002. ISBN 80-7226-636-5.

[15] CIESLAR, Jakub. *WUI FRAMEWORK*. NXP Semiconductor, 2017.

[16] HARTIKAINEN, Jani. *Promises in JavaSript Unit Tests: the Definitive Guide* [online]. [cit. 2018-03-08]. Dostupné z: https://www.sitepoint.com/promises-in-javascript-unit-tests-the-definitive-guide/

[17] Artifact [online]. [cit. 2018-04-14]. Dostupné z: http://metodikamezitest.asp2.cz/Metodika_testovani/workproducts/testovaci_skript_FA297F13.html

[18] KITNER, Radek. [online]. [cit. 2018-04-18]. Dostupné z: http://kitner.cz/testovani_softwaru/prehled-testovacich-technik/

[19] *NORMY.biz* [online]. [cit. 2018-04-18]. Dostupné z: https://shop.normy.biz/detail/97596

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

API	Rozhraní pro programování aplikací
CPU	Centrální procesorová jednotka
CSS	Kaskádové styly
DOM	Objektový model dokumentu
GUI	Grafické uživatelské rozhraní
BDD	Vývoj řízený chováním
HTML	Hypertextový značkovací jazyk
HTTP	Protokol hypertextových dokumentů
IEC	Mezinárodní elektrotechnická komise
IEEE	Institut pro elektrotechnické a elektronické inženýrství
JSDOM	JavaScript implementace mnoha webových standardů
ISO	Mezinárodní organizace pro normalizaci
JSON	JavaScriptový objektový zápis
MVC	Ovládání modelu prohlížeče
NodeJS	Vysoce výkonné prostředí pro Javascript
OOP	Objektově orientované programování
PoO	Bod pozorování
PoC	Kontrolní bod
RAM	Paměť s přímým přístupem
SVG	Škálovatelná vektrorová grafika
UAT	Jednotkové automatizované testy
URI	Univerzální identifikátor zdroje
URL	Jednotný popis umístění zdroje
UTF	Formát transformace Unicode

WUI	Webové uživatelské rozhraní
XML	Rozšiřitelný značkovací jazyk
XP	Extrémní programování

SEZNAM OBRÁZKŮ

SEZNAM PŘÍLOH

P I https://bitbucket.org/wuiframework/com-wui-framework-commons/raw/73319d327cd443725fd1ddc94a2c7465b5c22da7/test/unit/typescript/Com/Wui/Framework/Commons/Utils/ReflectionTest.ts

CD – test/unit.typescript/Com/Wui/Framework/Commons/Utils/ReflectionTest.ts

P II https://bitbucket.org/wuiframework/com-wui-framework-commons/raw/73319d327cd443725fd1ddc94a2c7465b5c22da7/test/unit/typescript/Com/Wui/Framework/Commons/Utils/ObjectDecoderTest.ts

CD - test/unit.typescript/Com/Wui/Framework/Commons/Utils/

ObjectDecoderTest.ts

P III https://bitbucket.org/wuiframework/com-wui-framework-commons/raw/73319d327cd443725fd1ddc94a2c7465b5c22da7/test/unit/typescript/Com/Wui/Framework/Commons/Primitives/BaseObjectTest.ts

CD - test/unit.typescript/Com/Wui/Framework/Commons/Primitives/

BaseObjectTest.ts

P IV https://bitbucket.org/wuiframework/com-wui-framework-commons/raw/73319d327cd443725fd1ddc94a2c7465b5c22da7/test/unit/typescript/Com/Wui/Framework/Commons/PersistenceApi/PersistenceFactoryTest.ts

CD - test/unit.typescript/Com/Wui/Framework/Commons/PersistenceApi/

PersistenceFactoryTest.ts

P V https://bitbucket.org/wuiframework/com-wui-framework-commons/raw/73319d327cd443725fd1ddc94a2c7465b5c22da7/test/unit/typescript/Com/Wui/Framework/Commons/Exceptions/ExceptionsManagerTest.ts

CD - test/unit.typescript/Com/Wui/Framework/Commons/ Exceptions/

ExceptionsManagerTest.ts

P VI https://bitbucket.org/wuiframework/com-wui-framework-commons/raw/73319d327cd443725fd1ddc94a2c7465b5c22da7/test/unit/typescript/Com/Wui/Framework/Commons/ErrorPages/Http404NotFoundPageTest.ts

CD - test/unit.typescript/Com/Wui/Framework/Commons/ErrorPages/

Http404NotFoundPageTest.ts

P VII https://bitbucket.org/wuiframework/com-wui-framework-commons/src/73319d327cd443725fd1ddc94a2c7465b5c22da7/test/unit/typescript/Com/Wui/Framework/Commons/Events/ThreadPoolTest.ts?at=2018.1.1&fileviewer=file-view-default

CD - test/unit.typescript/Com/Wui/Framework/Commons/Events/ThreadPoolTest.ts