VYSOKÁ ŠKOLA POLYTECHNICKÁ JIHLAVA

Katedra technických studií

Jednotkové testy, význam a využití v praxi

bakalářská práce

Autor práce: Martin Sládek

Vedoucí práce: Ing. Marek Musil

Jihlava 2021

Vysoká škola polytechnická Jihlava

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Autor práce:	Martin Sládek
Studijní program:	Elektrotechnika a informatika
Obor:	Aplikovaná informatika
Název práce:	Jednotkové testy, význam a využití v praxi
Cíl práce:	Cílem práce je představit unit testování hotového kódu ve frameworku cppunit. Popsat jednotkové testy (unittest), uvést jejich výhody a nevýhody, dále pak pro unittest představit význam a praktické užití formou ukázkových příkladů. Výstup práce bude doplněn příklady testovacího kódu, které budou ukazovat, jak testování kontroluje správný chod aplikace.

Ing. Marek Musil
vedoucí bakalářské práce

doc. Ing. Zdeněk Horák, Ph.D.
vedoucí katedry
Katedra technických studií

Abstrakt

Tato bakalářská práce se věnuje problematice unit testování. V první části práce je popsáno, co je to unit test, jaké náležitosti musí splňovat, jaké jsou výhody a nevýhody unit testování a proč je důležité unit testování používat. Ve druhé části jsou představeny konkrétní technologie unit testování včetně jejich vlastností. Práce se zaměřuje především na frameworky JUnit a CppUnit. V rámci praktických ukázek kódu je ukázáno využití testování v praxi.

Klíčová slova

automatizované testování, CppUnit, jednotkové testy, JUnit, unit test, testovací framework

Abstract

This bachelor thesis deals with the topic of unit testing. The first part of the thesis defines what a unit test is and what a good unit test is made of, what the pros and cons of unit testing are and why unit testing is important. In the second part of the thesis, specific technologies for unit testing are described, including their properties. The thesis focuses mainly on the JUnit and CppUnit frameworks. As a part of practical demonstrations of the code, practical usage of testing in practice is shown.

Key words

automated testing, CppUnit, JUnit, unit testing, unit testing frameworks

Prohlašuji, že předložená bakalářská práce je původní a zpracoval/a jsem ji samostatně. Prohlašuji, že citace použitých pramenů je úplná, že jsem v práci neporušil/a autorská práva (ve smyslu zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů, v platném znění, dále též „**AZ**“).

Souhlasím s umístěním bakalářské práce v knihovně VŠPJ a s jejím užitím k výuce nebo k vlastní vnitřní potřebě VŠPJ.

Byl/a jsem seznámen/a s tím, že na mou mé bakalářskou práci se plně vztahuje **AZ**, zejména § 60 (školní dílo).

Beru na vědomí, že VŠPJ má právo na uzavření licenční smlouvy o užití mé bakalářské práce a prohlašuji, že **s o u h l a s í m** s případným užitím mé bakalářské práce (prodej, zapůjčení apod.).

Jsem si vědom/a toho, že užít své bakalářské práce či poskytnout licenci k jejímu využití mohu jen se souhlasem VŠPJ, která má právo ode mne požadovat přiměřený příspěvek na úhradu nákladů, vynaložených vysokou školou na vytvoření díla (až do jejich skutečné výše), z výdělku dosaženého v souvislosti s užitím díla či poskytnutím licence.

V Jihlavě dne 28. dubna 2021

…….
Podpis studenta

Poděkování

Na tomto místě bych chtěl poděkovat mému vedoucímu práce panu Ing. Marku Musilovi za oborné vedení práce a cenné rady, které mi pomohli práci zkompletovat.

Obsah

Seznam obrázků

Seznam použitých zkratek

TDD *Test-Driven Devolopment* (vývoj řízený testy)

Úvod

Unit testování patří k důležité části vývoje softwaru. Čím větší je projekt a tým, který na projektu pracuje, tím důležitější je mít napsané automatizované testy, které po jakékoliv změně zkontrolují správný chod programu.

Myslím si, že problematika jednotkových testů je užitečná a důležitá, velmi pomůže při vývoji software a garantuje kvalitu softwaru. Patří to mezi sofistikované nástroje. Poprvé jsem se s problematikou jednotkového testování setkal ve výuce předmětu Pokročilé programovací techniky. Mám zkušenosti s testováním v Javě a měl jsem možnost se zabývat testováním v rámci praxe.

Automatizované testování programátorům dokáže ušetřit spoustu času při hledání chyb, a to v praxi znamená i ušetření spoustu peněz. Proto jsem se rozhodl tomuto tématu v rámci bakalářské práce věnovat. Mým cílem pro tuto práci je představit problematiku unit testů, vysvětlit jejich náležitosti, představit testovací frameworky včetně jejich testovacích metod a předvést praktické ukázky testovacího kódu, který jsem vytvořil během své praxe.

Textová část práce má následující strukturu. V první části je vysvětleno, co to testování je, následuje dělení testů podle fáze vývoje. Je důležité se zamyslet nad tím, kdy psaní jednotkových testů má smysl a kdy to význam nemá, proto se v práci věnuji jednotlivým výhodám a nevýhodám psaní unit testů. V další kapitole představuji, jak má kvalitní a důvěryhodný unit test vypadat včetně všech náležitostí. Poslední část teoretické části se zabývá dělením testování dle znalosti kódu. Ve druhé části jsou představeny testovací frameworky. Práce se zaobírá frameworky rodiny xUnit, nejvíce pozornosti je věnováno frameworkům CppUnit a JUnit. Rozebrána je jejich architektura a testovací metody. V poslední části práce je demonstrováno praktické využití unit testů v praxi.

V současné době je o jednotkovým testování vydáno několik publikací. První publikaci, se kterou jsem se setkal, hodnotím velice kladně, je určena pro začátečníky v problematice jednotkových testů. Jde o knihu The art of Unit testing, kterou napsal Roy Osherove. Tato kniha má tři edice, které se liší zaměřením na jiný programovací jazyk. Mezi další odborné publikace, které pojednávají o problematice jednotkového testování patří xUnit Test Patterns: Refactoring Test Code od Gerarda Meszarose. Tato kniha si klade za cíl naučit čtenáře zacházet s testovacími nástroji xUnit frameworků. Frameworky

pro psaní jednotkových testů se zabývá kniha Unit Test Frameworks, kterou napsal Paul Hamil. Kniha, která se specializuje na psaní jednotkových testů ve frameworku JUnit se jmenuje Pragmatic Unit Testing in Java with JUnit. Knihu napsali Andy Hunt a Dave Thomas. V knize Working Effectively with Legacy Code od Michaela C. Featherse je řešena problematika úpravy jakéhokoliv kódu. V knize je popsána důležitost testování.

1 Co je testování?

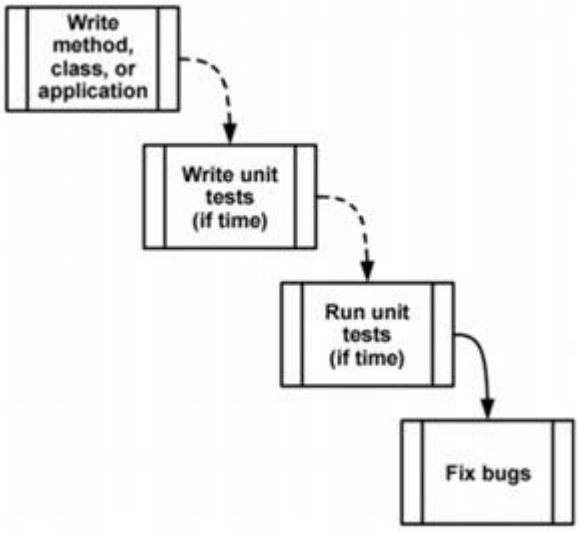
Obecně testování je činnost, která vede k odhalení chyb. Vývojáři softwaru se pomocí testování snaží odhalit všechny chyby napsaného kódu. Cílem testování je potvrzení toho, že program splňuje zadané požadavky (Rajkumar, 2021).

Dříve se testovalo v poslední fázi vývoje softwaru a testování se nepřidávala velká váha. Nyní se čím dál častěji objevuje testování ve všech fázích vývoje. Výsledek testu poskytuje vývojáři garanci správné funkčnosti software a dává mu jistotu (sebevědomí) pro další práci.

Tato kapitola představuje problematiku testování a popisuje jednotlivé druhy testu. Zabývá se vizualizací testu, která je pro testování velmi důležitá.

1.1 Vývoj řízený testy

Vývoj řízený testy z anglického Test-Driven Devolopment, zkráceně TDD je přístup k vývoji softwaru, který se charakterizuje tím, že se jednotkové testy píšou ještě dříve než metoda, kterou je zapotřebí otestovat. Patří mezi jednu z nejpoužívanějších metod v extrémním programování (Kulkarni, nedatováno).



Obrázek 1 - Tradiční cesta psaní unit testů

(Zdroj: (Osherove, c2014))

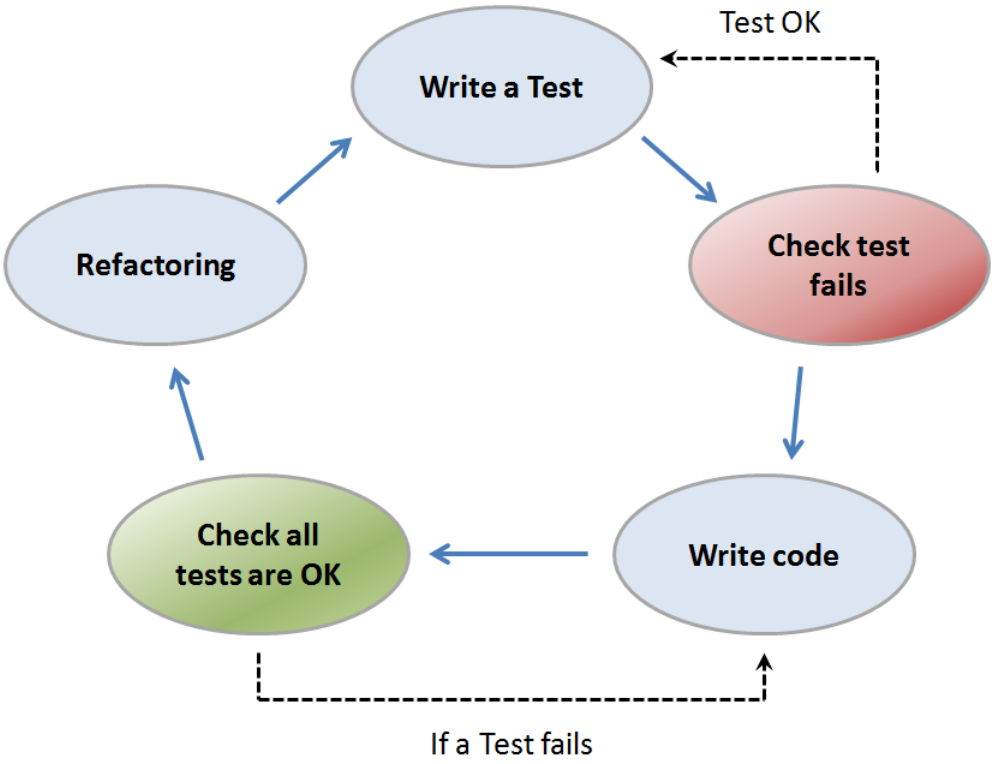
Test-Driven Development začíná návrhem a vývojem testů pro každou malou funkci aplikace. Metoda TDD říká vývojářům, aby napsali nový kód pouze v případě, že selhal jednotkový test. Tím se zabraňuje duplicitnímu kódu (Kulkarni, nedatováno).

Tři kroky TDD cyklu

TDD cyklus je rozdělen do tří podstatných kroků, které jsou popsány v (Hamill, 2004).

- Prvním krokem je napsat test, spustit jej a ověřit, že test selhal.
- Vytvořit minimální množství kódu tak, aby test prošel. Může se jednat o naprosto triviální implementaci jako např. vrácení konstantního čísla.
- Jakmile test projde, ověří se tím jak kód, tak test. Zbývá už jen kód refaktorovat.

Metoda TDD je znázorněna na obrázku č. 2.



Obrázek 2 - Metoda TDD

(Zdroj: https://medium.com/@dxshim90/tdd-test-driven-development-21d959138344)

Výhody TDD oproti tradičnímu testování

Během tradičního testování[1] správně napsaný test najde jednu nebo více závad. U metody TDD je to stejné. Pokud test neprojde, programátor již udělal pokrok, protože ví, že je třeba problém vyřešit.

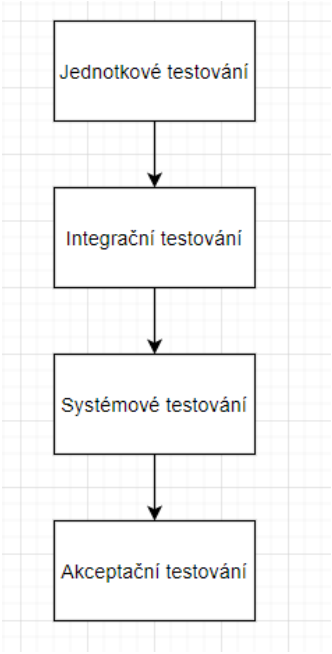
TDD má oproti tradičnímu testování tyto výhody.

- TDD zajišťuje, že aplikace splňuje definované požadavky.
- TDD se zaměřuje spíše na kód zajišťující chod programu, který ověřuje, že napsané testy fungují správně. Tradiční testování se zaměřuje více na návrh a kód testu.
- Aplikováním testů lze dosáhnout mnohem většího pokrytí kódu. (Kulkarni, nedatováno)

1.2 Úrovně testování a druhy testů

Testy lze rozdělit do několika skupin – implementace testů ze znalosti kódu, testování podle fáze vývoje, testování uživatelského rozhraní bez znalosti kódu. V této kapitole je představeno dělení testů podle fáze vývoje aplikace. Jde o čtyři fáze testování aplikace, než se finální produkt bude moci uvolnit pro použití (Eriksson, 2014).

[1] V tradičním testování se nejdříve implementuje metoda a poté se teprve testuje její funkčnost.



Obrázek 3 - Diagram úrovní testování

(Zdroj: vlastní)

Jednotkové testy

Jednotkový test je část kódu (obvykle metoda), která invokuje jinou část kódu a ověřuje správnost jednoho konkrétního výsledku. Jednotkový test je téměř vždy napsán za použití testovacího frameworku (Osherove, c2014).

Jednotkové testování neboli unit testing je nejzákladnější typ testování. Provádí se ihned po napsání nové třídy nebo metody, vytváří jej programátoři (Hlava, 2011).

Cílem je otestovat odizolovanou část kódu a ověřit její funkčnost (Hlava, 2011).

Integrační testy

Když je základní vývoj aplikace hotov, tak přichází na řadu integrační testování. Na vývoj integračních testů bývá většinou vymezen speciální tým, který ověřuje bezchybnou komunikaci mezi jednotlivými komponentami uvnitř aplikace (Eriksson, 2014) (Hlava, 2011).

Také se testuje integrace mezi softwarem a hardwarem. Např: mezi operačním systémem a komponentou. Tyto testy již nemusí být všechny automatizované, dochází i na manuální testování. Integrační testování se používá spíše u velkých projektů (Eriksson, 2014) (Hlava, 2011).

Systémové testy

Další úroveň testování je systémové testování. Všechny komponenty jsou testovány jako celek, aby bylo zajištěno, že vytvořený produkt vyhovuje všem zadaným požadavkům.

Simulují se všechny různé případy, které mohou při používání aplikace nastat. Testuje se tedy nestandardní chování, aby se ověřilo, že všechny chybné výstupy jsou správně ošetřeny (Eriksson, 2014) (Hlava, 2011).

Jakmile aplikace projde systémovým testováním, tak je téměř připravena k zveřejnění anebo prodeji (Eriksson, 2014) (Hlava, 2011).

Akceptační testy

Poslední úrovní vývoje je akceptační testování. Jde o akceptační testování ze strany zákazníka. Zákazník s jeho týmem odborníků provede testy na základě připravených scénářů. Cílem je otestovat správnou funkčnost všech potřeb zákazníka. Pokud dojde k odhalení nových chyb, nahlásí se zpráva o chybách vývojovému týmu. Nejdůležitější je co nejrychleji opravit nalezené chyby a předat opravenou aplikaci zákazníkovi k dalším testům. Kdyby se chyb objevilo více, než je vývojový tým schopný v krátkém časovém úseku opravit, dojde ke zpoždění termínu nasazení do softwarového provozu. Při větších komplikacích může dojít k fatálním následkům. (Hlava, 2011).

1.3 Výhody a nevýhody unit testování

Výhody

1. Zefektivnění práce

Jednou z hlavních výhod unit testování je, že zefektivní celý proces vytváření projektu nebo aplikace (Novoseltseva, 2017).

Pokud programátor chce do programu přidat nějaké nové prvky nebo funkce, je zapotřebí učinit v kódu některé změny. Změna již funkčního kódu je ale riskantní a při špatné úpravě i časově náročná.

Když je jednotkový test správně napsaný, tak při každé změně kódu programátor zjistí, jestli se funkčnost programu nezměnila. (Novoseltseva, 2017).

2. Kvalita kódu

Psaní jednotkových testů nutí programátora se daleko více zamyslet nad všemi problémy a chybami, které se mohou vyskytnout (Novoseltseva, 2017).

3. Refaktorizace a údržba kódu

Jednotkové testování povoluje vývojářům kdykoliv refaktorizovat nebo pozměnit kód bez jakékoliv obavy, že by se při změně kódu něco pokazilo v jiné části projektu (Novoseltseva, 2017).

4. Testy pomáhají v orientaci

Unit testy lze brát jako součást dokumentace. Vývojáři, kteří se snaží porozumět funkčnosti dané metody se mohou podívat do jednotkového testu, aby dostali základní povědomí o vstupech a výstupech testované metody (Novoseltseva, 2017).

5. Redukce nákladů

Na chyby v kódu se přijde, jakmile se unit testy spustí. To pro vývojáře znamená spoustu ušetřeného času a trpělivosti. Pokud by vývojář, který pravidelně netestuje, našel chybu v závěrečná fázi vývoje aplikace, je vysoce pravděpodobné že opravení takové chyby bude velmi časově náročné (Novoseltseva, 2017).

Nevýhody

Unit testování není vhodné k odhalení komplexních chyb v programu. Jednotkové testy nejsou schopné zachytit všechny chyby, jelikož se každý modul testuje samostatně.

Cílem unit testování je testovat izolovaný kód, proto není možné otestovat integraci.

Může se stát, že mezi testovací platformou a platformou, ve které byla aplikace vyvíjena, budou znatelné rozdíly (*professionalqa*, 2019).

Pokud unit test nesplňuje všechna základní kritéria, která jsou zmíněny v nadcházející kapitole, nelze se na něj spolehnout.

1.4 Náležitosti unit testu a dimenze kvality testu

Jak má správně vypadat unit test?

Každý správně napsaný unit test musí být: automatický, nezávislý na pořadí spouštění a nezávislý na jiných testech a samozřejmě rychlý.

Pokud se v kódu nic nezmění, test by měl po každém spuštění ukázat stejné výsledky

Je vždy nutné testovat všechny extrémní případy.

Pokud test není vyhodnocen jako úspěšný, tak by mělo být snadný najít očekávaný výsledek a výsledek, který vyvolal chybu (Osherove, c2014).

Co musí unit test splňovat?

Aby unit test nebyl jen zbytečnou prací navíc, musí být důvěryhodný, musí být lehké se v něm orientovat, musí být udržitelný a nesmí být časově náročný.

Při vytváření testu se musí velmi pečlivě promyslet jaké vstupy a výstupy je nutné otestovat, aby test odhalil všechny možné chyby. Pokud test nepokryje všechny možnosti, stává se neefektivním a neužitečným.

Důvěryhodnost testu – Developeři si potřebují být jistý výsledkem testu. Důvěryhodné testy musí být bez chyb a testovat platné i neplatné vstupy.

Čitelnost – Kód musí být pro programátora čitelný. Pokud test neprojde, musí být lehce rozpoznatelný na jakém místě se problém objevil. Jakmile test přestává být čitelný, tak pro programátora přestává být efektivní a hledání chyby se stává časově náročným.

Udržitelnost testu – Pokud test není důvěryhodný a čitelný, stává se těžce udržitelným. Programátor přirozeně takovýto test přestane udržovat. Takový test se postupem času stává zbyteným.

Programátor díky správně napsanému testu si musí být jistý, že daná část kódu funguje správně, aby se mohl soustředit na jiné problémy (Osherove, c2014).

Dimenze kvality testu

Aby šlo lépe poznat, jestli test lze považovat za kvalitní a spolehlivý, je definovaná dimenze kvality dle normy ISO/IEC 25010:2011 (Kitner, 2015).

Mezi klíčové vlastnosti kvalitního testu patří:

Funkčnost (Functionality) – zaměřuje se na správné chování programu, kontroluje správnou funkčnost dle specifikací.

Výkonnost (Performance) – kontroluje se, zda test netrvá příliš dlouhou dobu. Také se sleduje zatížení testu na systém.

Kompatibilita (Compatibility) – možnost kombinace s jiným softwarem nebo hardwarem,

Použitelnost (Usability) – Hodnotí, jestli je test použitelný, uživatelsky přívětivý, jasný, čitelný a hodnotí se jeho dokumentace.

Spolehlivost (Reliability) – kontroluje chování testů při výskytu nečekané chyby nebo problému. Test musí být schopný chybu odhalit a zapsat.

Udržitelnost – Hodnotí, zda se dá systém refaktorizovat, předělat nebo aktualizovat na vyšší verzi.

Přenositelnost – zaměřuje se na funkčnost testů na jiném zařízení (Kitner, 2015).

1.5 Izolace Unit testů

Jednotkové testy musí být vždy naprosto izolované, jinak se nejedná o unit test. Třída, která je testovaná musí využívat pouze svůj vlastní kód, nesmí načítat data z externího zdroje.

Při testování velkého množství kódu, který není od izolovaný se mohou objevit tyto problémy:

Error location (chybná lokace) - Je složité zjistit a najít příčinu chyby. Aby tester byl schopný odhalit důvod a původ chyby, musí se podívat na všechny testovací vstupy, zkontrolovat kudy prochází až k výstupu a zjistit, kde během této cesty program vyvolal chybu.

Execution time (doba provedení) – Větší testy se zpracovávají daleko delší dobu. To při každém spuštění vede k frustraci. Může se stát, že příliš časově náročné a dlouhé testy přestane programátor používat, což povede k těžšímu odhalení případně vzniklé chybě (Feathers, 2009).

1.6 Dělení testů podle znalosti kódu

Jednotkové testování se provádí na základě úplné znalosti kódu, existují ale i formy testování ve kterých tester nemá přístup ke zdrojovému kódu.

Black box testing

Při tomto typu testování programátor nevidí do kódu, ale používá pouze uživatelské rozhraní. Programátor tedy nezná, jakým způsobem se v kódu s daty pracuje. Správnou funkčnost posuzuje pouze na základě vstupů a výstupů (*professionalqa*, 2019).

Obrázek 4 - Schéma Black box testing
(Zdroj: https://www.testbirds.com/blog/black-box-testing/)

Výhody:

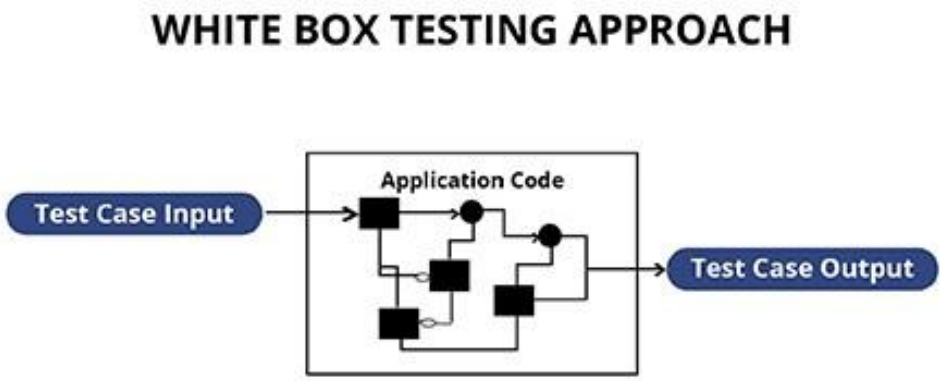
- Tester nemusí znát programovací jazyky.
- Pomáhá identifikovat neurčitost a nejednoznačnosti v uživatelském rozhraní (*professionalqa*, 2019).

Nevýhody:

- Nelze použít na komplexnější kódy a software
- Kontrolovat velké množství vstupů a výstupů je časově náročné
- Nelze se příliš spolehnout na toto testování (*professionalqa*, 2019)

White box testing

Zdrojový kód je testerům k dispozici. To testerům dovoluje zanalyzovat struktury a také implementace systému. Tester má tedy přístup jak ke zdrojovému kódu, tak i k dokumentaci příslušné aplikace. White box testing je pro jednotkové testování typický (*professionalqa*, 2019).



Obrázek 5 - Schéma White box testing
(Zdroj: https://dulaniherath.medium.com/black-box-testing-and-whitebox-testing-408dc327226b)

Výhody:

- Včasné odhalení chyb – na základě znalosti zdrojového kódu je daleko snazší odhalit chyby.
- Odhalení zbytečného kódu – programátor při zjišťování funkčnosti kódu může odhalit nadbytečný kód.
- Lehké zautomatizování testů (*professionalqa*, 2019).

Nevýhody:

- Složitost – tento typ testování vyžaduje odborné znalosti programovacího i testovacího jazyka. Také je nutné si nastudovat kód celé aplikace, aby tester byl schopný ho efektivně otestovat.
- Časová náročnost (*professionalqa*, 2019).

Greybox testing

Toto testování je kombinací předešlých dvou typů. Tester má k dispozici částečnou znalost systému, dokumentaci a zdrojové kódy, které může v případě potřeby použít.

Hlavní myšlenkou tohoto testování je, že jakmile tester ví, jak vypadá struktura programu, je schopný ho lépe otestovat z venku, tedy pomocí uživatelského rozhraní (Čermák, 2008).

Výhody:

- Inteligentní testy – Tester může pracovat mnohem efektivněji, díky znalosti kódu.

Nevýhody:

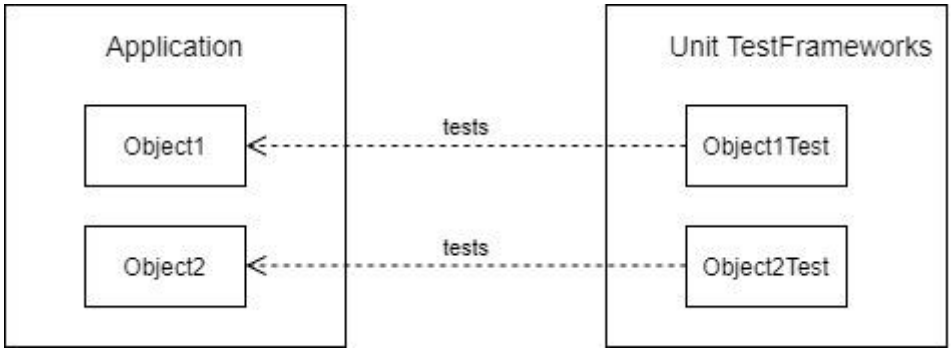
- Neúplné testování – Tester nevidí úplně všechny binární a zdrojové kódy, nemůže tedy otestovat všechny možnosti.
- Stejně jako u Black box testování se nelze úplně spolehnout na výsledky testování, pokud tester nezná celý zdrojový kód (Čermák, 2008).

2 Testovací frameworky

Při manuálním testování programátor musí vynaložit velké úsilí, aby byl schopný aplikaci správně otestovat. Při některých změnách v kódu musí svůj testovací kód upravovat a psát nový. Proto se využívají testovací frameworky, které práci zautomatizují, zefektivní a přidají nové testovací možnosti.

Frameworky pro jednotkové testování jsou softwarové nástroje, které ulehčují vytváření a spouštění unit testů. Přehledně také zpracovávají výsledky testů a zaznamenávají místo nalezené chyby (Hamill, 2004).

Jednotkové testy se píšou souběžně s vývojem aplikace, ale nejsou součástí dokončeného produktu, jak je vidět v následujícím schéma (Hamill, 2004).



Obrázek 6 - Diagram vztahu testů na aplikaci
(Zdroj: (Hamill, 2004))

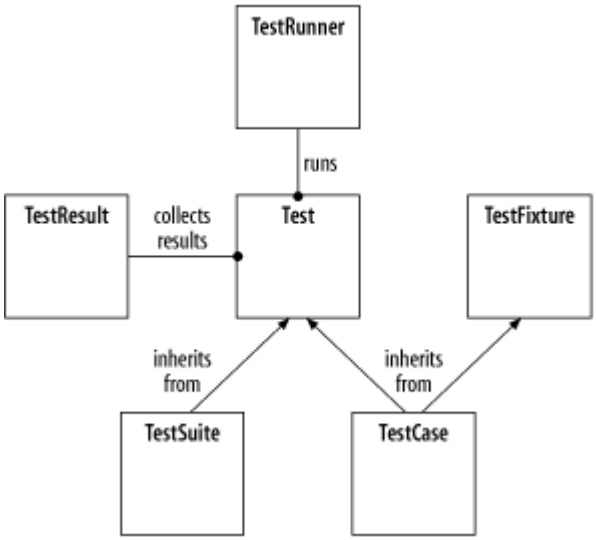
2.1 Skupina testovacích frameworků xUnit a jejich architektura

Nejprve v roce 1999 Kent Beck vytvořil testovací framework pro jazyk SmallTalk, který se nazýval SmalltalkUnit tedy SUnit. Tento framework přenesl Erich Gamma do Javy, vznik tedy velmi populární framework JUnit. Framework JUnit byl upraven pro použití v dalších programovacích jazycích a tím vzniklo mnoho dalších frameworků jako jsou: CppUnit, NUnit, PyUnit, XM-Lunit a další (Hamill, 2004).

Hlavní odlišností frameworků xUnit je použití pro jiný programovací jazyk.

2.1.1 Architektura frameworků xUnit

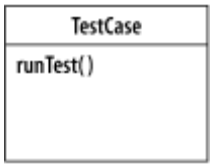
Všechny xUnit frameworky mají stejnou základní architekturu. Liší se až v podrobnější implementaci. Obecně tyto testovací frameworky obsahují klíčové třídy TestCase, TestRunner, TestFixture, TestSuite a TestResult (Hamill, 2004).



Obrázek 7 – Diagram základních tříd architektury frameworků xUnit
(Zdroj: (Hamill, 2004))

Třída TestCase

Nejzákladnější třídou pro testování je třída TestCase. Každá testovací třída musí dědit z této třídy. TestCase může být jedna podmínka, která kontroluje chod aplikace. (Hamill, 2004). Na obrázku č.8 lze vidět diagram této třídy a na obrázku č.9 je ukázán jednoduchý TestCase, který kontroluje správné počítání faktoriálu čísla 5.



Obrázek 8 – Diagram třídy TestCase
(Zdroj: (Hamill, 2004))

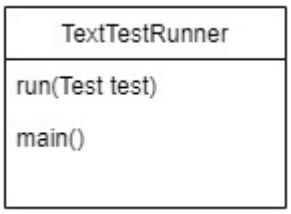
```
7⊖    @Test
8     void TestCase1()
9     {
10        GUI test = new GUI();
11        // Faktorial 5 = 120
12        double number = test.VypocitejFaktorial(5);
13        assertEquals(120,number);
14    }
```

Obrázek 9 - Ukázka kódu třídy TestCase v JUnit

(Zdroj: vlastní)

TestRunner

TestRunner poskytuje uživatelské rozhraní pro spouštění testů. Jeho úkolem je spustit jeden nebo více TestCases a ukázat jejich výsledky (Hamill, 2004).



Obrázek 10 - Diagram třídy TextTestRuner

(Zdroj: (Hamill, 2004))

TestFixture

Jedná se o funkce, které předpřipravují prostředí pro testování. Obvykle se používá při psaní testů, které se zabývají stejným objektem. Hlavní výhodou je redukce duplicitního kódu, přičemž je zachována izolace jednotkových testů. Typicky se v TestFixtures používají metody setUp a tearDown. Některé frameworky jako např. CppUnit mají přímo třídu TestFixture. Jiné frameworky např JUnit umožňují třídě TestCase se chovat jako TestFixture (Hamill, 2004).

TestSuite

Pro seskupení testCases se používá třída TestSuite. Účelem této třídy je exekuovat testy najednou. Když se spustí TestSuite, tak se spustí všechny jeho testCases (Hamill, 2004).

TestCases se přidávají do TestSuite pomocí metody addTest () (Hamill, 2004).

```
133  class Testt004PeripheralsBoard : public CPPUNIT_NS::TestFixture
134  {
135      CPPUNIT_TEST_SUITE(Testt004PeripheralsBoard);
136      CPPUNIT_TEST(TestAdcSupplyVoltage135PGet);
137      CPPUNIT_TEST(TestAdcSupplyVoltage135NGet);
138      CPPUNIT_TEST(TestAdcSupplyVoltage120PGet);
139      CPPUNIT_TEST(TestAdcSupplyVoltage120NGet);
140      CPPUNIT_TEST(TestAdcSupplyVoltage33Get);
141      CPPUNIT_TEST(TestAdcSupplyVoltage50Get);
142      CPPUNIT_TEST(TestAdcSupplyVoltage25PGet);
143      CPPUNIT_TEST(TestAdcSupplyVoltage25NGet);
144      CPPUNIT_TEST(Testt004BoardSupplyVoltageCheck);
145      CPPUNIT_TEST(Testt004BoardSupplyVoltageGet);
146      CPPUNIT_TEST_SUITE_END();
147
148  public:
149      void Testt004BoardSupplyVoltageCheck();
150      void TestAdcSupplyVoltage135PGet();
151      void TestAdcSupplyVoltage135NGet();
152      void TestAdcSupplyVoltage120PGet();
153      void TestAdcSupplyVoltage120NGet();
154      void TestAdcSupplyVoltage33Get();
155      void TestAdcSupplyVoltage50Get();
156      void TestAdcSupplyVoltage25PGet();
157      void TestAdcSupplyVoltage25NGet();
158      void Testt004BoardSupplyVoltageGet();
159  };
```

Obrázek 11 - Ukázka testFixture a testSuite ve frameworku CppUnit
(Zdroj: vlastní)

2.1.2 JUnit

JUnit patří již dlouho mezi nejpoužívanější testovací frameworky v jazyce Java. Tento framework vytvořil Erich Gamma a Kent Beck jako instanci xUnit (Mishra, 2020).

JUnit byl vytvořen pro kódování opakujících se testů, ale může být také použit pro automatické testování (Mishra, 2020).

Výhody JUnit

Mezi značné výhody tohoto frameworku patří:

- Brzká detekce chyb v kódu

- Rozličné funkce pro kontrolu a porovnání hodnot
- Čitelný a přehledný kód
- Pokud metoda nejde otestovat, je pravděpodobně špatně navržená, zvyšuje tedy kvalitu kódu (Hordějčuk, nedatováno) (Mishra, 2020).
- JUnit se stal standardem pro testování v programovacím jazyce Java a je podporován téměř každým vývojovým prostředím.

Nevýhody JUnit

- Není vhodný pro obrovské testy (Mishra, 2020).

2.1.3 CppUnit

CppUnit je testovací framework pro psaní jednotkových testů v jazyce C++. Patří mezi frameworky xUnit, kterým se základní architekturou podobá. Tvůrci CppUnit se jmenují Michael Feathers a Jerome Lacoste. Framework vznikl přepsáním frameworku JUnit. Stejně jako všechny xUnit frameworky i tento je volně k použití. Implementace CppUnit využívá výhod jazyka C++ jako jsou např: šablony, abstraktní třídy a vnořené třídy (Hamill, 2004).

3 Praktická část

V praktické části jsou popsány testovací frameworky skupiny xUnit včetně jejich architektury. Nejvíce pozornosti je věnováno frameworkům JUnit a CppUnit. Jsou zde popsány jejich základní testovací metody. V poslední části praktické části jsou ukázány příklady testování, které jsem napsal během své praxe.

3.1 Konvence pro pojmenování Unit testu

Při psaní jednotkových testů jsou zavedené určité konvence pojmenování metod a tříd. Pokud chceme otestovat metodu *checkVoltage* vytvoříme testovací metodu s názvem *testCheckVoltage.* Testovací metoda test*CheckVoltage* bude volat metodu *checkVoltage* se všemi parametry (Andy Hunt, 2003).

3.2 Testovací metody

V testovacích frameworkách máme hned několik testovacích metod, které testování usnadní. Obecně se tyto metody jmenují *asserts.* Umožňují uživateli zkontrolovat platnost podmínky nebo porovnat, jestli se dvě čísla shodují nebo neshodují atd (Andy Hunt, 2003).

3.2.1 Testovací metody v JUnit

Asserts jsou základním stavením kamenem pro jednotkové testy. Knihovna JUnit poskytuje řadu různých forem těchto testovacích metod.

AssertEquals

```
assertEquals([String message], expected, actual)
```

Jednou z nejpoužívanějších testovacích metod je metoda assertEquals. Tato metoda porovná dvě hodnoty. *Message* je volitelná zpráva, která se objeví, zaznamená-li test chybu. *Expected*, tedy očekávaná hodnota je ta, která by při správném chodu programu měla nastat. *Actual*, tedy aktuální hodnota je hodnota, kterou test zachytí. Pokud se hodnoty neshodují, tak test zaznamená chybu (Andy Hunt, 2003).

AssertNull

```
assertNull([String message], java.lang.Object object)
```

```
assertNotNull([String message], java.lang.Object object)
```

Tato metoda porovná, jestli testovací objekt je nulový nebo nenulový. Pokud test neprojde, vypíše volitelnou zprávu (Andy Hunt, 2003).

AssertSame

```
assertSame([String message], expected, actual)
```

Kontroluje, zda *expected* a *actual* ukazují na stejný objekt. Pokud test neprojde, vypíše volitelnou zprávu (Andy Hunt, 2003).

```
assertNotSame([String message], expected, actual)
```

Kontroluje, zda *expected* a *actual* neukazují na stejný objekt. Pokud test neprojde, vypíše volitelnou zprávu (Andy Hunt, 2003).

AssertTrue

```
assertTrue([String message], boolean condition)
```

Kontroluje, zda podmínka typu *boolean* se rovná true. Pokud test neprojde, vypíše volitelnou zprávu (Andy Hunt, 2003).

```
assertFalse ([String message], boolean condition)
```

Kontroluje, zda podmínka typu *boolean* se rovná false. Pokud test neprojde, vypíše volitelnou zprávu (Andy Hunt, 2003).

Fail

```
fail ([String message])
```

Metoda fail okamžitě vrátí neúspěšný test s volitelnou zprávou. Tato metoda se obvykle používá k označení k sekcí kódu, ke kterým by se za správného chodu aplikace nemělo dosáhnout např. u výjimky (Andy Hunt, 2003).

Další testovací metody v JUnit

- `assertArrayEquals ()` - Zkontroluje, zda 2 pole obsahují ty samé prvky (Andy Hunt, 2003).
- `assertFalse` () - Zkontroluje, zda je hodnota false (Andy Hunt, 2003).
- `assertNotEquals` () - Zkontroluje, zda 2 hodnoty nejsou stejné (Andy Hunt, 2003).

3.2.2 Testovací metody v CppUnit

Framework CppUnit implementuje testovací metody jako makra. Tato makra umožňují při kompilaci preprocesoru najít a zaznamenat místa testovacích metod, což je jinak v jazyce C++ obtížné (Hamill, 2004).

Assert

`CPPUNIT_ASSERT (condition)`

Zkontroluje, zda podmínka platí (Sommerlade, nedatováno).

`CPPUNIT_ASSERT_MESSAGE (message, condition)`

Zkontroluje, zda podmínka platí. Pokud podmínka neplatí, vypíše zvolenou zprávu (Sommerlade, nedatováno).

Fail

`CPPUNIT_FAIL (message)`

Test selže a vypíše zvolenou zprávu (Sommerlade, nedatováno).

Assert Equal

`CPPUNIT_ASSERT_EQUAL (expected, actual)`

Zkontroluje, zda se hodnoty *expected* a *actual* shodují (Sommerlade, nedatováno).

`CPPUNIT_ASSERT_EQUAL_MESSAGE (message, expected, actual)` – Zkontroluje, zda se hodnoty shodují. Pokud se hodnoty neshodují, tak metoda vypíše volitelnou zprávu (Sommerlade, nedatováno).

Assert throw

`CPPUNIT_ASSERT_THROW (expression, ExceptionType)`

Zkontroluje, zda daný výraz vrací správný typ výjimky (Sommerlade, nedatováno).

`CPPUNIT_ASSERT_THROW_MESSAGE (message, expression, ExceptionType)`
Zkontroluje, zda daný výraz vrací správný typ výjimky. Pokud dojde k chybě, tak metoda vypíše volitelnou zprávu (Sommerlade, nedatováno).

`CPPUNIT_ASSERT_NO_THROW (expression)`

Zkontroluje, zda daný výraz nevyvolává žádnou výjimku (Sommerlade, nedatováno).

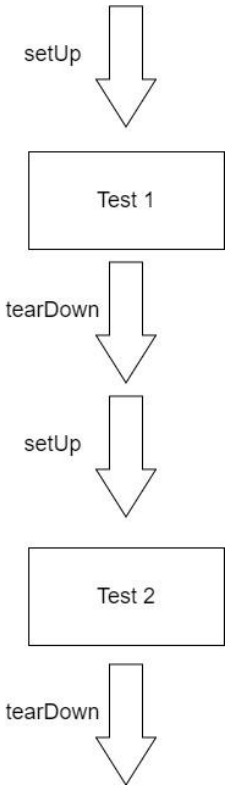
`CPPUNIT_ASSERT_NO_THROW_MESSAGE (message, expression)`

Zkontroluje, zda daný výraz nevyvolává žádnou výjimku. Pokud výraz vyvolá výjimku, tak metoda vypíše volitelnou zprávu (Sommerlade, nedatováno).

3.3 Konfigurace testovacího prostředí

Programátor potřebuje mít možnost spustit jakýkoliv test kdykoliv se mu to bude hodit. Je tedy důležité, aby jakákoliv data nebo instance z předchozích testů byly smazány a už se s nimi nadále nepracovalo. Při spuštění jakéhokoliv testu musí prostředí vypadat tak, jako při spuštění prvního testu (Osherove, c2014).

Pokud by při spuštění testu nebyla odstraněna všechna data, se kterýma se v přechozím testu pracovalo, mohlo by to zapříčinit špatnou funkci následujícího testu. Posuzovat, jestli testy na sobě jsou nezávislé je složité a časově náročné. Proto existují metody setUp a tearDown. Tyto metody pomáhají připravit prostředí pro správnou funkci testů (Osherove, c2014).



Obrázek 12 - Diagram metod setUp a tearDown při volání více testů
(Zdroj: Vlastní zpracování podle: (Osherove, c2014))

Metody setUp a tearDown

`void setUp () {}` -Tato metoda se spustí před spuštěním každého unit testu.

`void tearDown () {}` -Tato metoda se spustí po konci každého unit testu.

Využití metod

Programátor musí otestovat databázi, do které se při každém spuštění testu musí přihlásit. K ušetření času a práce využije metodu setUp, která se před spuštěním každého testu do databáze přihlásí. Jakmile test skončí, metoda tearDown se z databáze odhlásí (Andy Hunt, 2003).

```
public class TestDB extends TestCase {
	private Connection dbConn;
	protected void setUp() {
		dbConn = new Connection("oracle","martin","passwd1*","foobar");
		dbConn.connect();
	}

	protected void tearDown () {
		dbConn.disconnect();
		dbConn = null;
	}

	public void testAccountAccess() {
		// some testing code
	}

	public void testEmployeeAccess() {
		// some testing code
	}


}
```

Obrázek 13 - Využití metod setUp a tearDown

(Zdroj: Vlastní zpracování podle (Andy Hunt, 2003, p. 30))

3.4 Testovací dvojníci

Pokud část, kterou programátor chce otestovat závisí na velké části systému nebo dokonce celé části systému, je velice obtížné napsat unit test. Bylo by nutné inicializovat téměř celý systém, což je časově velice náročné. Díky využití testovacích dvojníků lze zachovat nezávislost testů a ušetřit spoustu času (Andy Hunt, 2003).

Typickým příkladem je testování platebního procesu. Dvojník platební metody vždy vrátí, že transakce proběhla v pořádku nebo naopak neproběhla, nestará se o proces platby (Andy Hunt, 2003).

Testovací dvojník je typ objektu, se kterým je mnohem snazší pracovat. Umožňuje rychleji a efektivněji psát unit testy.

Proč používat falešné objekty?

- Opravdový objekt má nepředvídatelné chování jako např burza.
- Nastavit opravdový objekt je příliš složité.
- Práce s reálným objektem je příliš pomalá.
- Reálný objekt má vlastní uživatelské rozhraní.
- Reálný objekt zatím neexistuje např. webová aplikace.

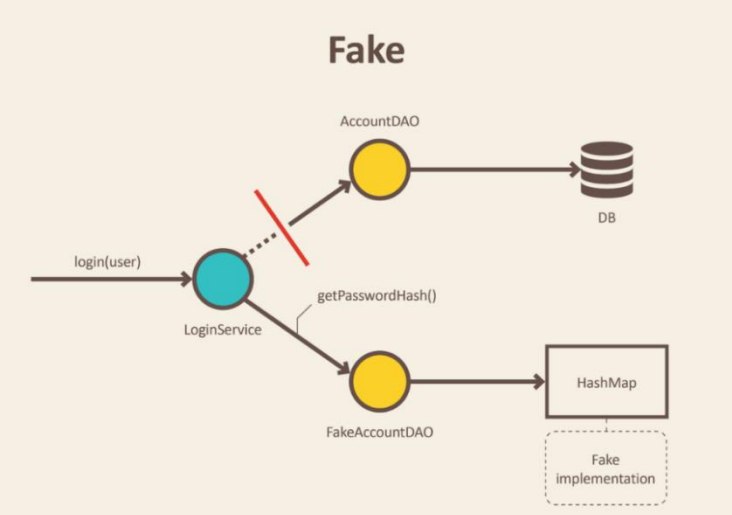
3.4.1 Dummy

Nejprimitivnější typ testovacího dvojníka je Dummy. Jde pouze o argument nebo atribut, který není doopravdy použit. Používá se jako placeholder k vyplnění parametrů, které jsou vyžadovány k spuštění testu (Meszaros, 2007).

3.4.2 Fake

Fakes jsou objekty, který nahradí kód implementací stejného rozhraní, ale již se neodkazují na jiné objekty. Fake metoda je většinou natvrdo naprogramovaný kód, který vrací pevně dané výsledky. (Lipski, 2017)

Příkladem může být fake metoda, která se nepřipojí do databáze, ale bude používat stejnou kolekci dat jako je v databázi (Lipski, 2017).

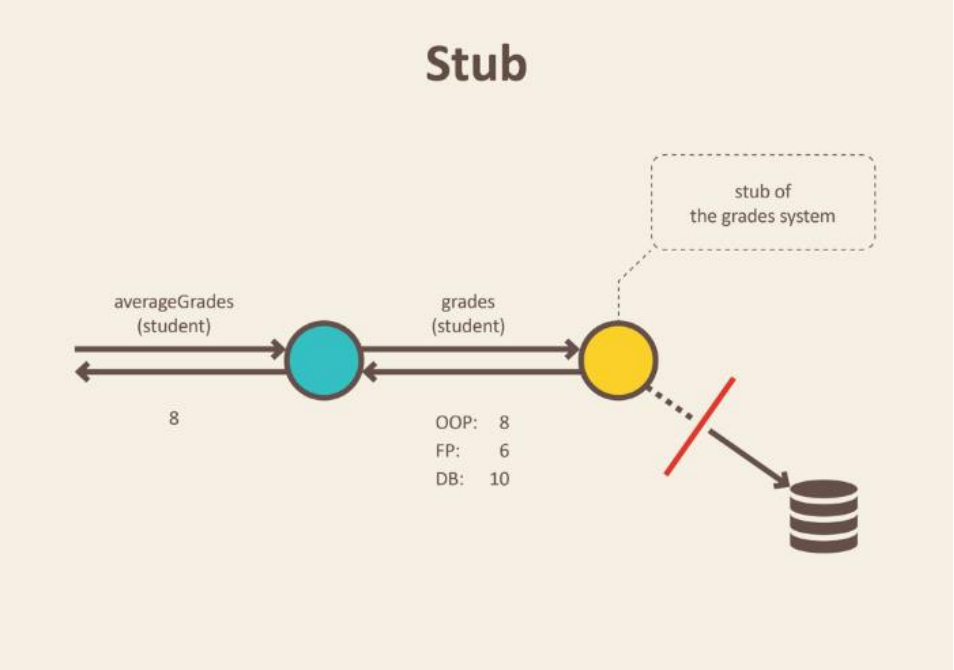


Obrázek 14 - Ukázka využití fakové implementace
(Zdroj: (Lipski, 2017))

3.4.3 Stub

Stub je objekt, který uchovává předdefinovaná data a vrací jej testovací metodě. Slouží jako náhrada za reálný objekt. Používá se, pokud je nežádoucí, aby test pracoval s opravdovými daty (Lipski, 2017).

Vhodné užití Stubu je například při testování tabulky, která obsahuje známky studentů. Místo toho, aby se při testování objekt připojoval do databáze a pracoval s jejími daty se testují předdefinované hodnoty (Lipski, 2017).



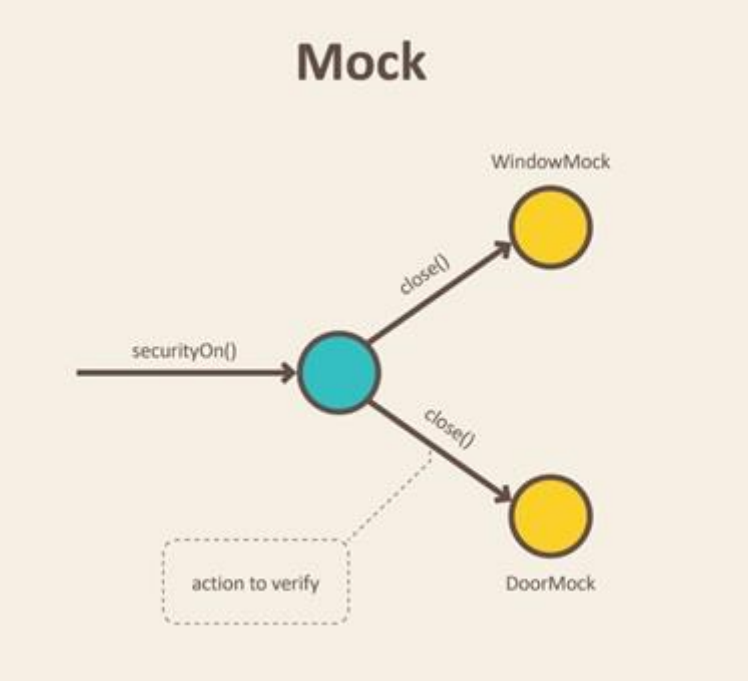
Obrázek 15 - Ukázka využití dvojníka Stub
(Zdroj: (Lipski, 2017))

3.4.4 Mock

Mock je komplikovanější verze Stubu. Mock také vrací hodnoty jako Stub. Na rozdíl od Stubu umožňuje definovat, v jakém pořadí a které volání proběhnou (Lipski, 2017).

Příklad využití mocku

Při testování funkčnosti e-mailového systému není efektivní v rámci každého testu posílat nový email. Použitím mocku je možné vytvořit falešnou zprávu, která ověří provedení metody, která odesílá e-maily. Obdobný příklad je ukázán na obrázku č.16



Obrázek 16 - Ukázka využití dvojníka Mock
(Zdroj: (Lipski, 2017))

4 Ukázky testování z praxe

Cílem této kapitoly je ukázat části testovacího kódu a demonstrovat využití jednotkových testů. Testy jsem vytvořil v rámci letní praxe ve firmě Datapartner s.r.o. ve frameworku CppUnit

4.1 Ukázka testování z praxe č. 1

Programátor dopíše metodu, která kontroluje správné nastavení napětí. Aby otestoval, zda metoda správně plní svojí funkci, musí promyslet všechny případy, které mohou nastat. Hodnotu napětí je nutné otestovat na povolené i nepovolené hranici. Maximální povolená hranice napětí je 150 + konstanta. Minimální povolená hranice napětí je 150 - konstanta. Definované makra DISCRIMINATION_VOLTAGE_DIFFERENCE má defaultní hodnotu 125 voltů. Maximální hodnota napětí tedy je 275 voltů a minimální hodnota napětí je 25 voltů.

Mohou nastat tyto případy:

1. Hodnota napětí je přesně na hranici maxima.
2. Hodnota napětí je za hranici povoleného maxima. Test neprojde.
3. Hodnota napětí je přesně před hranicí povoleného maxima.
4. Hodnota napětí je přesně na hranici minima.
5. Hodnota napětí je za hranicí povoleného minima. Test neprojde
6. Hodnota napětí je přesně před hranicí povoleného minima

1. Případ

```
159  void Testt004DiscriminationVoltage::TestAdcMeasDiscriminationVoltageCheck()
160  {
161      DISCRIMINATION_VOLTAGE actualDiscriminationVoltage;
162      DISCRIMINATION_VOLTAGE setDiscriminationVoltage;
163
164      actualDiscriminationVoltage.discriminationVoltage1 = 150;
165      setDiscriminationVoltage.discriminationVoltage1 = (150 + DISCRIMINATION_VOLTAGE_DIFFERENCE);
166      PALSTAT stat = MeasDiscriminationVoltageCheck(&actualDiscriminationVoltage, &setDiscriminationVoltage);
167      CPPUNIT_ASSERT_EQUAL((uint16_t)150, actualDiscriminationVoltage.discriminationVoltage1);
168      CPPUNIT_ASSERT_EQUAL((uint16_t)275, setDiscriminationVoltage.discriminationVoltage1);
169      CPPUNIT_ASSERT_EQUAL(PALS_OK, stat);
```

Obrázek 17 - Ukázka testovacího kódu

(Zdroj: vlastní)

Důležité jsou řádky 165,168 a 169. Na řádku 165 se nastavuje napětí na maximální povolenou hranici, která je nastavena na 275 voltů. Správné nastavení této hodnoty je testováno na řádku 168 testovací metodou CPPUNIT_ASSERT_EQUAL. Na posledním řádku se testuje makro PALS_OK, které říká, že test by neměl zaznamenat žádnou chybu.

2. Případ

```
171      actualDiscriminationVoltage.discriminationVoltage1 = 150;
172      setDiscriminationVoltage.discriminationVoltage1 = (150 + DISCRIMINATION_VOLTAGE_DIFFERENCE + 1);
173      stat = MeasDiscriminationVoltageCheck(&actualDiscriminationVoltage, &setDiscriminationVoltage);
174      CPPUNIT_ASSERT_EQUAL((uint16_t)150, actualDiscriminationVoltage.discriminationVoltage1);
175      CPPUNIT_ASSERT_EQUAL((uint16_t)276, setDiscriminationVoltage.discriminationVoltage1);
176      CPPUNIT_ASSERT_EQUAL(PALE_RANGE, stat);
```

Obrázek 18 - Ukázka testovacího kódu

(Zdroj: vlastní)

Na řádku 172 k maximální povolené hranici napětí je přičtena jednička. Na řádku 175 se kontroluje, zda se hodnota napětí správně uložila. Jelikož se testuje nepovolená hranice napětí, tak na řádku 176 se kontroluje chybové makro PALE_RANGE. Pokud test nezaznamená žádnou chybu, uživatel se o tom ihned dozví, protože toto makro dává programu najevo, že chyba je očekávána.

3. Případ

```
178     actualDiscriminationVoltage.discriminationVoltage1 = 150;
179     setDiscriminationVoltage.discriminationVoltage1 = (150 + DISCRIMINATION_VOLTAGE_DIFFERENCE - 1);
180     stat = MeasDiscriminationVoltageCheck(&actualDiscriminationVoltage, &setDiscriminationVoltage);
181     CPPUNIT_ASSERT_EQUAL((uint16_t)150, actualDiscriminationVoltage.discriminationVoltage1);
182     CPPUNIT_ASSERT_EQUAL((uint16_t)274, setDiscriminationVoltage.discriminationVoltage1);
183     CPPUNIT_ASSERT_EQUAL(PALS_OK, stat);
```

Obrázek 19 - Ukázka testovacího kódu

(Zdroj: vlastní)

Na řádku 179 se od maximální povolené hranice napětí odečítá jednička, napětí je tedy nastaveno na 274 voltů. Na řádku 182 se testuje nastavení této hodnoty. Test by neměl najít v programu žádnou chybu, což je testováno na posledním řádku makrem PALS_OK.

4. Případ

```
185     actualDiscriminationVoltage.discriminationVoltage1 = 150;
186     setDiscriminationVoltage.discriminationVoltage1 = (150 - DISCRIMINATION_VOLTAGE_DIFFERENCE);
187     stat = MeasDiscriminationVoltageCheck(&actualDiscriminationVoltage, &setDiscriminationVoltage);
188     CPPUNIT_ASSERT_EQUAL((uint16_t)150, actualDiscriminationVoltage.discriminationVoltage1);
189     CPPUNIT_ASSERT_EQUAL((uint16_t)25, setDiscriminationVoltage.discriminationVoltage1);
190     CPPUNIT_ASSERT_EQUAL(PALS_OK, stat);
```

Obrázek 20 - Ukázka testovacího kódu

(Zdroj: vlastní)

V tomto případu testujeme minimální možnou hranici napětí, což je hranice 25 voltů. Na řádku 186 se nastavuje testovaná proměnná na 25 voltů. Na řádku 188 se nastavení této hodnoty testuje. Test by měl proběhnout bez chyby.

5. Případ

```
192     actualDiscriminationVoltage.discriminationVoltage1 = 150;
193     setDiscriminationVoltage.discriminationVoltage1 = (150 - DISCRIMINATION_VOLTAGE_DIFFERENCE - 1);
194     stat = MeasDiscriminationVoltageCheck(&actualDiscriminationVoltage, &setDiscriminationVoltage);
195     CPPUNIT_ASSERT_EQUAL((uint16_t)150, actualDiscriminationVoltage.discriminationVoltage1);
196     CPPUNIT_ASSERT_EQUAL((uint16_t)24, setDiscriminationVoltage.discriminationVoltage1);
197     CPPUNIT_ASSERT_EQUAL(PALE_RANGE, stat);
```

Obrázek 21 - Ukázka testovacího kódu

(Zdroj: vlastní)

V páté případu se testuje přesáhnutí minimální možné hranice napětí. Testovaná hodnota je 24 voltů. Tato hodnota se nastavuje na řádku 193 a testuje na řádku 196. Pomocí makra PALE_RANGE testujeme, zda test neprošel.

6. případ

```
199     actualDiscriminationVoltage.discriminationVoltage1 = 150;
200     setDiscriminationVoltage.discriminationVoltage1 = (150 - DISCRIMINATION_VOLTAGE_DIFFERENCE + 1);
201     stat = MeasDiscriminationVoltageCheck(&actualDiscriminationVoltage, &setDiscriminationVoltage);
202     CPPUNIT_ASSERT_EQUAL((uint16_t)150, actualDiscriminationVoltage.discriminationVoltage1);
203     CPPUNIT_ASSERT_EQUAL((uint16_t)26, setDiscriminationVoltage.discriminationVoltage1);
204     CPPUNIT_ASSERT_EQUAL(PALS_OK, stat);
```

Obrázek 22 - Ukázka testovacího kódu

(Zdroj: vlastní)

V poslední testované možnosti je testovaná minimální možná hranice, ke které je přičtena jednička. Testovaná hodnota je 26 voltů. Tato hodnota se nastavuje na řádku 200 a testuje na řádku 203. Test by neměl nahlásit žádnou chybu.

4.2 Ukázka testování z praxe č. 2

Aplikace, na který programátor pracuje, si umí zaznamenávat vzniklé chyby a uchovávat si je v paměti. Testovací metoda, která kontroluje, zda se chyby správně ukládají do paměti může vypadat takto:

```
198 void TestNiEnhancedErrorReason::testEnhancedErrorReasonFirstTenErrorGet()
199 {
200     EnhancedErrorReasonInitialize();
201     ENHANCED_ERROR_REASON value[ENHANCED_ERROR_REASON_FAULT_SEQUENCE_PAGE_COUNT];
202     CPPUNIT_ASSERT_NO_THROW(EnhancedErrorReasonAdd(ENHANCED_ERROR_REASON_RELAY_NMIN2_FAILURE)); //51
203     CPPUNIT_ASSERT_NO_THROW(EnhancedErrorReasonAdd(ENHANCED_ERROR_REASON_CAMPBELL_COMPENSATION_CHANGE)); //20
204     CPPUNIT_ASSERT_NO_THROW(EnhancedErrorReasonAdd(ENHANCED_ERROR_REASON_VOLTAGE_33_DEVIATION)); //4
205     CPPUNIT_ASSERT_NO_THROW(EnhancedErrorReasonAdd(ENHANCED_ERROR_REASON_PROTOCOL_CRC)); //43
206     CPPUNIT_ASSERT_NO_THROW(EnhancedErrorReasonAdd(ENHANCED_ERROR_REASON_DAC_NOT_SET)); //35
207     CPPUNIT_ASSERT_NO_THROW(EnhancedErrorReasonAdd(ENHANCED_ERROR_REASON_VOLTAGE_COMPENSATION_DEVIATION)); //9
208     CPPUNIT_ASSERT_NO_THROW(EnhancedErrorReasonAdd(ENHANCED_ERROR_REASON_TEST_N711_FAILURE)); //53
209     CPPUNIT_ASSERT_NO_THROW(EnhancedErrorReasonAdd(ENHANCED_ERROR_REASON_NRST_RESET));  //70
210     CPPUNIT_ASSERT_NO_THROW(EnhancedErrorReasonAdd(ENHANCED_ERROR_REASON_OSCILATOR_RESET)); //69
211     CPPUNIT_ASSERT_NO_THROW(EnhancedErrorReasonAdd(ENHANCED_ERROR_REASON_WATCHDOG_RESET)); //68
212     EnhancedErrorReasonFaultSequenceGet(10,value);
213     CPPUNIT_ASSERT_EQUAL((ENHANCED_ERROR_REASON)51, value[0]);
214     CPPUNIT_ASSERT_EQUAL((ENHANCED_ERROR_REASON)20, value[1]);
215     CPPUNIT_ASSERT_EQUAL((ENHANCED_ERROR_REASON)4,  value[2]);
216     CPPUNIT_ASSERT_EQUAL((ENHANCED_ERROR_REASON)43, value[3]);
217     CPPUNIT_ASSERT_EQUAL((ENHANCED_ERROR_REASON)35, value[4]);
218     CPPUNIT_ASSERT_EQUAL((ENHANCED_ERROR_REASON)9,  value[5]);
219     CPPUNIT_ASSERT_EQUAL((ENHANCED_ERROR_REASON)53, value[6]);
220     CPPUNIT_ASSERT_EQUAL((ENHANCED_ERROR_REASON)70, value[7]);
221     CPPUNIT_ASSERT_EQUAL((ENHANCED_ERROR_REASON)69, value[8]);
222     CPPUNIT_ASSERT_EQUAL((ENHANCED_ERROR_REASON)68, value[9]);
223 }
```

Obrázek 23 - Ukázka testovacího kódu

(Zdroj: vlastní)

Na prvním řádku metody se inicializuje seznam chyb. Na řádcích 202–211 se naplní chybový seznam a pomocí testovací metody CPPUNIT_ASSERT_NO_THROW se zkontroluje, jestli výraz nevyvolává výjimku. Na řádcích 213–222 se už testovací

metodou CPPUNIT_ASSERT_EQUAL porovnávají očekávaná čísla chyb s čísly, kterými byl seznam chyb naplněn.

5 Diskuze

V této práci je vysvětleno, co to testování je a k čemu slouží. Dále jsou zde popsány typy testování dle fáze vývoje softwaru. V další kapitole jsou vyjmenovány jednotlivé výhody a nevýhody jednotkového testování. Výhody plně přesahují nevýhody, proto ho doporučuji při jakékoliv tvorbě náročnější aplikace. Nesmí se ale zapomenout, že pokud unit test není správně napsaný a nesplňuje náležitosti vyjmenované v této prací nelze se na něj spolehnout. Za nejdůležitější vlastnost unit testu považuji důvěryhodnost. Jakmile má programátor jistotu, že kód, který vytvořil plní požadovaný cíl, tak může přejít k dalšímu úkolu. Dále jsou v práci popsány testovací frameworky skupiny xUnit, které se nejvíce liší tím, že jsou psaný pro jiný programovací jazyk. Popsána je jejich základní architektura. U frameworků JUnit a CppUnit jsou představeny jejich nejzákladnější testovací metody. V poslední části práce je ukázáno využití jednotkových testů v praxi za použití frameworku CppUnit.

6 Závěr

Cílem této bakalářské práce bylo popsat problematiku jednotkových testů, představit nástroje pro implementaci jednotkových testů a ukázat využití unit testů na příkladech, které ukážou, jak testování kontroluje správný chod aplikace.

Práce vysvětluje, proč je dobré jednotkové testy vytvářet. Popisuje náležitosti kvalitního unit testu a demonstruje využití jednotkových testů na ukázkách kódu, který jsem vytvořil během své praxe.

Myslím si, že cíl se mi podařilo naplnit, ačkoliv jsem do práce nezařadil tolik praktických příkladů, kolik jsem z počátku plánoval.

Práci je nadále možné obohatit o jiné testovací frameworky a srovnání jednotlivých testovacích frameworků.

7 Citovaná literatura

Andy Hunt, D. T., 2003. *Pragmatic Unit Testing in Java with JUnit.* 2 editor United States of America: autor neznámý

Anon., 2019. *professionalqa.* [Online]
Available at: https://www.professionalqa.com/
[Přístup získán 12 1 2021].

Čermák, M., 2008. *cleverandsmart.* [Online]
Available at: https://www.cleverandsmart.cz/grey-box-test/
[Přístup získán 1 12 2021].

Eriksson, U., 2014. *ReQtest.* [Online]
Available at: https://reqtest.com/testing-blog/different-levels-of-testing/
[Přístup získán 11 1 2021].

Feathers, M. C., 2009. *Working Effectively with Legacy Code.* Londýn: Computer Press.

Hamill, P., 2004. *Unit Test Frameworks.* místo neznámé:O'Reilly Media, Inc..

Hlava, T., 2011. *testovanisoftwaru.* [Online]
Available at: http://testovanisoftwaru.cz/tag/urovne-testovani/
[Přístup získán 11 1 2021].

Hordějčuk, V., nedatováno *voho.* [Online]
Available at: http://voho.eu/wiki/java-junit/
[Přístup získán 11 1 2021].

Kitner, R., 2015. *kitner.* [Online]
Available at: https://kitner.cz/testovani_softwaru/typy-testovani-software-trideni-testu/
[Přístup získán 12 1 2021].

Kulkarni, K., nedatováno *guru99.* [Online]
Available at: https://www.guru99.com/test-driven-development.html
[Přístup získán 19 4 2021].

Lipski, M., 2017. *Pragmatists.* [Online]
Available at: https://blog.pragmatists.com/test-doubles-fakes-mocks-and-stubs-1a7491dfa3da
[Přístup získán 18 4 2021].

Meszaros, G., 2007. *xUnit Test Patterns: Refactoring Test Code.* Boston: autor neznámý

Mishra, P., 2020. *lambdatest.* [Online]
Available at: https://www.lambdatest.com/blog/11-best-unit-testing-frameworks-for-selenium-automation/
[Přístup získán 11 1 2021].

Novoseltseva, E., 2017. *dzone.* [Online]
Available at: https://dzone.com/articles/top-8-benefits-of-unit-testing
[Přístup získán 11 1 2021].

Osherove, R., c2014. *The art of unit testing with examples in C#. 2nd ed.*. New York: Manning.

Rajkumar, 2021. *softwaretestingmaterial.* [Online]
Available at: https://www.softwaretestingmaterial.com/software-testing/
[Přístup získán 12 1 2021].

Sommerlade, E., nedatováno *cppunit.* [Online]
Available at: http://cppunit.sourceforge.net/doc/cvs/index.html
[Přístup získán 17 4 2021].