FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Refactoring Unit testů na backendu
Student:	Jakub Čapek
Vedoucí:	Ing. Petr Sobotka
Studijní program:	Informatika
Obor / specializace:	Webové a softwarové inženýrství, zaměření Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2023/2024

Pokyny pro vypracování

Tato práce je vypracována ve spolupráci se společností KOMIX s.r.o. Cílem práce je vytvořit seznam doporučení pro zefektivnění, zpřehlednění a optimalizaci Unit testů backendu reálné Java aplikace, kterou společnost KOMIX vyvíjí pro zákazníka.
Postupujte v následujících krocích:
1. Na základě analýzy stávajícího provedení vytvořte seznam míst vhodných pro zlepšení a úpravu.
2. Navrhněte vlastní řešení částí, které byly v minulém kroku vytipovány k úpravě.
3. Vytvořte příklady implementace návrhu tak, aby funkčnost původních příkladů testů zůstala zachována.
4. Diskutujte původní a novou verzi testů z hlediska efektivity a přehlednosti.

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 6. února 2023 v Praze.

Bakalářská práce

REFACTORING UNIT TESTŮ NA BACKENDU

Jakub Čapek

Fakulta informačních technologií
Katedra softwarového inženýrství
Vedoucí: Ing. Petr Sobotka
10. května 2023

Obsah

Seznam tabulek

Seznam výpisů kódu

Rád bych vyjádřil své poděkování všem, kteří mi jakkoliv pomohli při psaní této bakalářské práce. Za umožnění vytvoření této práce bych chtěl poděkovat společnosti KOMIX s.r.o. Za přínosné rady a vedené této práce bych chtěl poděkovat mému vedoucímu. Také bych chtěl poděkovat rodině, která mě podporovala během celého studia. V neposlední řadě bych rád vyjádřil svůj vděk všem mým přátelům a spolužákům, kteří mě podporovali a pomáhali v průběhu studia. Nakonec chci poděkovat všem, kteří se mnou diskutovali o tématu této práce a pomohli mi k této problematice získat pohled z více úhlů.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací. Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 citovaného zákona.

V Praze dne 10. května 2023 .....

Abstrakt

Tato práce se zabývá řešením nedostatků, které se vyskytují v Unit testech Java aplikace. Nedostatky, které jsou probírané do hloubky, jsou globálně využívané třídní proměnné testových tříd, neefektivní inicializace testů, nepřehledné nastavování testových objektů, nesprávné členění testů, nedodržování jednotné jmenné konvence a duplikování kódu v testech. Pro všechny tyto nedostatky je v práci navržen doporučený postup, který je vždy předveden v ukázce.

Práce se dále zabývá dalšími běžnými chybami při testování, jako jsou například používání data, potřeba testování privátních metod a mockování statických tříd.

Veškeré navržené postupy jsou použity na reálném projektu s pozitivními výsledky. Refactoring dle těchto postupů vedl ke zlepšení přehlednosti a efektivity testů.

Klíčová slova Unit testy, refactoring testů, testování softwaru, Java, JUnit, Mockito

Abstract

This thesis focuses on creating solutions to found problems in Unit tests of existing Java application. Problems which are discussed are global usage of test class variables, inefficient test initialization, not transparent setting of test objects, incorrect division of tests, non-compliance with the uniform naming convention and code duplication in tests. There is created recommended solution for each problem. Every solution is shown in example.

The thesis also deals with other common mistakes in testing, such as using dates, need to test private methods, and mocking static classes.

All proposed solutions are used on a real project with positive results. Refactoring according to these solutions led to an improvement in the clarity and effectiveness of the tests.

Keywords Unit tests, test refactoring, software testing, Java, JUnit, Mockito

Seznam zkratek

FURPS	metoda pro ověření kvality softwaru (Functionalitu, Usability, Reliability, Performance, Supportability)
GUI	uživatelské rozhraní
IDE	integrované vývojové prostředí

Kapitola 1
Úvod

Testování softwaru je běžná a důležitá součást vývoje aplikace, ve které se ověřuje správnost konkrétních částí aplikace. Testování se provádí, aby se v co největší míře zamezilo chybám v kódu, které stojí vývojáře čas a zákazníky peníze. Zároveň psaní testů aplikace nebývá pro vývojáře nejoblíbenější částí vytváření softwaru. Z tohoto důvodu se stává, že testy obsahují nedostatky a chyby. Těmto nedostatkům a jejich řešením se bude tato práce zabývat.

Práce je vytvořena ve spolupráci se společností KOMIX s.r.o. na existující Javovské aplikaci, ve které se tyto nedostatky také nacházejí. Výsledná práce může být využita vývojáři zabývajícími se refactoringem testů této aplikace.

Téma bakalářské práce bylo vybráno, jelikož je na výše zmíněném projektu refactoring Unit testů potřeba, takže tato práce může být dobře využita v praxi. Zároveň mě problematika testování zajímá a zabývám se jí delší dobu.

Ačkoliv existuje více druhů testů například integrační, systémové, akceptační… Tak práce se bude zabývat pouze refactoringem Unit testů.

Cílem práce je tedy vytvoření seznamu doporučení a postupů pro vyřešení nedostatků a problémů v Unit testech backendu existující Java aplikace. Tato doporučení a postupy jsou zaměřena na zlepšení efektivity, přehlednosti a optimalizaci Unit testů. Mezi cíle práce dále patří, aby seznam byl dostatečně srozumitelný a obecný aby mohl být využit i na jiných projektech.

Práce bude rozdělena do vícero kapitol. Následující kapitola se bude věnovat testování. Nejdříve se popíšou důvody a přínosy testování. Poté se probere jakým způsobem se testy dělí na konkrétní typy. A nakonec se objasní pojem „Unit test“.

Další kapitola bude zaměřena na popis aplikace. Uvede k čemu aplikace slouží, jaké se v ní využívají technologie. A na závěr přiblíží aktuální implementaci Unit testů.

Ve čtvrté kapitole se provede analýza aktuálního řešení Unit testů s cílem získání seznamu nedostatků a chyb.

Další kapitola se bude věnovat jiným chybám, které se často objevují při testování aplikací.

Šestou kapitolou je praktická část, ve které se postupně využije seznam nedostatků a chyb získaný v analýze. Budou zde postupně rozebrány všechny nalezené nedostatky. Rozebrání nedostatku se bude dělit na čtyři fáze. V první fázi bude popis nedostatku, druhá fáze obsahuje vysvětlení jaké chyby nebo potenciální chyby mohou kvůli konkrétnímu nedostatku vznikat. Ve třetí fázi bude návrh, jak takový problém správně řešit a poslední fáze bude obsahovat implementovanou ukázku navrženého řešení.

Poslední kapitolou práce bude závěr, ve kterém bude zhodnoceno splnění cílů. Efektivita bude zhodnocena pomocí měření času a zlepšení přehlednosti kódu bude popsáno slovně.

Kapitola 2
Testování

V této čistě teoretické kapitole bude vysvětleno, proč se testování aplikace provádí a jaké jsou jeho přínosy. Dále bude popsáno, jakým způsobem se dají testy rozdělit do skupin podle různých kritérií. Na závěr proběhne konkrétnější objasnění termínu Unit test.

2.1 Důvody a přínosy testů

Testování softwaru je výzkum kvality testovaného produktu. Je to také nedílná součást vývoje softwaru. Při testování se objektivně zhodnotí kvalita systému, čímž může být získána důvěra zákazníka. Dále testy objevují chyby vzniklé při implementaci systému. To ovšem neznamená, že pokud všechny testy proběhnou úspěšně, tak software je bez chyb. Testy obecně mohou pomoci nalézt chyby a potvrdit chybovost aplikace, ale nedokážou prokázat, že aplikace je bezchybná. Testy pomáhají ověřit zda nově implementovaná funkce nějakým způsobem nezpůsobila chyby ve „starším“ kódu. Hlavními účely testování je tedy omezení výskytu chyb ve vydaném softwaru a zhodnocení kvality softwaru dle více kritérií. [1] [2]

2.2 Druhy testů

Rozdělit softwarové testy se dá podle mnohých kritérií. Toto kritérium určuje, jakým způsobem je na test nahlíženo a do jaké skupiny bude zařazen. Kritéria podle kterých se testy dělí tedy jsou:

- rozsah
- způsob provádění
- znalost vnitřní struktury
- způsob vyhodnocování
- zaměření

[3]

2.2.1 Dle rozsahu

Toto dělení rozděluje testy podle kódu který je testován. Pokud je test zaměřený pouze na jednu konkrétní funkci, metodu nebo třídu nazveme ho Unit testem. Když se ověřuje funkčnost celého

modulu nebo komponenty, tak se test označuje jako Komponent test. Komponent test se obvykle velice podobá Unit testu, ale hlavním rozdílem je rozsah testovaného kódu. [4]

Obě tyto kategorie testů zpravidla píší samy vývojáři, ale existují i jiné kategorie testů, které mají na starost většinou testeři, jedna z nich jsou Akceptační testy. Tyto testy ověřují správnost a funkčnost komunikace mezi moduly a komponentami uvnitř aplikace. Další kategorií jsou testy Systémové. „*Během těchto testů je aplikace ověřována jako funkční celek. Tyto testy jsou používány v pozdějších fázích vývoje. Ověřují aplikaci z pohledu zákazníka. Podle připravených scénářů se simulují různé kroky, které v praxi mohou nastat*“. [5]

- Unit
- Komponent
- Akceptační
- Systémové

2.2.2 Dle způsobu provádění

Podle tohoto způsobu dělení existují dvě kategorie testů. První kategorií je Statické testování, to může probíhat i na nespustitelném kódu. Odhaluje například chyby syntaxe nebo upozorňuje na škodlivý kód, který může v průběhu spuštění způsobit problémy. Další kategorií je Dynamické testování, které probíhá při běhu aplikace a zkoumá, že kód se zadanými vstupy vrátí očekávané výsledky. [6]

- Dynamické
- Statické

2.2.3 Dle znalosti vnitřní struktury

Toto dělení se zaměřuje na rozdělení testů podle znalostí použité implementace v testovaném kódu a dělí testy na tři skupiny. První skupina se nazývá White Box. White Box testy využívají kompletní znalosti použitých datových struktur i algoritmů a dokáží velmi dobře identifikovat možné problémy. Typickým příkladem jsou Unit testy. Další skupinou jsou Black Box testy, které nemají žádnou znalost použitých technologií a ověřují pouze rozhraní a chování kódu. Poslední skupinou jsou Gray Box testy, což jsou testy prováděné s částečnou znalostní vnitřní implementace. [3]

- WhiteBox
- GrayBox
- BlackBox

[3]

2.2.4 Dle způsobu vyhodnocování

Pokud rozdělíme testy podle způsobu vyhodnocování, vzniknou dvě skupiny. První skupinou jsou testy automatické. Do druhé skupiny patří testy manuální. Nejčastějším příkladem automatického testu je Unit test. Manuální testování stojí hodně času i práce, jelikož musí být vždy někdo, kdo test vykoná a na konci rozhodne o výsledku provedeného testu. Tuhle neefektivitu odstraňují automatizované testy, kde téměř všechnu práci zabere napsání takového testu a pozdější spouštění a vyhodnocování probíhá bez další práce vývojáře. Další výhodou automatizovaných testů je, možnost opakovaného spouštění. [7]

- Automatické
- Manuální

2.2.5 Dle zaměření

Většinu základních vlastností softwaru lze zařadit pod některý ze základních rozměrů kvality FURPS. Testy lze tedy rozdělit podle toho, jaký ze základních rozměrů kvality FURPS ověřují. První skupina „F“ jako functionality obsahuje testy zaměřené na testování funkčních požadavků, jako například zda jsou poskytovány funkčnosti uvedené v návrhu systému.

„U“ označuje anglické slovo usability, přeložené jako použitelnost. Tyto testy ověřují GUI (uživatelské rozhraní), testy on-line nápověd a testy školících materiálů.

Jako třetí kategorií je písmeno „R“, vyjadřující slovo reliability (spolehlivost). Mezi tyto testy spolehlivosti patří zejména kontroly zachování poskytovaných funkcí a vlastností systému.

Další skupinou jsou testy výkonnostní („P“ performance). Výkonnostní testy jsou specifické svými nároky na testovaný systém. Obvykle jsou prováděny specialisty. Ověřují odezvy systému a funkčnosti systému při zvýšené zátěži.

Do poslední kategorie spadají testy podporovatelnosti („S“ supportability). Tyto testy ověřují například jestli bude daný systém funkční i na jiných hardwarových nebo softwarových konfiguracích, než jaké jsou nastaveny v současném testovacím nebo produkčním prostředí. [8]

- Funkčnost
- Použitelnost
- Spolehlivost
- Výkonost
- Podporovatelnost

2.3 Unit testy

Jelikož se tato práce zabývá refactoringem právě Unit testů. Je potřeba si lépe popsat, co Unit test je. Unit test má za úkol ověřit funkčnost právě jedné metody, funkce, třídy nebo funkcionality podle toho, jak se v daném kontextu definuje slovo Unit. Je vhodné, aby se test jmenoval stejně jako funkce nebo funkcionalita kterou testuje a měl by ověřovat funkčnost pouze té části kódu, pro kterou je psán a žádné jiné. Mělo by být na první pohled zřejmé, co daný test kontroluje a neměl by být závislý na žádných jiných testech. [9]

Unit test obvykle mívá tři fáze. První fáze spočívá ve vytvoření a nastavení všech potřebných proměnných a objektů, které jsou k testu potřebné. V druhé fázi je s těmito předem nastavenými (nasetovanými) objekty testovaná operace nebo metoda spuštěna. Třetí fáze, neboli vyhodnocení určuje výsledek daného testu. Odehrává se v ní porovnání vykonané operace s předem určeným

očekávaným výsledkem. Pokud nastane odchylka mezi skutečným a očekávaným výsledkem, znamená to, že test proběhl neúspěšně. To indikuje výskyt chyby v testovaném kódu nebo špatně napsaný test. V opačném případě test proběhl úspěšně.

Kapitola 3
O aplikaci

Následující kapitola přiblíží aplikaci na které je tato práce vypracována. Vysvětlí technologie, které jsou v aplikaci využívány. A na závěr popíše, jakým způsobem funguje momentální implementace backendových Unit testů.

3.1 Popis aplikace

Tato práce je vypracována na již existující aplikaci. Aplikace je psána v Javě. Komunikuje s dalšími aplikacemi pomocí webových služeb. Aplikace se zaměřuje na procesy s občanskými průkazy jako jsou například prodloužení průkazu, vytvoření nového průkazu nebo zrušení průkazu.

3.2 Použité technologie

V této části práce budou popsány nejdůležitější technologie využívány při testování aplikace.

3.2.1 JUnit5

JUnit5 je framework určený pro jednotkové (Unit) testování javovského kódu. Tento framework nabízí anotace pro konfiguraci jednotlivých testů. Framework je podporován nejčastěji využívanými vývojovými prostředími, například IntelliJ IDEA, Eclipse, NetBeans a Visual Studio Code. Dále je framework podporován těmito nástroji pro sestavení aplikace (build tools): Maven, Gradle, Ant. JUnit5 vyžaduje pro správnou funkcionalitu, aby systém běžel na Javě 8 a vyšší. [10]

3.2.1.1 Anotace *@Test*

Anotace *@Test* označuje, že daná metoda se stává testovací metodou. Tato metoda může být spuštěna samostatně, nebo i hromadně při spuštění více až všech testů. [11]

3.2.1.2 Anotace *@BeforeEach*

Touto anotací *@BeforeEach* se označují metody, které slouží pro přípravu objektů před vykonáním testů. Metody označené touto anotací se provedou před každou testovou metodou v této konkrétní testové třídě. Tento fakt platí i pro dědění. Metody anotovány tímto způsobem jsou

děděny od rodiče, pokud nejsou přepsány pomocí klíčového slova *@Override*. Tudíž metody s anotací *@BeforeEach* z rodičské třídy se provedou před všemi testovacími metodami podtřídy. Tato metoda nesmí být statická a její návratový typ musí být *void*. [12] [13]

Využití této anotace spočívá ve vytvoření inicializační metody, ve které se provede nastavení nebo namockování potřebných proměnných. Je vhodná pro použití v situacích, kde je třeba nastavovat a mockovat tyto proměnné před každou testovou metodou.

3.2.1.3 Anotace *@BeforeAll*

Anotace *@BeforeAll* slouží k označení metody, která provede přípravu prostředí k testům. Metoda s touto anotací proběhne právě jednou, a to před spuštěním testů v její třídě. Anotace pracuje při dědění stejně jako předchozí anotace *@BeroreEach* 3.2.1.2. Tato metoda by měla být statická a její návratový typ musí být *void*. Pokud tato metoda není statická, je vhodné, aby celá testová třída byla označena anotací *TestInstance* 3.2.1.6. [14] [15]

Hlavní využití této anotace je pro vytvoření, nastavení a namockování proměnných potřebných k testům. Pokud je tato operace časově náročná nebo ji není potřeba vykonávat pravidelně před každým testem, tak je anotace *@BeforeAll* ideální.

3.2.1.4 Anotace *@AfterEach*

Anotací *@AfterEach* se označují metody, které jsou určeny pro vyčištění prostředí po testu. Metoda označena touto anotací bude vykonána po každém testu dané třídy. Tato metoda nesmí být statická a musí vracet datový typ *void*. Metody anotované pomocí této anotace jsou děděny, pokud nejsou v třídě potomka přepsány klíčovým slovem *@Override*. [16]

Využití této anotace je při potřebě vykonat jakoukoliv operaci po každém testu.

3.2.1.5 Anotace *@AfterAll*

Anotace *@AfterAll* slouží pro označení metod, které mají být spuštěny po ukončení všech testů dané třídy. Narozdíl od anotace *@AfterEach* se tato metoda provede právě jednou. Tato metoda musí mít návratový datový typ *void* a měla by být statická. Pokud metoda není statická, tak by třída měla být označena anotací *TestInstance* 3.2.1.6. Dědění metod s touto anotací má stejnou logiku jako u anotace *AfterEach* 3.2.1.4. [17]

Anotace je využívána pro vyčištění a úklid testovacího prostředí po provedení všech testů dané třídy.

3.2.1.6 Anotace *@TestInstance*

Tato anotace slouží pro označení celé testové třídy. Umožňuje vytvořit novou instanci dané třídy před spuštěním jejích testů.

Existují dvě varianty použití této anotace. První možností je využití „PER_CLASS“, které vytvoří instanci před spuštěním všech testů dané třídy. Tato varianta je předvedena v ukázce 3.1. [18]

Výpis kódu 3.1 Ukázka použití anotace TestInstance varianty pro třídu

```
@TestInstance(TestInstance.Lifecycle.PER_CLASS)
```

Druhou možností je využití „PER_METHOD“, které vytvoří novou instanci před každým jedním testem dané třídy. Nastavení této varianty je předvedeno v ukázce 3.2. Tato varianta je základní, tudíž při využití této anotace bez dalšího upřesnění bude nastavena tato varianta. [18]

Výpis kódu 3.2 Ukázka použití anotace TestInstance varianty pro metodu

```
@TestInstance(TestInstance.Lifecycle.PER_METHOD)
```

3.2.2 Maven

Maven je nástroj pro správu softwarových projektů, který se primárně využívá u projektů založených v Javě. Maven pomáhá spravovat sestavení, dokumentaci, závislosti, konfiguraci, vydání a distribuci softwaru. Mnoho integrovaných vývojových prostředí (IDE) poskytuje plug-iny nebo doplňky pro Maven, což Mavenu umožňuje kompilovat projekty přímo z IDE. Základní jednotkou v Mavenu je projektový model projektu (POM), soubor XML, který obsahuje informace o softwarovém projektu, podrobnosti o konfiguraci, které Maven používá při vytváření tohoto projektu. Dále XML soubor obsahuje informace o veškerých závislostech na externích komponentách nebo modulech a pořadí sestavení. [19]

Maven rozděluje proces sestavení projektu do jednotlivých fází. To umožňuje Mavenu projekt spouštět předem definovaným způsob a zajišťuje stejnou funkcionalitu na různých zařízeních. Tato skutečnost je důležitá pro vývoj aplikací v týmech. Tyto fáze sestavení je možné manuálně spouštět, což ulehčuje vývojářům práci při vývoji aplikace. Proces sestavení se dělí do těchto fází:

- validate
- compile
- test
- package
- verify
- install
- deploy

Pro tuto práci je nejdůležitější fáze „test“. Ta spouští proces testování, neboli spustí zkompilovaný kód pomocí nastaveného frameworku určeného na Unit testování. [20]

3.2.3 Mockito

Mockito je framework určený pro psaní Unit testů v Javě. Slouží k mockování tříd. Mockování je proces, při kterém není volána konkrétní instance dané třídy, ale pouze její Mock. Mock slouží jako náhražka reálného objektu. Této náhražce lze definovat chování při různých situacích, aby se s ní mohli v testech volat metody mockovaného objektu. Tento mock dále může získávat informace o jeho volání, například počet volání nějaké jeho metody. [21]

Hlavním účelem frameworku Mockito je usnadnit vývoj testů mockováním závislostí a voláním těchto mocků v testech. [22]

Mockito poskytuje řadu předdefinovaných metod, které mohou být při testování volně využity. [23]

3.2.3.1 Metoda *mock()*

Metoda *mock()* dovoluje vytvoření mockovaného objektu třídy nebo rozhraní (interface). Takto vytvořený mock je ihned připraven k dalšímu použití. [24]

Výpis kódu 3.3 Ukázka volání metody mock()

```
Person mockedPerson = Mockito.mock(Person.class);
```

3.2.3.2 Metoda *when()*

Metoda *when()* umožňuje definovat chování metod mockovaného objektu. Je používána v situaci, kdy je potřeba, aby metoda vracela konkrétní hodnoty.

Nejčastěji se používá v kombinaci s metodou *thenReturn()*. Ve výpisu kódu 3.4 je ukázáno volání této kombinace, chování tohoto kódu je: „Když je zavolána metoda ABC třídy class, tak vrať XYZ.“ [23]

Výpis kódu 3.4 Ukázka volání metody when() s kombinací s metodou thenReturn()

```
Mockito.when(class.ABC()).thenReturn(XYZ);
```

3.2.3.3 Metoda *verify()*

Metoda *verify()* se používá pro ověření, zda nějaké metody byly volány, nebo ne. Slouží k validaci chování kódu. Metoda se používá na konci testovací metody pro ověření zda byly definované metody zavolány.

Framework Mockito si udržuje informace o všech metodách a parametrech zavolaných na mockovaných objektech. Tento způsob testu neověřuje výsledek metody, ale namísto toho ověřuje, zda byly namockované metody volány se správnými parametry.

Touto metodou je možné ověřit i počet zavolání dané metody. Pomocí doplňujících metod: *times()*, *atLeastOnce()*, *atMost()*.

V ukázce 3.5 je předvedeno použití této metody pro ověření, zda byla metoda *age()* zavolána právě jednou. [23]

Výpis kódu 3.5 Ukázka volání metody verify()

```
Person mockedPerson = mock(Person.class);
mockedPerson.age();
Mockito.verify(mockedPerson, times(1)).age();
```

3.3 Aktuální implementace Unit testů

Unit testy aplikace jsou implementovány pomocí frameworku JUnit5. Využívají výše popsané anotace, pro konkrétní testovací metody je využita anotace *@Test* 3.2.1.1. Dále se ve většině třídách používají anotace *@BeforeEach* 3.2.1.2 a *@BeforeAll* 3.2.1.3 pro inicializaci objektů před spuštěním testů. Hlavním zaměřením Unit testů je ověřit správnost uživatelských procesů.

Testy dodržují jednotnou strukturu souborů i jednotné pojmenovávání. Testový soubor obsahuje jednu testovou třídu. Testovací třída se nazývá stejně jako testovaná metoda se zakončením „Test“ a konkrétní testy uvnitř této třídy se jmenují jako daná operace nebo situace, která je testována. Jelikož testová třída obsahuje testy pouze k jedné metodě, tak pro třídy obsahující více metod existuje více testovacích tříd, každý pro jinou metodu. Tyto testovací třídy jsou dále rozděleny do balíčků podle třídy, která je právě testována. Tyto třídy zpravidla mají stejné třídní proměnné, z tohoto důvodu je v implementaci často využíváno dědění od společného předka. Tento předek se nazývá jako testovaná třída se zakončením „Init“. Tato struktura umožňuje přehled nad testovaným kódem a nabízí rychlou identifikaci případné chyby.

Celá aplikace je řízena výjimkami (Exceptions), v testech tato skutečnost znamená, že pokud všechny operace proběhnou úspěšně kód nevyhodí žádnou výjimku. Při jakékoliv neúspěšné akci naopak kód vyhazuje výjimku, ta se v testu odchytává a kontroluje se, zda se jedná o ten správný a očekávaný typ výjimky. Tyto testy, které jsou navrženy, aby výjimky způsobovaly a ihned je odchytávaly, mají zpravidla v názvu koncovku „_vratiChybu“.

Kapitola 4
Analýza nedostatků

V této kapitole bude provedena analýza Unit testů aplikace. Cílem analýzy je získaní seznamu nedostatků, které se v implementaci nachází.

4.1 Požadavky

Aby bylo možné provést analýzu implementace Unit testů, je potřeba si nejdříve definovat, jaký je očekávaný stav Unit testů a jaké vlastnosti mají testy splňovat.

První takovou vlastností je vzájemná nezávislost jednotlivých testů. Je potřebné, aby na sobě testy nebyly nijak závislé, a aby se mezi sebou vzájemně neovlivňovaly. Tedy pokud budou testy spuštěny hromadně, je chtěné, aby měly stejný výsledek, jako když se každý test spustí samostatně. Výhodou této vlastnosti je možnost spouštět testy paralelně a tím urychlit čas běhu testů.

Další vlastnost, která je očekávaná, je, aby dané testy ověřovaly jen ten kód, pro který je test psán. Tedy aby se v testovacích metodách netestovaly jiné části kódu, které zasahují do kódu jiných tříd nebo metod.

Mezi požadované vlastnosti patří i čitelnost testů. Je třeba, aby z testované metody bylo na první pohled zřejmé, co daný test ověřuje a k čemu slouží. Tato vlastnost podporuje lepší orientaci v testech při situaci, kdy nastane chyba v produkčním kódu.

Další chtěnou vlastností je dodržování jednotné struktury. Tím je myšleno, aby testy dodržovaly pravidelnou strukturu souborů a aby byla dodržována jednotná jmenná konvence. Tím je myšleno, aby se podobné operace se stejnou funkcionalitou napříč testy jmenovaly stejně. Například operace *create* a *mock*.

Poslední očekávanou vlastností je efektivita testů. Aplikace mohou z pohledu počtu řádek kódu dorůst obrovských rozměrů a tím pádem by měl správně i vzrůst počet napsaných testů. Není tedy žádoucí, aby testy byly zbytečně neefektivní, proto je důležité aby se hledělo na efektivitu i v testovém kódu.

4.2 Nalezené nedostatky

Po stanovení požadovaných vlastností, které mají testy splňovat je na řadě projít momentální implementaci testů a prověřit zda jsou tyto vlastnosti splněny. Při procházení jednotlivých testovacích balíčků bylo zjištěno, že testy v balíčku „Přestupek“ často porušují uvedené vlastnosti.

Testy v tomto balíčku používají globální třídní proměnné, čímž porušují nezávislost jednotlivých testů v souboru. Další nalezenou chybou je neefektivní inicializování objektů před spuštěním testovacích metod. Dalším nalezeným problémem je, že v jednotlivých testech není zřejmé jaké

hodnoty se do objektů nastavují, tím se prudce zvyšuje nečitelnost a nepřehlednost kódu. V tomto balíčku bylo dále zjištěno, že některé testy ověřují více kódu než patří do testované třídy, proto je potřebné provést úpravu a rozdělit testovací metody do více balíčků podle testovaných funkcionalit. Pátým nedostatkem je nedodržování jmenných konvencí pro metody stejných účelů, konkrétně metody sloužící pro inicializaci testovacích objektů se v každém souboru jmenují rozdílně. Dalším objeveným nedostatkem je duplikování konstant a pomocných metod mezi soubory, kód se tímto stává zbytečně delší a méně přehledný.

Kapitola „Řešení problémů“ 6 se bude těmito nedostatky a problémy zabývat dopodrobna. Se snahou najít pro každý z nich adekvátní způsob řešení.

4.3 Seznam nedostatků

Všechny nalezené nedostatky jsou shrnuty a uvedeny v tomto seznamu:

- globální třídní proměnné
- neefektivní inicializace testů
- nepřehledné nastavování proměnných
- nesprávné členění testů
- nedodržování jmenné konvence
- duplikování metod a konstant

Kapitola 5

Jiné časté chyby v testech

Zbytek práce se zaměří na nedostatky nalezené při analýze. Existují, ale i jiné chyby, které se často v Unit testech vyskytují. Těmto chybám se bude věnovat tato kapitola.

5.1 Práce s datumy

Aplikace či systémy často pracují s datumy. Aby testy správně ověřily funkčnost produktu je potřeba, aby byly datumy v testech správně použity.

Když je při psaní testů potřeba použít nějaké datum, existují dvě možnosti, jak to udělat. První možností je dynamické použití datumu, které je závislé na aktuálním datumu systému. Opačný přístup je statické použití datumu, které je vždy stejné a neměnné.

Je doporučené využívat výhradně statický přístup k používání datumů. Mezi výhody které tento přístup přináší, patří dřívější ověření funkčnosti mezních hodnot. Pokud by byl v testování využit dynamický způsob, tak je možné ověřit případy jako přestupný rok pouze při datumech ve kterých tento případ nastane. Statický přístup umožňuje tyto případy ověřit vždy a hned. [25]

Další výhodou statického přístupu je vyšší přehlednost v použitých datech, jelikož se v tomto přístupu datum nastavuje celé jako řetězec tak je na první pohled zřejmé jaké datum je použito.

5.2 Privátní metody

Privátní metody by se správně neměly testovat. V nejnutnějším případě to je umožněno pomocí reflexe.

Jedním z pravidel psaní Unit testů je testování jen veřejných metod. Privátní metody jsou implementační detaily třídy o kterých ostatní třídy ani objekty nemají žádné informace. A změna této vnitřní implementace by neměla vést ke změně testů.

Potřeba psát testy k privátním metodám poukazuje na některý z těchto problémů

- V privátní metodě se nalézá mrtvý kód [1].
- Privátní metoda je příliš komplexní a měla by patřit do jiné třídy.
- Metoda by neměla být privátní ale veřejná.

Pokud tedy existuje potřeba testovat privátní metodu, je doporučené se nejdříve zamyslet nad těmito možnými chybami a případně je opravit. [26]

[1] mrtvý kód - část kódu, který se nikdy neprovede, protože je nedosažitelný

Výpis kódu 5.1 Získání přístupu pro volání privátní metody

```
    private Method getSpecificMethod() throws NoSuchMethodException {
        Method method = Utils.class.getDeclaredMethod(
            "method name",
            class name.class);
        method.setAccessible(true);
        return method;
    }
```

Výpis kódu 5.2 Volání privátní metody v testu pomocí invoke

```
@Test
void privateMethodTest() throws NoSuchMethodException,
InvocationTargetException, IllegalAccessException {
    assertEquals(expected value,
                 getSpecificMethod().
                 invoke( class name.class,
                         tested method input));
}
```

5.2.1 Testování privátních metod pomocí reflexe

Pokud výše uvedené chyby nejsou odstraněny a je opravdu potřeba otestovat privátní metodu existuje možnost použití reflexe.

K tomu je potřebné, vytvořit přístup k této metodě, a aby tato privátní metoda měla viditelnost jako metoda veřejná. To je možné docílit pomocí obalující třídy *Utils* a jejích metod, což je ukázáno ukázce 5.1.

S takto připravenou metodou pro získání možnosti volání dané metody je možné za pomoci metody `invoke()` volat vybranou privátní metodu přímo v testech. V ukázce 5.2 je demonstrován způsob, jak tohoto volání docílit. Metoda *invoke()* se volá se skupinou parametrů. Prvním parametrem je třída, které tato privátní metoda patří, v případě že je metoda statická a není pro její volání potřebná žádná instance se používá *null*. Zbytek parametrů je pole vstupních parametrů volané metody. [26]

5.3 Mockování statických tříd

Při testování bývá využíváno mockování statických tříd. To ale v čistě objektově orientované architektuře programu by nemělo být třeba. Skutečnost že mockujeme statickou třídu poukazuje na možnou chybu v návrhu tříd nebo na code smell[2] v aplikaci. Třídy které závisejí na statické metodě vedou ke špatně testovatelnému kódu. Tyto závislosti by měly být nejlépe řešeny například injektáží. Tudíž před použitím mockingu statických tříd je vhodné se zamyslet nad refactoringem kódu, aby se stal lépe testovatelným. To jistě nelze provést ve všech případech, a proto existuje způsob jakým mockování statických tříd zajistit. Tento způsob nabízí framework Mockito. [27]

5.3.1 Konfigurace statického mockování

Před použitím statického mockování nabízeného frameworkem Mockito je potřeba aktivovat *MockMaker*. Pro aktivaci je třeba přidat do projektového repozitáře do složky s cestou

[2] code smell - jakákoli chyba poukazující na hlubší problém v zdrojovém kódu

Výpis kódu 5.3 Ukázka mockování statické metody

```
@Test
void mockingStaticMethodTest() {
    try (MockedStatic<static class name> mockedClass =
        Mockito.mockStatic(Static class name.class)) {
            mockedClass.when(static class name::
                static method name).
                thenReturn("wanted value");

        assertThat(StaticUtils.static method name()).
            isEqualTo("wanted value");
    }
}
```

„src/test/resources/mockito-extensions“ soubor pojmenovaný „org.mockito.plugins.MockMaker“ s jedinou řádkou „mock-maker-inline“. [27]

5.3.2 Implementace statického mockování

S takto nakonfigurovaným projektem je možné začít s mockováním pomocí metody *mockStatic()*. Tuto metodu je vhodné používat výhradně v *try* bloku, aby staticky namockovaný objekt nenarušil jiné spuštěné testy a byl hned po ukončení bloku vymazán.

V ukázce 5.3 je ukázáno jakým způsobem lze inicializovat statický mock metodou *mockStatic()* a jak pomocí kombinace metod *when().thenReturn()* ho nastavit k očekávanému chování. [27]

Kapitola 6
Řešení nedostatků

Tato kapitola se bude podrobně zabývat konkrétními chybami a nedostatky. Postupně pro všechny nalezené nedostatky z analýzy 4.3 se provede několik fází.

První fáze spočívá v popisu nalezené chyby. Ve druhé fázi se vysvětlí, jaké jsou hrozící problémy způsobené danou chybou. Třetí fáze slouží k navržení řešení pro danou chybu a poslední fáze bude obsahovat ukázky implementace navrženého řešení.

Nalezené nedostatky v testech nemusí vždy znamenat špatnou implementaci testů, ale mohou upozornit na návrhovou chybu ve zdrojovém kódu. Je tedy vhodné se před refactoringem testů zamyslet nad úpravou právě zdrojového kódu. Správně upravený zdrojový kód může sám o sobě odstranit daný testový nedostatek.

6.1 Globální třídní proměnné

6.1.1 Popis nedostatku

Nedostatkem „Globální třídní proměnné“ se rozumí globální používání třídních proměnných testovacích tříd v jednotlivých testovacích metodách. Problémové jsou především proměnné, které slouží pouze jako kontejnery dat. Tyto proměnné se poté využívají napříč všemi testovacími metodami v dané testovací třídě a jejich jedno nastavení se přenese i do ostatních testů. To neplatí pro funkčnější proměnné (např. service, facade, repository), které třídě dodávají nějakou funkcionalitu a v testovacích metodách se používají převážně namockované s nastaveným chováním, které může být ve všech testových metodách stejné.

6.1.2 Hrozící problémy

Používání těchto třídních proměnných globálně napříč více testy způsobí vzájemnou závislost testů. Pokud se v jednom testu do těchto objektů cokoliv nastaví, tak v druhém testu tyto dané objekty zůstanou nastaveny. To zapříčiní jiné než očekávané chování druhého testu.

Takto globálně využívané proměnné napříč více testy mohou na první pohled vypadat v pořádku. Při hromadném spuštění testů mohou všechny testovací metody skončit jako úspěšné, ale toto se může změnit při jednotlivém spuštění testů. Testy které byly úspěšné jen díky předem nastaveným objektům z předchozích testů nyní skončí neúspěšně. Stejná situace nastane při hromadném spuštění testů, ale v jiném pořadí. Některé testy které byly závislé na minulých testech skončí neúspěšně a některé testy zůstanou úspěšné. Toto chování není od Unit testů chtěné a je na obtíž.

Dalším problémem takto navzájem závislých testů je nemožnost je spouštět paralelně. Paralelní spouštění by urychlilo celý proces testování podle toho na kolika jednotkách by bylo provozováno. Ale při paralelním spuštění by testy neproběhli úspěšně a tudíž taková možnost kvůli globálním třídním proměnným není dostupná.

6.1.3 Návrh řešení

Způsobem řešení této problematiky je rozdělení třídních proměnných do dvou kategorií.

První kategorií jsou funkční proměnné, které se používají pro nějakou přidanou funkcionalitu. Tyto proměnné se ponechají ve stejném stavu.

Druhou kategorií jsou proměnné sloužící jako kontejnery dat, které se využívají v testované funkcionalitě. Ty budou odstraněny a nahrazeny v jednotlivých testovacích metodách. V každé testové metodě je tedy třeba vytvořit nové objekty, které budou odpovídat odstraněným proměnným a nastavit je do požadovaného stavu. Pokud touto náhradou vznikne opakované vytváření a nastavování stejného typu objektu je doporučené přesunout tento opakující se kód do samostatné pomocné funkce.

Tato změna přinese ztrátu nechtěné vzájemné závislosti a zpřehlední testovací metody, jelikož v každém testu bude přímo poznat jaké objekty jsou vytvářeny a jaké jsou jejich počáteční hodnoty.

6.1.4 Ukázka

Ukázková implementace se týká třídy „Person“, která má dvě proměnné datového typu *String*. V kódu 6.1 je použita globální proměnná, které se v následujících testech nastavují hodnoty a ověřuje se správnost daných hodnot. Tato proměnná je podle výše uvedených důvodů nežádoucí.

S využitím navrženého postupu pro chybu globální proměnné je v ukázce 6.2 tato implementace upravena do stavu, ve kterém se v každém testu vytváří nová lokální proměnná. Přínosem je ztráta vzájemné závislosti testů.

Výpis kódu 6.1 Testová třída s globální třídní proměnou

```
class PersonTest {

    public Person person;

    @BeforeAll
    public static void initialize() {
        person = new Person();
    }

    @Test
    public void personNameIsGeorge() {
        person.setName("George");
        assertEquals("George", person.name());
    }
    @Test
    public void personSurnameIsSmith() {
        person.setSurname("Smith");
        assertEquals("Smith", person.surname());
    }
}
```

Výpis kódu 6.2 Testovací třída po odstranění globální proměnné

```
class PersonTest {

    @Test
    public void personNameIsGeorge() {
        Person person = new Person();
        person.setName("George");
        assertEquals("George", person.name());
    }
    @Test
    public void personSurNameIsSmith() {
        Person person = new Person();
        person.setSurname("Smith");
        assertEquals("Smith", person.surname());
    }
}
```

6.2 Neefektivní inicializace testů

6.2.1 Popis nedostatku

Inicializací testů se rozumí vytvoření, nastavení a namockování všech potřebných objektů a proměnných které jsou potřebné pro dané testy. Tato příprava může být zdlouhavá a časově náročná, proto je požadavkem, aby inicializace nebyla zbytečně neefektivní. Inicializaci testů je možné provádět více způsoby. V tomto projektu se pro ní využívají anotace *@BeforeEach* 3.2.1.2 a *@BeforeAll* 3.2.1.3 z frameworku JUnit5. Chyba které byla objevena je neefektivita, způsobena přílišným využíváním anotace *@BeforeEach*. V každém testovém souboru se obvykle nachází jedna metoda s touto anotací, která v sobě obsahuje všechny potřebné operace.

6.2.2 Hrozící problémy

Tento způsob inicializace způsobuje, že veškeré mockování a vytváření objektů probíhá před každým jedním testem a zabírá spoustu času. Z tohoto důvodu je testování pomalé a zdržuje i ostatní části vývoje softwaru.

Dalším problémem je špatně měnitelný systém pro inicializaci, který nenabízí dostatek možností pro úpravu.

6.2.3 Návrh řešení

Cílem navrženého řešení tohoto problému je odstranění neefektivity. Řešení spočívá v přidání další inicializační metody s anotaci *@BeforeAll* která se provede pouze jednou. Do této nové metody se přesunou všechny operace, které nejsou potřeba provádět mezi každým testem (např. vytvoření třídních proměnných, inicializování mockovaných objektů).

Mohou nastat dvě situace. V prvním případě nezůstane v původní metodě žádná operace. Původní inicializační metoda bude odstraněna a zůstane pouze nová. V druhém případě se nepřesunou všechny operace, tudíž testovací třída si ponechá obě dvě inicializační metody. Tato situace ničemu nevadí a je v pořádku. Při spuštění testů bude první prováděnou inicializační metodou ta s anotací *@BeforeAll*.

Pomocí těchto úprav urychlí proces testování a předejde se zbytečného opakovaného volání konkrétních operací. Dalším přínosem této změny je vytvoření funkčnějšího, přehlednějšího a lépe upravovatelného systému pro inicializaci testů.

6.2.4 Ukázka

Chybná implementace ve výpisu kódu 6.3 obsahuje veškeré potřebné operace pro přípravu prostředí před testy. Je použita anotace pro provádění před každým testem. Nejdříve se vytvoří namockované proměnné, poté se vytvoří a nastaví potřebná data pro provedení testu pomocí metody *prepareTestData()* a na závěr se nastaví chování mockovaných proměnných při operaci *getNameById()*.

Po úpravě pomocí doporučeného postupu se implementace změní na ukázku 6.4. Původní metoda se rozdělí na dvě samostatné metody, kde každá používá jinou anotaci, podle potřeby volaných operací. Do metody s anotací *@BeforeAll* se vloží operace které stačí volat pouze jednou. A do druhé metody s anotací *@BeforeEach* se dostanou operace, které jsou potřeba volat před každým testem.

Výpis kódu 6.3 Neefektivní inicializace testů

```
@BeforeEach
public void initialize() {

    service = Mockito.mock(Service.class);
    prepareTestData();
    Mockito.when(service.getNameById(JOHN_ID))
        .thenReturn(new Osoba(John));

    Mockito.when(service.getNameById(JAMES_ID))
        .thenReturn(new Osoba(James));
}
```

Výpis kódu 6.4 Opravená inicializace testů

```
@BeforeAll
public void initializeBeforeAll() {

    service = Mockito.mock(Service.class);

    Mockito.when(service.getNameById(JOHN_ID))
        .thenReturn(new Osoba(John));

    Mockito.when(service.getNameById(JAMES_ID))
        .thenReturn(new Osoba(James));
}

@BeforeEach
public void initializeBeforeEach() {

    prepareTestData();
}
```

6.3 Nepřehledné nastavování proměnných

6.3.1 Popis nedostatku

Nedostatek „Nepřehledné nastavování proměnných“ se zaměřuje na zpřehlednění testového kódu. Poukazuje na časté a zdlouhavé nastavování proměnných a objektů používaných v testových metodách. Tyto objekty mají obvykle více proměnných a nastavení všech těchto hodnot zabírá mnoho řádků. Mezi těmito řádky se poté jednoduše přehlédnou ostatní části testu.

6.3.2 Hrozící problémy

Existence těchto částí kódu prudce zmenšuje čitelnost a přehlednost daných testových metod. Tím se zpomaluje postup vývojáře při práci na testech nebo i při snaze testy zkontrolovat.

U tohoto nedostatku může dojít k dvěma hlavním problémům. Prvním je nastavování objektů s mnoha proměnnými. V tomto případě je hlavním problémem počet řádků, který je touto operací zabrán. Tento počet často překoná rozsah celého zbytku testu. Tímto se test stane velmi nečitelným.

Druhým možným problémem je opět špatná čitelnost kódu, ale způsobena jiným důvodem. Nečitelnost se může objevit i při nastavování jedné samostatné proměnné. Pokud je daná proměnná nastavována zdlouhavě pomocí dlouhé kaskády metod.

6.3.3 Návrh řešení

Pro řešení této chyby existují dva způsoby.

Prvním způsobem je zavedení pomocných metod. Tyto metody budou obsahovat veškerý kód nutný pro nastavení celého objektu. Je vhodné metodu navrhnout, takovým způsobem, aby hodnoty nastavované do proměnných mohly být parametrizované, tedy aby byly řízeny pomocí vstupních proměnných. Tyto metody je potřeba vhodně pojmenovávat, aby jména metod odpovídala objektům které tyto metody vytvoří.

Druhý způsob je zavedení pomocných konstant. Tyto konstanty slouží pro nastavování konkrétních hodnot do jednotlivých proměnných. Pro konstanty a jejich pojmenovávání platí stejná logika jako u pomocných metod. Správně pojmenovaná konstanta může v testovém kódu jednoduše nahradit dlouho kaskádu metod, nebo těžce pochopitelnou vloženou hodnotu.

6.3.4 Ukázka

Ve výpisu kódu 6.5 je ukázán test metody *isBornBefore2000()*, která jako vstupní parametr bere instanci třídy „Person“ a vrací „boolean“ podle toho jestli daná osoba se narodila před rokem 2000. V testu se nejdříve vytváří instance testované validační třídy. Následně se vytvoří instance třídy Person, a nastaví se všechny její třídní proměnné. Nakonec se ověří funkčnost testované metody.

Nastavování třídních proměnných třídy Person je zdlouhavé a v testovém kódu překáží. Další chybou je nastavování proměnné „BirthDate“, které není přímočaré a je matoucí a nepřehledné. Pomocí doporučeného postupu je tento kód zrefaktorován a ukázán ve výpisu kódu 6.6.

Pro nastavení data narození byla zavedena konstanta pojmenovaná jako *DATE_OF_BIRTH*, která může být použita i v jiných případech.

Pro proměnnou „Id“ byla také zavedena pomocná konstanta se jménem *PERSON_ID*.

Celé nastavování instance Person je přesunuto do pomocné metody *createPerson()*. Tato nová metoda podporuje parametrizaci proměnné data narození a může být použita i v dalších testech dané třídy.

Testová metoda „validateTest()“ byla pomocí refactoringu zpřehledněna a zkrácena na pouhé tři řádky, aniž by ztratila jakoukoliv funkcionalitu.

Výpis kódu 6.5 Nepřehledné nastavování proměnných a objektů

```
class ValidatorTest{

    @Test
    public void validateTest() {
        PersonValidator validator = new PersonValidator();

        Person person = new Person();
        person.setName("John");
        person.setSurname("Smith");
        person.setId(27);
        person.setBirthDate(LocalDate.of(2001, 1, 1));

        assertEquals(false, validator.isBornBefore2000(person));
    }
}
```

Výpis kódu 6.6 Opravené nastavování proměnných a objektů

```
class ValidatorTest{

    public static final LocalDate DATE_OF_BIRTH =
        LocalDate.of(2001, 1, 1);
    public static final Integer PERSON_ID = 27;

    Person createPerson(LocalDate dateOfBirth) {
        Person person = new Person();
        person.setName("John");
        person.setSurname("Smith");
        person.setId(PERSON_ID);
        person.setBirthDate(dateOfBirth);
        return person;
    }

    @Test
    public void validateTest() {
        PersonValidator validator = new PersonValidator();

        Person person = new createPerson(DATE_OF_BIRTH);

        assertEquals(false, validator.isBornBefore2000(person));
    }
}
```

6.4 Nesprávné členění testů

6.4.1 Popis nedostatku

Tento nedostatek se zabývá pravidlem, že Unit testy konkrétní metody by měly ověřovat pouze kód dané metody, nikoliv kód metod jiných tříd volaných zevnitř testované metody. Metody volané z vnitřku mají být otestovány zvlášť v samostatné testovací třídě zaměřené na konkrétní metodu. Tudíž testy by měli být správně a pravidelně členěné do balíčků, tříd a metod podle zaměření testů. Toto členění může být na každé aplikaci jiné ale je důležité ho dodržovat.

6.4.2 Hrozící problémy

Hlavní problémy které špatným členěním testů mohou vznikat jsou dva.

První problém je nedostatečné otestování z vnitřku volaných metod. Jelikož tyto metody nemají svoje vlastní testovací třídy, tudíž nemají ani žádné Unit testy na ně zaměřené. Je tedy možné, že nastane situace, že nebudou ověřeny všechny možné možnosti daných metod.

Druhým problémem je nepřehlednost testů. Jelikož testy z vnitřku volané metody jsou roztroušeny po jiných testovacích třídách a nemají svůj vlastní soubor, tak pro vývojáře vzniká problém dané testy vůbec najít a to mu značně znesnadní práci.

6.4.3 Návrh řešení

Této chybě lze zamezit už při samém psaní testů, tím že se každé třídě či metodě vytvoří samostatný testový soubor, ve kterém budou umístěny veškeré Unit testy zaměřené na danou třídu či metodu.

Pokud je tato chyba objevena až v existující implementaci testů, tak řešením je, odstranění testů z nesprávných míst a jejich vložení do nových samostatných souborů zaměřených na testování daných metod. Je pravděpodobné, že nebude stačit testy pouze přesunout, ale že je bude potřeba korektním způsobem poupravit.

6.4.4 Ukázka

Výpis kódu 6.7 Nesprávné členění testů

```
class Calculator {
    private Validator validator;

    public int divide (Integer divident, Integer divisor)
    throws IllegalArgumentException {

        validator.validateNotZero(divisor);
        return divident / divisor;
    }
}

class CalculatorTest {

    @Test
    public void divideTest_divisorIsZero() {
        Calculator calc = new Calculator();

        Assertions.assertThrows(IllegalArgumentException.class,
            () -> {
                calc.divide(15, 0);
        });
    }
}
```

Výpis kódu 6.8 Opravené členění testů

```
class ValidatorTest{

    @Test
    public void validateNotZeroTest() {
        Validator validator = new Validator();

        Assertions.assertThrows(IllegalArgumentException.class,
            () -> {
                validator.validateNotZero(0);
        });
    }
}
```

V ukázce kódu 6.7 je předveden test metody *divide()* v třídě „Calculator“. Konkrétní test ověřuje chování validační metody *validateNotZero()* při hodnotě dělitele nula. Tato validační metoda v případě, kdy hodnota je nenulová nevrací nic a v případě, že vstupní hodnota je nula vyhazuje instanci výjimky „IllegalArgumentException“.

Tento test není správný, jelikož by měl správně patřit do testů validační třídy „Validator“. Tím nedodržuje správné členění testů.

Pomocí doporučeného postupu pro řešení této chyby je vytvořena ukázka kódu 6.8. Výše uvedený test je upraven a přesunut do samostatné testové třídy validátoru a odstraněn z původního testové třídy.

6.5 Nedodržování jmenné konvence

6.5.1 Popis nedostatku

Jmenná konvence je zavedený systém pro pojmenovávání metod, tříd, testů i proměnných. Je zaváděna, aby kód který je psán vícečlenným týmem byl srozumitelný a čitelný pro každého člena týmu. Tento systém je zpravidla dodržován ve zdrojovém kódu aplikace, ale jeho přínos je i v kódu testovém. Tento nedostatek se tedy zabývá dodržováním stejného pojmenovávání stejně fungujících funkcionalit, metod a proměnných v testech. Pro příklad metody které vytvářejí a nastavují nějaký mockovaný objekt se hromadně jmenují „mock...()“.

6.5.2 Hrozící problémy

V případě, že daný systém pro pojmenovávání není dodržován může dojít k zhoršení čitelnosti kódu. V některých případech může dojít i ke zmatení vývojáře. Vývojář si může špatně pojmenovanou metodu vyložit ve špatném smyslu a použít ji nesprávně, to může zapříčinit chyby v testech. Výhodou zavedené jmenné konvence je urychlení postupu vytváření kódu, jelikož vývojář nemusí přemýšlet nad vhodným jménem pro dané metody, proměnné nebo třídy.

6.5.3 Návrh řešení

Řešením chyby různě pojmenovaných, ale stejně pracujících funkcionalit je zavedení hromadné jmenné konvence hned na začátku vývoje aplikace.

Pokud se tato chyba objeví v již existující implementaci je třeba kód postupně projít a všechny názvy upravit nebo sjednotit.

Pro určitý typ metod (např. inicializační metody) může být vytvořen jejich předpis. K tomu

lze použít abstraktní metodu[1] nějakého společného předka daných testových tříd. V testových třídách se následně tato metoda implementuje pomocí anotace *@Override*. Další možností je vytvoření rozhraní[2], které bude sloužit podobným způsobem.

6.5.4 Ukázka

V ukázkovém kódu 6.9 jsou předvedeny dvě jednoduché testové třídy. Každá vlastní jednu metodu pro inicializaci a jeden test. V testech jsou používány pomocné metody pro vytvoření objektů *animalCreator()* a *createPerson()*. Tyto testové třídy nedodržují stejný způsob pojmenovávání právě v inicializačních metodách a v pomocných metodách.

Pomocí navrženého postupu je vytvořena ukázková implementace 6.10 ve které je vytvořen společný předek „TestInitializator“ s abstraktní metodou *initialize()*. Předek zařídí sjednocení jmen pro inicializaci testů. Další úpravou je sjednocení jmen pomocných funkcí na „create...()“.

Výpis kódu 6.9 Nedodržování jmenné konvence v testech

```
class AnimalTest{

    @BeforeEach
    public void prepareData {}

    @Test
    public void animalTest() {
        Animal animal = animalCreator();
        assertEquals("expected value", animal.method());
    }
}

class PersonTest{

    @BeforeEach
    public void initialize {}

    @Test
    public void personTest() {
        Person person = createPerson();
        assertEquals("expected value", person.method());
    }
}
```

[1] abstraktní metoda - metoda, která nemá vyplněnou implementaci, ale pouze vytváří předpis metody
[2] interface - třída, která může vlastnit pouze konstanty a neimplementované metody

Výpis kódu 6.10 Opravené testy s jednotnou jmennou konvencí

```
abstract class TestInitializor{

    @BeforeEach
    public abstract void initialize() {}
}

class AnimalTest extends TestInitializor{

    @Override
    @BeforeEach
    public void initialize {}

    @Test
    public void animalTest() {
        Animal animal = createAnimal();
        assertEquals("expected value", animal.method());
    }
}

class PersonTest extends TestInitializor{

    @Override
    @BeforeEach
    public void initialize {}

    @Test
    public void personTest() {
        Person person = createPerson();
        assertEquals("expected value", person.method());
    }
}
```

6.6 Duplikování metod a konstant

6.6.1 Popis nedostatku

Při psaní testů je vhodné využívat konstanty a pomocné metody dle sekce 6.3. Není ale vhodné, aby se stejná konstanta nebo metoda objevovala v kódu dvakrát či vícekrát. Je možné, že daná konstanta nebo metoda je potřebná k implementaci více testů. Tím se stane, že se stejný kód objeví ve všech souborech, kde je používán. Tím vznikne duplikace kódu a ten vede na hrozící problémy.

6.6.2 Hrozící problémy

Problémů které vznikají duplikací kódu je vícero. První z nich vzniká při snaze danou metodu, či konstantu pouze přejmenovat. To musí být provedeno na všech místech, kde je metoda nebo konstanta umístěna a některý z případů může být zapomenuto. Tím vznikne problém.

Další problém je velice podobný a vzniká při potřebě editovat duplikovaný kód.

Obecně řečeno duplikace kódu není správná a mělo by se jí vyhýbat. Projekt ve kterém je duplikovaný kód je špatně editovatelný, tedy má špatný objektově orientovaný návrh.

6.6.3 Návrh řešení

Řešením této chyby je vytvoření pomocné třídy pro shromáždění všech potřebných pomocných metod a konstant. Tato pomocná třída by měla být pojmenována způsobem, ze kterého bude jednoduše poznat k čemu slouží (např. „Utility“). Do této třídy se následně vloží veškeré proměnné metody a konstanty, které nepatří přímo do testových souborů.

6.6.4 Ukázka

Výpis kódu 6.11 Duplikování metod a konstant v testech

```
class CityAreaTest {

    public static final Integer LONDON_POPULATION = 500000;
    public static final Integer LONDON_AREA = 1000;

    public City createCity() {
        City city = new City();
        city.setName("London");
        city.setArea(LONDON_AREA);
        city.setPopulation(LONDON_POPULATION);
        return city;
    }

    @Test
    public void cityAreaTest() {
        City city = createCity();
        assertEquals(LONDON_AREA, city.area());
    }
}

class CityPopulationTest {

    public static final Integer LONDON_POPULATION = 500000;
    public static final Integer LONDON_AREA = 1000;

    public City createCity() {
        City city = new City();
        city.setName("London");
        city.setArea(LONDON_AREA);
        city.setPopulation(LONDON_POPULATION);
        return city;
    }

    @Test
    public void cityPopulationTest() {
        City city = createCity();
        assertEquals(LONDON_POPULATION, city.area());
    }
}
```

Výpis kódu 6.12 Opravené testy bez duplikování kódu

```
class CityUtility {

    public static final Integer LONDON_POPULATION = 500000;
    public static final Integer LONDON_AREA = 1000;

    public City createCity() {
        City city = new City();
        city.setName("London");
        city.setArea(LONDON_AREA);
        city.setPopulation(LONDON_POPULATION);
        return city;
    }
}

class CityAreaTest {

    @Test
    public void cityAreaTest() {
        City city = CityUtility.createCity();
        assertEquals(CityUtility.LONDON_AREA, city.area());
    }
}

class CityPopulationTest {

    @Test
    public void cityPopulationTest() {
        City city = CityUtility.createCity();
        assertEquals(CityUtility.LONDON_POPULATION, city.area());
    }
}
```

Ve výpisu z kódu 6.11 jsou vytvořeny dvě testovací třídy. Každá ověřuje nějakou funkci třídy „City“. Obě tyto třídy využívají ve své implementaci stejné konstanty a stejnou pomocnou metodu pro vytvoření výchozího testovacího města. Tyto části kódu jsou tedy duplikáty.

Po provedení doporučeného postupu pro opravu této chyby je vytvořena ukázka opravené implementace 6.12. V této nové implementaci je vytvořena pomocná třída „CityUtility“, do které jsou přidány společné konstanty a pomocná metoda. A původní testy jsou upraveny s ohledem na existenci této nové třídy.

Tímto procesem byl z ukázky odstraněn všechen duplikovaný kód a samotné testové třídy se zpřehlednily.

Kapitola 7
Závěr

Závěrečná kapitola se bude věnovat popisem provedeného refactoringu Unit testů na aplikaci. Následně se zaměří na zhodnocení úspěšnosti refactoringu s ohledem na efektivitu a přehlednost testů.

7.1 Provedený refactoring

V rámci práce byla zrefaktorována část Unit testů aplikace. Refactoring byl proveden podle navržených strategií a postupů v kapitole 6. Konkrétně byl refactoring zaměřen na balíček „Přestupek“.

Původní testy, které v jednom balíčku sjednocovaly testy validací, třídy „Service“ a třídy „Facade“. Byly rozděleny do více balíčků podle testované třídy.

Dále byly odstraněny veškeré nežádoucí třídní proměnné ze všech vzniklých testových tříd podle návrhu 6.1.3.

Podle návrhu řešení inicializace testů 6.2.3 byla provedena úprava inicializačních metod testových tříd. Ve většině případů se metoda s anotací *@BeforeEach* předělala na inicializační metodu s anotací *@BeforeAll*. Pro sjednocení jmen inicializačních metod byla jako společný předek zavedena abstraktní třída „TestInitializer“.

Konstanty využívané napříč testy tohoto balíčku byly přesunuty do nově vzniklé pomocné třídy „PrestupekTestUtil“.

Testy validací byly upraveny podle postupu 6.4.3. Kvůli této úpravě vznikly nové testové třídy v odděleném balíčku.

Další úprava byla provedena podle postupu 6.3.3. Nastavování objektů bylo přesunuto do pomocných metod a často měněné hodnoty jsou v těchto metodách parametrizovány. Pro některé nastavované hodnoty byly vytvořeny konstanty.

7.2 Porovnání výsledků

7.2.1 Efektivita

Pro zhodnocení úspěšnosti refactoringu z hlediska efektivity je potřebné porovnat čas běhu testů v nové implementace oproti času běhu stejných testů na staré implementaci. Jelikož časy běhů jednotlivých testů jsou velice krátké a při každém měření může vzniknout nějaká odchylka, tak měření bude provedeno opakovaně za účelem získání více dat k analýze. Následně se z těchto dat získají průměrné časy běhů testů.

Očekávaným výsledkem tohoto měření je lehké zlepšení v nové implementaci oproti té staré. Hlavním důvodem zlepšení by měla být změna inicializací testů.

Tabulka 7.1 Tabulka naměřených délek časů běhů testů

měření	1	2	3	4	5	6	7	8	9	10	**průměr**
stará verze	653	648	643	648	655	654	643	671	641	648	**650,4**
nová verze	72	66	64	62	70	68	74	68	77	62	**68,3**

V tabulce 7.1 jsou uvedeny naměřené hodnoty. Veškeré časové údaje jsou uváděny v milisekundách (ms). Druhý řádek tabulky obsahuje jednotlivé časy běhů testů ve staré implementaci před začátkem refactoringu a třetí řádek obsahuje časy běhů testů po provedeném refactoringu.

Je vidět, že refactoring z hlediska efektivity byl úspěšný, jelikož průměrný čas běhu testů v balíčku „Přestupek“ se zlepšil téměř 10krát. Toto zlepšení bylo dosaženo hlavně úpravou v inicializaci testů, konkrétně využitím anotace *@BeforeAll* namísto anotace *@BeforeEach*. Navržený postup v sekci 6.2.3 je tedy přínosný.

7.2.2 Přehlednost

Porovnání přehlednosti není jednoduše měřitelné jako například efektivita. Z tohoto důvodu bude úspěšnost refactoringu z pohledu přehlednosti vyhodnocena slovně.

Z původních testů v jednom balíčku vznikly testy ve třech balíčcích. V jednotlivých testových třídách se zmenšil počet testových metod, jelikož testy byly rozčleněny podle testovaných funkcionalit. Tím se zvětšil počet testových tříd z původních šesti a jedné inicializační na aktuálních patnáct testových tříd, dvě inicializační třídy a jednu pomocnou. v samotných testových metodách byl ve většině případech zmenšen počet řádků. Konstanty byly sjednoceny v nové pomocné třídě a byla vytvořena abstraktní třída pro inicializaci testů.

Všechny tyto změny vedly svým způsobem ke zlepšení přehlednosti Unit testů v tomto balíčku, tudíž z tohoto hlediska byl provedený refactoring úspěšný. Jelikož celý refactoring byl proveden pomocí navržených postupů z kapitoly 6, tak celkový výstup práce je považován za funkční a přínosný.

Bibliografie

1. KITNER, Radek. *O čem je testování software? (pro znalé) - Radek Kitner* [online]. Idk: Idk, 2021 [cit. 2023-04-24]. Dostupné z: https://kitner.cz/testovani_softwaru/co-je-testovani-software/.
2. SKILLMEA. *Co je testování softwaru? | Skillmea* [online]. 2021. [cit. 2023-04-24]. Dostupné z: https://skillmea.cz/blog/co-je-testovanie-softveru.
3. MLEJNEK, Jiří. *Softwarové inženýrství I: Přednáška 9* [online]. 2022. [cit. 2023-04-24]. Dostupné z: https://moodle-vyuka.cvut.cz/pluginfile.php/532124/mod_resource/content/3/09.prednaska.pdf.
4. KITNER, Radek. *Typy testování software* [online]. 2021. [cit. 2023-04-24]. Dostupné z: https://kitner.cz/testovani_softwaru/typy-testovani-software-trideni-testu/.
5. HLAVA, Tomáš. *Integrační Testování | Testování softwaru* [online]. 2011. [cit. 2023-04-24]. Dostupné z: http://testovanisoftwaru.cz/tag/integracni-testovani/.
6. DIFFERENCE, SPOT THE. *Rozdíl mezi statickým a dynamickým testováním* [online]. 2023. [cit. 2023-04-24]. Dostupné z: https://cs.spot-the-difference.info/difference-between-static.
7. ZAPTEST. *Co je automatizace testování? Jednoduchý průvodce bez žargonu* [online]. 2023. [cit. 2023-04-24]. Dostupné z: https://www.zaptest.com/cs/co-je-automatizace-testovani-jednoduchy-pruvodce-bez-zargonu.
8. ZELINKA, Bořek. *Testování softwaru* [online]. UNICORN systems, 2013 [cit. 2023-04-24]. Dostupné z: https://d3s.mff.cuni.cz/legacy/teaching/commercial_workshops/previous/1213/zelinka-zajisteni_kvality_softwarovych_produktu.pdf.
9. BULEJ, Lubomír. *Doporučené postupy v programování* [online]. 2023. [cit. 2023-04-24]. Dostupné z: https://d3s.mff.cuni.cz/f/teaching/nprg043/05-testing.html.
10. BECHTOLD, Stefan; BRANNEN, Sam; LINK, Johannes; MERDES, Matthias; PHILIPP, Marc; RANCOURT, Juliette de; STEIN, Christian. *JUnit 5 User Guide* [online]. 2022. [cit. 2023-04-25]. Dostupné z: https://junit.org/junit5/docs/current/user-guide/%5C#overview.
11. RIMENTI, Donato. *JUnit 5 @Test Annotation | Baeldung* [online]. 2023. [cit. 2023-04-25]. Dostupné z: https://www.baeldung.com/junit-5-test-annotation.
12. *BeforeEach (JUnit 5.0.2 API)* [online]. 2017. [cit. 2023-04-25]. Dostupné z: https://junit.org/junit5/docs/5.0.2/api/org/junit/jupiter/api/BeforeEach.html.
13. *BeforeEach (JUnit 5.0.1 API)* [online]. 2017. [cit. 2023-04-25]. Dostupné z: https://junit.org/junit5/docs/5.0.1/api/org/junit/jupiter/api/BeforeEach.html.

14. GONZALEZ, Marcos Lopez. *@Before vs @BeforeClass vs @BeforeEach vs @BeforeAll | Baeldung* [online]. 2022. [cit. 2023-04-25]. Dostupné z: https://www.baeldung.com/junit-before-beforeclass-beforeeach-beforeall.
15. *BeforeAll (JUnit 5.0.1 API)* [online]. 2017. [cit. 2023-04-25]. Dostupné z: https://junit.org/junit5/docs/5.0.1/api/org/junit/jupiter/api/BeforeAll.html.
16. *AfterEach (JUnit 5.0.1 API)* [online]. 2017. [cit. 2023-04-25]. Dostupné z: https://junit.org/junit5/docs/5.0.1/api/org/junit/jupiter/api/AfterEach.html.
17. *AfterAll (JUnit 5.0.1 API)* [online]. 2017. [cit. 2023-04-25]. Dostupné z: https://junit.org/junit5/docs/5.0.1/api/org/junit/jupiter/api/AfterAll.html.
18. BALASUBRAMANIAM, Vivek. *@TestInstance Annotation in JUnit 5 | Baeldung* [online]. 2022. [cit. 2023-05-01]. Dostupné z: https://www.baeldung.com/junit-testinstance-annotation.
19. *Co je maven? - definice z techopedie - Rozvoj 2023* [online]. 2023. [cit. 2023-04-25]. Dostupné z: https://cs.theastrologypage.com/maven.
20. APACHE. *Maven – Introduction to the Build Lifecycle* [online]. 2023. [cit. 2023-04-25]. Dostupné z: https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html.
21. KUNČAR, Petr. *Mockito - unit test framework* [online]. 2023. [cit. 2023-04-25]. Dostupné z: https://www.itnetwork.cz/java/testovani/mockito-unit-test-framework.
22. JAVATPOINT. *Mockito Tutorial | Mockito Framework Tutorial - Javatpoint* [online]. 2021. [cit. 2023-04-25]. Dostupné z: https://www.javatpoint.com/mockito.
23. JAVATPOINT. *Methods of Mockito - Javatpoint* [online]. 2021. [cit. 2023-04-25]. Dostupné z: https://www.javatpoint.com/methods-of-mockito.
24. BAELDUNG. *Mockito.mock() vs @Mock vs @MockBean | Baeldung* [online]. 2023. [cit. 2023-04-25]. Dostupné z: (https://www.baeldung.com/java-spring-mockito-mock-mockbean.
25. PETERSON, Andy. *You Should Use Static Dates For Your Unit Tests* [online]. 2016. [cit. 2023-04-25]. Dostupné z: https://spin.atomicobject.com/2016/12/23/static-dates-unit-tests/.
26. BAELDUNG. *Unit Test Private Methods in Java | Baeldung* [online]. 2022. [cit. 2023-04-25]. Dostupné z: https://www.baeldung.com/java-unit-test-private-methods.
27. COOK, Jonathan. *Mocking Static Methods With Mockito | Baeldung* [online]. 2022. [cit. 2023-04-25]. Dostupné z: https://www.baeldung.com/mockito-mock-static-methods.

Obsah přiloženého média

- `license.txt` ..... licenční ujednání pro SW přílohy
- `readme.txt` ..... stručný popis obsahu média
- `src`
  - `capekj14` ..... zdrojová forma práce ve formátu LaTeX
  - `tests` ..... zdrojové kódy implementace
- `text` ..... text práce
  - `BP_Capek_Jakub_2023.pdf` ..... text práce ve formátu PDF