Univerzita Hradec Králové
Fakulta informatiky a managementu
Katedra informatiky a kvantitativních metod

Automatizované testování webových aplikací psaných v PHP

Diplomová práce

Autor: Bc. Radek Meduna

Studijní obor: Aplikovaná informatika

Vedoucí práce: doc. Ing. Filip Malý, Ph.D.

Hradec Králové duben 2021

Prohlášení

Prohlašuji, že jsem diplomovou práci zpracoval samostatně a s použitím uvedené literatury.

V Hradci Králové dne 17. dubna 2021 Radek Meduna

Poděkování

V této sekci bych rád poděkoval svému vedoucímu diplomové práce doc. Ing. Filipu Malému, Ph.D., za věcné konzultace týkající se obsahu práce. Dále bych rád poděkoval své rodině, která mě vřele podporovala při cestě za studiem a byla mi vždy velkou oporou. Klíčovou roli při psaní této práce sehrála i pandemie, která mi zajistila minimální rozptýlení při psaní následujících řádků.

Anotace

Diplomová práce se zabývá tématem automatizovaného testování webových aplikací psaných v programovacím jazyce PHP. Popisuje jednotlivé typy testů s různou granularitou, které při správném využití dokáží výrazně snížit chybovost webové aplikace a přispět k její udržitelnosti. Práce předkládá teoretický podklad i praktický návod na nasazení jednotkového testování, integračního testování, mutačního testování nebo využití statické analýzy kódu za pomoci nástrojů s otevřeným zdrojovým kódem. Tyto zmíněné postupy byly nasazeny na webové aplikaci z oblasti e-commerce a výrazně zvýšily detekci chyb v raných fázích vývoje softwaru. Hlavním přínosem této diplomové práce je popis různých typů testů, benefitů, které jejich implementace přináší a popis jejich automatizace v rámci kontinuální integrace.

Annotation

Title: Automated testing of web applications written in PHP

The diploma thesis deals with the topic of automated testing of web applications written in the PHP programming language. Describes individual types of tests with different granularity, which can significantly reduce the error rate of a web application and contribute to its sustainability. The thesis presents a theoretical basis and practical instructions for the deployment of unit testing, integration testing, mutation testing or the use of static code analysis using open source tools. These procedures were deployed on a web application in the field of e-commerce and significantly increased the detection of errors in the early stages of software development. The main benefit of this thesis is a description of the different types of tests, the benefits that their implementation brings and a description of their automation in continuous integration.

Obsah

1 Úvod do testování

„Chceme-li to s kvalitou myslet vážně, je na čase přestat s hledáním chyb a začít zabraňovat jejich vzniku.“—**Alan Page**

Automatizované testování dnes již neodmyslitelně patří mezi základní stavební kameny úspěšných softwarových produktů. Automatizované testování softwaru odhaluje chyby v raných fázích vývoje a tím zajišťuje koncovému uživateli příjemnější a pohodlnější používání produktu. Dobře otestovaný kód přináší benefity nejen uživatelům, ale i samotným vývojářům. Otestovaný kód se mnohem lépe udržuje, rozšiřuje, upravuje a poskytuje vývojářům mnohem větší sebevědomí při implementaci nových funkcionalit. Pravidelné a systematické psaní testovacích případů dává smysl i z ekonomického hlediska. Z dlouhodobého pohledu je pro firmu mnohem nákladnější oprava již existujících chyb, než jejich předcházení.

1.1 Ekonomické hledisko testování

Automatizované testování hraje klíčovou roli v nákladech vynaložených na vývoj softwaru. **Čím dříve dojde k odhalení chyby, tím méně finančních prostředků je potřeba vynaložit na její opravu.** Dopad důsledků nedostatečného testování lze rozdělit do čtyř následujících kategorií:

- zvýšený počet chyb,
- zvýšení ceny na vývoj,
- prodloužení času nasazení na trh,
- zvýšení nákladů na transakci.[1]

Nejzávažnějším problémem nedostatku testování je zvýšený počet výskytu defektů, které jsou objeveny až po nasazení na produkci. Při zpětném pohledu, proces identifikace a opravy chyb při vývoji softwaru představuje více než polovinu nákladů na vývoj. Včasné zjištění závad může výrazně snížit náklady. Nedostatek testovacích případů také prodlužuje dobu uvedení produktu na trh. Tato časová prodleva často vede ke ztrátě příležitosti na případný zisk.[1]

Jakýkoli kód, bez ohledu na senioritu vývojářů, bude obsahovat nějaké chyby. Některé chyby budou detekovány a odstraněny během jednotkového testování, některé během integračního a systémového testování. Nicméně vždy se některé dostanou až na produkční prostředí. Proto je důležité stanovit vhodnou úroveň testování softwaru. Stanovení vhodné úrovně testování softwaru je subjektivní proces. Ani nekonečné množství testovacích případů neprokáže, že software neobsahuje žádnou chybu. Na druhou stranu platí tvrzení, že více testovacích případů odhalí více potencionálních chyb.[1]

Dalším důvodem existence softwarových chyb je ekonomické hledisko. **I kdyby bylo technicky možné zbavit se všech chyb, zřídkakdy by to bylo ekonomicky výhodné**. Samotné testování spotřebovává zdroje a zároveň zvyšuje kvalitu produktu. Nicméně když se počet testů blíží k nekonečnu, k nekonečnu se blíží i náklady s testováním spojené. Každá firma si tak musí určit riziko a hranici, kdy je produkt už dostatečně dobrý na to, aby byl vpuštěn na trh. Správné určení této hranice je obzvláště zásadní na velice konkurenčních trzích, kde je extrémně důležité nabídnout produkt jako první.[1]

Při odhalení chyby je důležitým faktorem fáze vývoje, ve které je chyba identifikována. Když dojde k odhalení chyby na začátku procesu vývoje softwaru, oprava chyby bude méně nákladná, než když vývojáři chybu odhalí až v dalších fázích vývoje. Relativní ceny opravy chyb v závislosti na fázi vývoje jsou zobrazeny v tabulce 1.1.[1]

Tabulka 1.1: Příklad, jak může vypadat cena opravy chyby v závislosti na fázi vývoje. Proměnná x reprezentuje jednotku ceny.[1]

Návrh	Jednotkové t.	Integ. a syst. t.	Beta t.	Po nasazení
$1x$	$5x$	$10x$	$15x$	$30x$

1.2 Vývojáři a testeři

Vývojáři a testeři mají dvě různé, dalo by se říci protichůdné, role v procesu budování aplikace. **Hlavním úkolem vývojářů je návrh a následná implementace kódu, který splňuje všechny požadavky**, které jsou na něj kladeny. Jinými slovy, jejich hlavním cílem je, aby kód fungoval. Na druhou stranu, **úkolem testera je vzít stejné požadavky, vytvořený kód a ten následně rozbít**. Jejich práce je objevit trhlinu v kódu, kterou musí vývojáři následně opravit.[2]

Vzhledem k tomu, že vývojáři a testeři sledují jiné cíle, částečně se jedná o konfliktní stav. Z uvedených cílů také vyplývá, že **vývojáři jsou často špatnými testery**. Vývojáři se soustředí především na stav, kdy kód funguje. Méně už na to, aby ho rozbili testy. Vývojáři mají tendenci psát testy na základě typických scénářů. Mívají často příliš op-

timistický pohled na to, jak velkou část jejich kódu testy pokrývají. Při psaní testů pracují s předpokladem, že kód bude fungovat bezchybně. Koneckonců, jedná se o jejich vlastní kód.[2]

Z tohoto důvodu **má většina organizací pro vývoj softwaru samostatný testovací tým**, zejména pro integrační a systémové testování. Testeři píší své vlastní testy a případné chyby reportují zpět vývojářům, jejichž úkolem je chyby opravit. Výjimku tvoří jednotkové testy, které jsou zpravidla odpovědností vývojářů a píší si je sami.[2]

1.3 Chytré testování

I když mají softwarové firmy k dispozici vývojáře, kteří píší jednotkové testy a testovací tým, jehož jediným úkolem je identifikovat sebemenší chybu ještě před produkčním nasazením, stejně v drtivé většině případů **nedojde k odchycení všech chyb** a alespoň nějaká se dostane na produkci. Existují 2 hlavní důvody, proč tomu tak je.[2]

První příčinou je **lidská nedokonalost**. Nedokonalost hraje svou roli zejména u rozsáhlých projektů, které mají obrovský zdrojový kód. Není vždy v lidských silách přečíst a pochopit obrovské kusy kódu a odhalit všechny možné dynamické interakce mezi jednotlivými moduly.[2]

Druhým důvodem, proč chyby unikají z jedné testovací fáze do druhé a nakonec až k cílovému uživateli je fakt, že **software**, více než jakýkoli jiný produkt, který lidé vyrábějí, **je velmi složitý**. I malé programy mají mnoho průchodů kódem a mnoho různých typů chyb, ke kterým může dojít. Obrovský počet možných průchodů programem se nazývá **kombinatorická exploze**. S každým přidáním příkazu `if` do programu dojde ke zdvojnásobení počtu možných průchodů programem. To znamená, že u rozsáhlých programů není možné otestovat všechny možné průchody programem a všechny možné vstupní hodnoty.[2]

Pokud není možné otestovat všechny možné kombinace, je potřeba testovat chytře na základě plánu. Plánem je **identifikace nejpravděpodobnějších případů použití a jejich následné otestování**. Je nutné identifikovat nejpravděpodobnější vstupní hodnoty, hraniční podmínky a promítnout je do testovacích případů.[2]

Další otázka zní, v jaké fázi vývoje softwaru je nejefektivnější testy psát. V současnosti zde existují 2 hlavní proudy. **Tradičnějším přístupem je nejprve implementace funkcionality**, zkompilování kódu (aby došlo k vyloučení syntaktické chyby) a v okamžiku, kdy má vývojář pocit, že kód je hotový, napsání testů. Pro vývojáře je výhodné, že při samotné implementaci funkcionality dobře pochopil požadavky a měl možnost přemýšlet o konkrétních testovacích případech. Samotné testování a odhalování chyb probíhá současně a v případě, kdy vývojář odhalí chybu, opraví ji a okamžitě může znovu neúspěšné testy spustit.[2]

Novější přístup, který vychází z agilních metod vývoje softwaru, se nazývá **programování řízené testy** (test driven development). Vývojáři nejprve napíší jednotkové testy a až poté přistoupí k samotné implementaci požadavků. Při prvním spuštění samozřejmě všechny testy selžou. Po vytvoření testů je cílem implementovat požadavky tak, aby všechny testy prošly. Až tento okamžik nastane, vývojář ví, že jeho práce je u konce.[2]

Obecně platí, že při použití strategie programování řízené testy, programátor dřív přemýšlí o hraničních hodnotách, speciálních případech a dalších nástrahách, což využije při následné implementaci. Na druhou stranu, psaní testů bez existující implementace může být alespoň zprvu velice náročné. Vývojář si musí umět dobře představit výslednou implementaci dopředu. Programování řízené testy funguje dobře zejména pro menší a středně velké projekty.[2]

1.4 Testovací pyramida

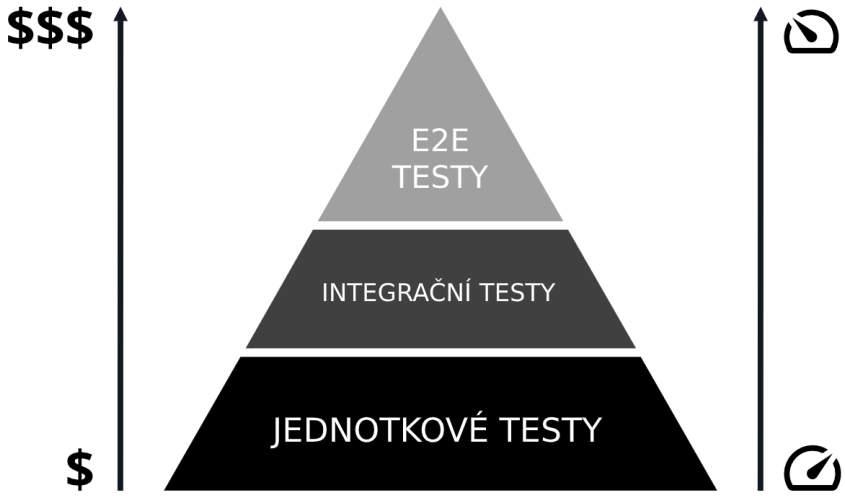
Testovací pyramida je metaforou, která nám říká, abychom seskupili softwarové **testy do segmentů s různou granularitou**. Poskytuje také představu o tom, kolik testů bychom měli mít v každé z těchto skupin. Ačkoli koncept testovací pyramidy už nějakou dobu existuje, týmy stále bojují se správným zavedením do praxe.[3]

Testovací pyramida v podstatě přichází se dvěma následujícími doporučeními:

- pište testy s různou granularitou,
- čím jsou testy komplexnější (testují více integrace), tím méně by takových testů mělo být.

Vývojáři by měli psát velké množství jednotkových testů, které testují dobře izolované jednotky a jsou velice rychlé a menší počet integračních testů, které už nejsou tak rychlé a pro jejich běh jsou potřeba další služby. End-to-end testy by měly pokrývat pouze klíčové aplikační části, na kterých stojí daná služba.

Opakem testovací pyramidy je **zmrzlinový kornout** (**ice cream cone**). Tento anti vzor preferuje velké množství komplexních testů a proti tomu malé množství izolovaných testů. Důsledkem následování tohoto vzoru je dlouhý běh testovacích sad a jejich obtížná udržitelnost.



Obrázek 1.1: Testovací pyramida[4]

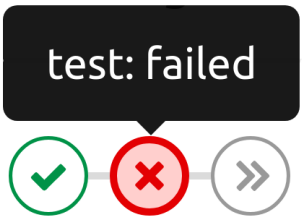
1.5 Testy součástí kontinuální integrace (CI)

Kontinuální integrace (CI) je způsob vývoje softwaru, kdy vývojáři často integrují svou práci, obvykle několikrát denně. **Pro každou integraci je spuštěn automatický build, který obsahuje i automatizované testy**. Cílem je objevit případné chyby co nejdříve. Mnoho týmů zjistilo, že tento přístup vede k významnému snížení počtu problémů souvisejících s integrací a umožňuje rychleji rozvíjet udržitelný software.[5]

Samotný princip integrace práce velice úzce souvisí s použitým verzovacím nástrojem. V dnešní době převládají distribuované verzovací nástroje (nejpoužívanější je Git). Moderní verzovací systémy pracují s větvemi. Zpravidla bývá jedna hlavní vývojová větev, jejíž kód je nahrán na produkčních serverech. Když vývojáři implementují novou funkcionalitu, založí si na základě hlavní větve novou větev, kde implementují funkcionalitu. Jakmile vývojáři dokončí svou práci ve vedlejší větvi, tuto větev integrují do větve hlavní a následně se tak jejich kód dostane na produkční servery.

V praxi to často vypadá tak, že při každém odeslání změn do repozitáře se spustí CI pipeline. Pipeline může obsahovat různé sekce, nicméně velice běžné jsou fáze sestavení, spuštění testů, vytvoření review domény a nahrání změn na produkční servery. Testovací fáze spustí automatizované testy a v případě, kdy alespoň jeden z testů skončil chybovým kódem, pipeline skončí a nepokračuje následující fází. Na obrázku 1.2 lze vidět 3 fáze. První fáze sestavení prošla, druhá fáze automatických testů neprošla a došlo ke grafickému znázornění a ukončení pipeline. Vývojáři, který odeslal dané revize, přijde notifikace, která informuje o tom, že CI pipeline neproběhla v pořádku. Vývojář

následně snadno zjistí, jaký test spadl a může provést okamžitou opravu. Kontinuální integrace pomáhá s odhalením chyb v raných fázích vývoje, což jak bylo zmíněno v kapitole 1.1 má i své ekonomické výhody.



Obrázek 1.2: Znázornění CI pipeline na platformě GitLab

1.6 FURPS

Metoda FURPS představuje model kvality softwaru, který definuje základní pravidla, jak ověřit kvalitu dodávaného softwaru. Model definuje základní charakteristiky, které by měly být brány na zřetel při vývoji softwaru. Jsou to:

- **F**uncionality — funkčnost. Zda aplikace splňuje vydefinované požadavky na funkčnost, které podporují byznysové procesy a zároveň bezpečnost.
- **U**sability — použitelnost. Zaměřuje se na konzistenci v uživatelském rozhraní, on-line a kontextovou nápovědu a celkově na intuitivní ovládání aplikace koncovým uživatelem.
- **R**eliability — spolehlivost. Zaměřuje se na frekvenci a závažnost selhání aplikace, obnovitelnost, přesnost a střední dobu mezi poruchami.
- **P**erformance — výkon. Definuje požadavky jako rychlost, dostupnost, doba odezvy, doba zotavení a využití zdrojů.
- **S**upportability — rozšiřitelnost a podporovatelnost. Aplikace musí být snadno rozšiřitelná, testovatelná, udržovatelná, konfigurovatelná nebo např. lokalizovatelná.[6]

Model kvality softwaru FURPS pochází již z roku 1987, kdy tento model poprvé představila společnost Hewlett-Packard. V současné době bývá často používán model FURPS+, který vznikl rozšířením původního modelu společností IBM a přidává modelu další charakteristiky jako jsou např.

- implementace,
- rozhraní,
- obchodní a právní hledisko.[6]

1.7 F.I.R.S.T

V knize [7] jsou popsány základní principy, které by měly testy splňovat. Pro snadnější zapamatování jednotlivých pravidel slouží zkratka F.I.R.S.T. Konkrétně se jedná o následující pravidla.

- **F**ast — rychlé. Testy by měly být rychlé a jednoduché. V případě, kdy testy poběží dlouho, budou spouštěny méně často.
- **I**ndependent — nezávislé. Testy by na sobě neměly záviset. Zejména by testy neměly vytvářet a upravovat data, na kterých závisí ostatní testy.
- **R**epeatable — opakovatelné. Testy by měly být kdykoli spustitelné v libovolném pořadí.
- **S**elf-Validating — ověřitelné. Každý test by měl buďto projít nebo neprojít. Ověření by mělo být jednoduché.
- **T**imely — včasné. Testy by měly být psány tak, aby v okamžiku, kdy jsou zapotřebí, byly dostupné. Např. pro strategii programování řízené testy to znamená napsání testů před samotnou implementací.

2 Druhy testů a programovací konvence

V typickém projektu vývoje softwaru existují tři úrovně testování: **jednotkové testování**, **integrační testování** a **systémové testování**.[2] Pojem systémové testování je v řadě zdrojů zaměněn za termín end-to-end testování.

Jednotkové testování obvykle provádí vývojáři a dochází k testování jednotlivých metod a tříd, nicméně nedochází ke komplexnějšímu testování větších částí programu. Obvykle také nedochází k testování knihoven třetích stran. Při jednotkovém testování autoři testů ví, jak mají vypadat vstupní parametry metod, jaké mají mít typy, jaké jsou návratové hodnoty a jaká je vnitřní struktura kódu. Takový případ nazýváme **white-box testováním**.[2]

Integrační testování obvykle provádí oddělené týmy testerů, nikoli samotní vývojáři. Na této úrovni jsou testování podrobeny kolekce tříd nebo moduly, které spolu vzájemně interagují. Testeři píší testy s vědomím toho, jak přesně vypadá rozhraní dané třídy, ale nemají informace o konkrétní implementaci a vnitřní struktuře kódu. Z toho důvodu spadají do kategorie **grey-box testování**. Tyto testy se provádí zpravidla poté, co dochází k integraci již otestovaného kódu jednotkovými testy do zbytku aplikace.[2]

Systémové testování obvykle provádí samostatné týmy testerů a cílem je podrobit testování celý systém. Testovací tým provádí testy na základě požadavků na daný systém, aniž by věděl, jak je daný systém navržen nebo napsán. Takový přístup nazýváme **black-box testováním**. Cílem testerů na této úrovni je ověření faktu, že systém splňuje všechny předepsané požadavky. Black-box testování může zahrnovat také zátěžové testování, testování použitelnosti nebo akceptační testování. Na této úrovni mohou být do testování zapojeni i koncoví uživatelé.[2]

2.1 Jednotkové testování

Jednotkové testy se provádí na jednotlivých modulech zdrojového kódu. Vývojáři provádí jednotkové testy, aby se ujistili, že komponenty, které naprogramovali, fungují správně.

V objektově orientovaném přístupu je většinou nejmenší jednotkou pro testování myšlena třída. Jelikož jednotky se následně integrují v komponenty, vývojáři tak dostanou **reálný obraz toho, zda výsledný produkt může být funkční**. Jednotkové testy

zpravidla píší vývojáři, nikoli testeři.[8] Příklad jednotkového testu lze vidět v ukázce kódu 2.1.

```php
<?php
use PHPUnit\Framework\TestCase;

final class EmailTest extends TestCase
{
    public function testCanBeCreatedFromValidEmailAddress(): void
    {
        // ověření, zda metoda vrátí objekt požadované instance
        $this->assertInstanceOf(
            Email::class,
            Email::fromString('user@example.com')
        );
    }

    public function testCannotBeCreatedFromInvalidEmail(): void
    {
        // ověření, zda metoda vyhodí výjimku pro nevalidní e-mail
        $this->expectException(InvalidArgumentException::class);

        Email::fromString('invalid');
    }

    public function testCanBeUsedAsString(): void
    {
        // ověření, zda objekt lze použít jako řetězec
        $this->assertEquals(
            'user@example.com',
            Email::fromString('user@example.com')
        );
    }
}
```

Ukázka kódu 2.1: Jednotkový test metody `fromString` ze třídy `Email`[10]

V knize [9] jsou popsány základní strategie efektivního a robustního vytváření jednotkových testů. Jedná se o strategie:

- testování hraničních hodnot,
- testování ekvivalentních tříd,

- testování na základě rozhodovací tabulky,
- testování toku dat.

2.1.1 Testování hraničních hodnot

Strategie testování hraničních hodnot se zaměřuje na vstupní parametry testované funkce nebo metody. Na základě vstupních parametrů jsou určeny hodnoty, které budou následně podrobeny testování.

Hlavní myšlenka, která stojí za testováním hraničních hodnot je taková, že **k chybám často dochází poblíž hraničních (maximálních nebo minimálních) vstupních hodnot**. V praxi se tak vezme minimální hraniční hodnota, hodnota o jedna menší a hodnota o jedna větší. Stejně tak se postupuje u maxima. Identifikuje se maximální hraniční hodnota a ta se testuje, stejně jako hodnota o jedna menší a o jedna vetší.

Testování hraničních hodnot vyžaduje, aby bylo možné hraniční hodnoty mezi sebou uspořádat. Tzn. pro každý pár vstupních hodnot $\langle a, b \rangle$ je možné určit, zda $a \leq b$ nebo $b \geq a$.

Např. existuje-li metoda, která přijímá dva celočíselné parametry a a b, kde a reprezentuje týden v roce a b reprezentuje den v týdnu. Pro každý parametr je nutné určit hraniční hodnoty. Pro parametr a bude minimální hodnotou 1 a maximální hodnotou 53 (pozemský rok nemůže mít víc než 53 týdnů). Minimální hodnotou parametru b bude 0 (reprezentuje pondělí) a maximální 6 (reprezentuje neděli). Hraniční hodnoty jsou znázorněny v tabulce 2.1.

Tabulka 2.1: Hraniční hodnoty parametrů

	parametr a	**parametr** b
min - 1	0	-1
min	1	0
min + 1	2	1
max - 1	52	5
max	53	6
max + 1	54	7

2.1.2 Testování ekvivalentních tříd

Cílem metody testování ekvivalentních tříd je maximální redukce testovacích případů a zároveň otestování všech možných průchodů testovaným kódem. Vstupní parametry

se rozdělí do ekvivalentních tříd na základě výstupu testovaného kódu. Tyto skupiny jsou poté považovány za ekvivalentní a test je proveden pouze na jednom reprezentantovi ekvivalentní třídy. Premisou je fakt, že kód by měl vždy vrátit stejný výstup pro libovolného reprezentanta ekvivalentní třídy.

Tuto strategii lze demonstrovat na příkladu funkce, která má vstupní parametr a, který reprezentuje věk. Na základě věku funkce určí, zda si daný člověk může koupit alkohol v České republice. Parametr a by z principu neměl být menší než 0 a zároveň by neměl být větší než např. 130. Když je parametr a v daných mantinelech, funkce kontroluje, zda je a alespoň 18 (minimální věk nutný k nakoupení alkoholu v České republice). Ekvivalentní třídy pro tento konkrétní příklad jsou znázorněny v tabulce 2.2.

Tabulka 2.2: Ekvivalentní třídy pro parametr a

	parametr a
třída A	$(-\infty, 0)$
třída B	$\langle 0, 18)$
třída C	$\langle 18, 130\rangle$
třída D	$(130, \infty)$

2.1.3 Testování na základě rozhodovací tabulky

Rozhodovací tabulky slouží k identifikaci konkrétních testovacích případů tak, aby došlo k otestování všech možných kombinací vstupních parametrů. Tato testovací strategie má silný logický základ a oporu v řadě vědeckých odvětví, jako je např. teorie her.

Rozhodovací tabulka obsahuje popis podmínek vstupních parametrů, hodnotu pravda/nepravda, zda jsou podmínky splněny a očekávaný výstup. Po vytvoření takové tabulky jsou vybrány konkrétní hodnoty, které splňují zadané podmínky a tyto hodnoty jsou použity pro konkrétní testovací případy spolu s očekávánou výstupní hodnotou.

Pro nastínění použití rozhodovací tabulky bude použit příklad ze sekce 2.1.1, kde parametr a reprezentuje týden v roce a parametr b den v týdnu. Pro tento případ bude mít rozhodovací tabulka následující podobu 2.3. Rozhodovací tabulka s konkrétními hodnotami je zobrazena v tabulce 2.4.

Tabulka 2.3: Rozhodovací tabulka

	validní	**validní**	**validní**	**validní**
parametr *a*	nepravda	pravda	nepravda	pravda
parametr *b*	nepravda	nepravda	pravda	pravda
očekávaný výstup	Nevalidní *a* i *b*	Nevalidní *b*	Nevalidní *a*	OK

Tabulka 2.4: Rozhodovací tabulka s hodnotami

parametr *a*	-10	30	60	40
parametr *b*	-1	8	2	4
očekávaný výstup	Nevalidní *a* i *b*	Nevalidní *b*	Nevalidní *a*	OK

2.1.4 Testování toku dat

Testování toku dat je strategie navržena k ověření interakce mezi definicí proměnné a jejím použitím. Tato strategie identifikuje anomálie v kódu, které jsou nelogické, často signalizují výskyt chyby a v kódu by se vůbec neměly vyskytovat. Příklady takových anomálií:

- použití proměnné, která ještě nebyla definována,
- odstranění proměnné, která ještě nebyla definována,
- odstranění proměnné, která nebyla nikdy použita,
- odstranění proměnné, která už byla odstraněna,
- použití proměnné, která už byla odstraněna,
- definice proměnné, která není následně nikde použita.[11]

2.2 Integrační testování

Integrační testování ověřuje správnou komunikaci mezi jednotlivými jednotkami kódu. Hlavním důvodem je fakt, že i když je každý model poctivě otestován jednotkovými testy, automaticky to neznamená, že tyto komponenty spolu budou bezchybně komunikovat a budou spolu perfektně kompatibilní. Přesně takovým potencionálním chybám se snaží vývojáři a testeři vyhnout za pomoci integračního testování. Příklad integračního testu lze vidět v ukázce 2.2.

```php
<?php
use PHPUnit\Framework\TestCase;

final class OrderFactoryTest extends TestCase
{
    public function testAddProductToOrder(): void
    {
        // vytvoření testovacího košíku
        $basket = $this->basketDataProvider->createTestBasket();
        // přidání prvního produktu do košíku
        $basket->addProduct(new BasketProduct('Produkt 1'));
        // přidání druhého produktu do košíku
        $basket->addProduct(new BasketProduct('Produkt 2'));
        // vytvoření objednávky
        $order = $this->orderFactory->createOrderForBasket($basket);
        // ověření, zda objednávka obsahuje přesně 2 produkty
        $this->assertSame(2, $order->getProductCount());
        // ověření, zda objednávka obsahuje první produkt
        $this->assertTrue($order->hasProduct('Produkt 1'));
        // ověření, zda objednávka obsahuje druhý produkt
        $this->assertTrue($order->hasProduct('Produkt 2'));
    }
}
```

Ukázka kódu 2.2: Integrační test vytvoření objednávky pomocí třídy `OrderFactory`

Dle [9] existuje řada strategií, jak provádět integrační testování:

- shora dolů,
- zdola nahoru,
- sendvičová (smíšená) strategie.

2.2.1 Shora dolů

Při implementaci integračního testování shora dolů dochází **nejprve k implementaci samotného jádra systému**. Všechny nižší jednotky, které ještě nemusí být implementovány, jsou pro účely testování nahrazeny tzv. **stuby**. Stuby jsou jednoduché funkční celky, které zjednodušeně simulují volání jednotlivých jednotek, které vstupují do integrace. Výhodou je, že po implementaci stubů vznikne použitelný prototyp pro předvedení.

2.2.2 Zdola nahoru

Integrace zdola nahoru je zrcadlovým opakem zmíněné strategie shora dolů s tím rozdílem, že stuby jsou nahrazeny **drivery**. Drivery jsou moduly, které simulují chování jednotek na vyšší úrovni. **Testování tak začíná u nejspodnějších jednotek**, které jsou volány za pomocí driverů.

2.2.3 Sendvičová strategie

Sendvičová strategie kombinuje oba dva výše zmíněné přístupy (shora dolů, zdola nahoru) a využívá principu stubů a driverů. Jak samotný název napovídá, sendvičová strategie identifikuje 3 různé vrstvy:

- cílová vrstva,
- vrstva nad cílovou vrstvou,
- vrstva pod cílovou vrstvou.

V sendvičovém testování je testování zaměřeno především na cílovou vrstvu. Toto testování je vybráno na základě charakteristik systému a struktury kódu. Strategie se snaží minimalizovat počet stubů a driverů. Uživatelská rozhraní jsou testována izolovaně pomocí stubů. Funkce na nejnižší úrovni jsou testovány pomocí driverů.

Sendvičová strategie se hodí především pro rozsáhlé projekty, které se skládají z dílčích projektů. Sendvičové testování poskytuje větší pokrytí testy při stejném počtu stubů a umožňuje paralelní testování.

2.3 End-to-end testování

End-to-end (E2E) testování odkazuje na metodu testování softwaru, která zahrnuje testování aplikace od začátku do konce. Tato metoda si v zásadě klade za cíl replikovat scénáře chování skutečných uživatelů. Cílem je otestování komunikace s hardwarem, síťovým připojením, externími závislostmi, databázemi a dalšími aplikacemi.[12]

Tato sekce se věnuje zejména **automatizovanému end-to-end testování**, nicméně end-to-end testování lze provádět i **manuálně**. Při manuálním testování tester představuje roli zákazníka (uživatele) a provádí kroky podle předem zadaných scénářů s cílem zjistit, zda lze danou cestu projít bez chyb s požadovaným výsledkem. Manuální testování může být při rapidním vývoji softwaru velice nákladné.

Při implementaci end-to-end testování je třeba také brát v potaz **vysokou náročnost tohoto typu testů na zdroje a čas** (např. v porovnání s jednotkovými testy). Je potřeba pečlivě vybírat testovací scénáře. V praxi se týmy často uchylují k pokrytí pouze svých

klíčových aplikačních částí (např. přihlášení do systému, vytvoření objednávky apod.), aby množství testů zhruba odpovídalo schématu testovací pyramidy, jak je uvedeno v kapitole 1.1. Příklad konkrétního testu lze vidět v ukázce 2.3.

```
<?php
use PHPUnit\Framework\TestCase;

final class SearchTest extends TestCase
{
    public function testSearch(): void
    {
        // získání elementu uživatelského vstupu pro vyhledávání
        $s = $this->webDriver->findElement(WebDriverBy::id('id'));
        // kliknutí na element
        $s->click();
        // zadání hledané fráze do uživatelského vstupu
        $this->webDriver->getKeyboard()->sendKeys('fráze');
        // odeslání formuláře pomoci enteru
        $this->webDriver->getKeyboard()->pressKey(WebDriverKeys::ENTER);
        // získání elementu nadpisu po odeslání formuláře
        $h = $this->webDriver->findElem(WebDriverBy::tagName('h1'));
        // otestování, zda nadpis obsahuje hledanou frázi
        $this->assertStringContainsString('fráze', $h->getText());
    }
}
```

Ukázka kódu 2.3: Příklad end-to-end testování vyhledávání na webu

2.3.1 Objektový model stránky (Page object model)

End-to-end testování má mimo zjevných benefitů i svá úskalí. V případě ignorování těchto nástrah a nedodržování základních zásad end-to-end testování se testy mohou stát těžko udržitelné při rapidním vývoji anebo může být jejich udržování velice nákladné. Jak je popsáno v práci [13], dva základní problémy end-to-end testování jsou **křehký testovací kód** (fragile test code) a **problém silné provázanosti a nízké soudržnosti** (high coupling and low cohesion).

Hlavním problémem je udržování testovacího kódu při souběžném vývoji testované aplikace. Při psaní automatických end-to-end testů se používají tzv. **lokátory**. Lokátory jsou specifické příkazy používané nástroji pro automatizaci testů k identifikaci webových prvků v grafickém uživatelském rozhraní.

Příklad lokátoru pomocí jazyku XPath, který odkazuje na druhý uživatelský vstup pro e-mailovou adresu na stránce.

```
(//input[@type="email"])[2]
```

V případě, kdy nastane změna na dané stránce a dojde např. k přidání dalšího uživatelského vstupu typu e-mail, může dojít k rozbití automatických testů, protože lokátor již nebude odkazovat na požadovaný element. **Takový případ nazýváme problém křehkého testovacího kódu**.

Když testovací kód obsahuje na tom samém místě testovací scénář a zároveň implementační detaily jako je např. získání elementu na základě lokátoru, **dochází k tzv. problému silné provázanosti a nízké soudržnosti**. V takovém případě většinou dochází k častému opakování jednotlivých lokátorů a zdůrazňuje se tak problém křehkého testovacího kódu.

Řešením obou výše zmíněných problémů je návrhový vzor **Page object model**. Martin Fowler popisuje Page object model takto: *„Když píšete testy pro webovou aplikaci, musíte se odkazovat na jednotlivé elementy stránky. Nicméně, pokud se budete odkazovat přímo na HTML elementy, vaše testy budou křehké vůči změnám. Page objekt poskytne přístup k elementům vaší stránky pomocí API, aniž byste se museli zabývat HTML elementy.“*[14]

Fowler tedy jinými slovy nabádá k tomu, aby pro každou stránku existovalo API, které poskytuje přístup k požadovaným vlastním elementům. Mmj. dojde k tomu, že lokátory budou použity na úrovni implementace API a nedojde tak k míchání testovacího scénáře a implementačních detailů. Zároveň díky API mohou být elementy použity ve více testech pomocí jednoho kódu. Testovací kód se tak zbaví své křehkosti a stane se robustním vůči aplikačním změnám.

2.4 Výkonnostní testování

Výkonnost aplikace je nezbytnou součástí při snaze zajistit koncovému uživateli přívětivý uživatelský zážitek a výkonnostní testování hraje klíčovou roli při jejím dlouhodobém zajišťování. Výkonnostní testování se neomezuje pouze na webové aplikace, kde se pozornost soustředí především na koncového uživatele. Využívá se i pro jiné architektury jako je klasický klient-server, distribuovaná nebo vestavěná. Dle ISO 25010 je výkonnost (efektivita výkonu) kategorizována jako nefunkcionální charakteristika kvality se třemi dalšími podkategoriemi.

- Chování v čase — obecně nejběžnější cíl výkonnostního testování. Zkoumá schopnost komponenty nebo systému reagovat na uživatelské nebo systémové vstupy v rámci stanoveného času a za stanovených podmínek.

- Práce se zdroji — zabývá se zkoumáním efektivity využívání zdrojů jako je např. alokace paměti RAM.
- Kapacita — zkoumá vhodnost architektury v závislosti na jejich kapacitách (např. počet uživatelů nebo množství dat).

Výkonnostní testování má často formu experimentu, který umožňuje provést měření a analýzu konkrétních parametrů systému. Tato měření často bývají prováděna opakovaně a umožňují přijímat architektonická rozhodnutí, která podporují jednotlivé vybrané aspekty testovaného systému.[15]

2.4.1 Typy výkonnostního testování

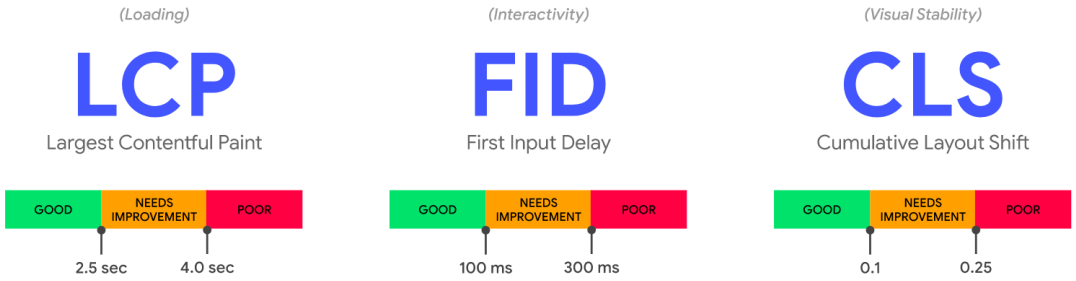
Existují různé typy výkonnostního testování. Každý z nich může být použitelný pro určitý projekt v závislosti na cílech testu. Dle [15] existují následující typy.

- Zátěžový test (load testing). Zaměřuje se na schopnost systému zvládat rostoucí zátěž generovanou kontrolovaným počtem souběžných uživatelů.
- Test hraniční zátěže (stress testing). Zaměřuje na schopnost systému zvládat enormní zatížení, které je na hranici nebo nad specifikovanou hranicí systému.
- Test škálovatelnosti (scalability testing). Zaměřuje se na schopnost systému splnit budoucí požadavky, které mohou být nad rámec aktuálně požadovaných. Cílem je určit schopnost systému růst (např. větší počet uživatelů, větší počet uložených dat) bez selhání.
- Test špiček (spike testing). Zaměřuje se na schopnost systému správně reagovat na náhlé extrémní zatížení a poté se vrátit do ustáleného stavu.
- Test vytrvalosti (endurance testing). Zaměřuje se na stabilitu systému v rámci delšího časového horizontu. Tento test ověřuje, že neexistují žádné problémy se zdroji (únik paměti, spojení s databází, ...), které mohou snížit výkon aplikace nebo vést k jejímu selhání.
- Test souběžnosti (concurrency testing). Zaměřuje se na dopad akcí, které se dějí současně (např. když se současně přihlásí velký počet uživatelů).

2.4.2 Web Vitals

V roce 2020 vydala společnost Google sadu metrik — Web Vitals[16], které jsou pro ni důležité při posuzování kvality uživatelského zážitku na webu a úzce souvisí s výkonností. Takových metrik je celá řada, nicméně Google vypíchl hlavní 3 metriky, které

označuje jako klíčové — Core Web Vitals. Jedná se o metriky LCP, FID, CLS. Google hodnotí zmíněné metriky třemi stupni. Jejich mezní hodnoty lze vyčíst na obrázku 2.1.



Obrázek 2.1: Metriky Core Web Vitals[16]

- good — daná metrika je splněna,
- needs improvement — metrika je zčásti splněna, ale potřebuje vylepšení,
- poor — metrika není splněna.

LCP (largest contentful paint — vykreslení největšího obsahu) měří dobu **vykreslení největšího obsahového prvku viditelného ve viewportu**. Pro zajištění dobrého uživatelského zážitku by se LCP měl objevit do **2,5 sekundy od prvního spuštění načítání stránky**.

FID (first input delay — prodleva prvního vstupu) sleduje dobu první interakce uživatele s webem. Pro zajištění dobrého uživatelského dojmu by weby měly mít **FID menší než 100 milisekund**.

CLS (cumulative layout shift — kumulativní změna rozvržení) popisuje, jak moc se při načítání mění rozvržení stránky a nabývá hodnot mezi 0 a 1. Pro zajištění dobrého uživatelského dojmu by weby měly udržovat **CLS menší než 0,1**.

2.5 Jednotný coding standard

Jednotný coding standard je sada pravidel definovaných pro konkrétní projekt. Tato pravidla předepisují způsob, jak má a nemá kód vypadat. Jednotný coding standard zpravidla předepisuje způsob odsazení kódu, maximální cyklomatickou složitost, maximální délku řádku nebo způsob psaní víceslovných frází (např. velbloudí notace — camel case nebo hadí notace — snake case).

Hlavními důvody, proč týmy používají jednotný coding standard jsou:

- jednotný vzhled kódu, který píší různí vývojáři,

- zlepšení čitelnosti, udržitelnosti a snížení složitosti kódu,
- zlepšení znovupoužitelnosti kódu a detekce chyb,
- podpora správných programátorských přístupů a zvýšení efektivity.[17]

Kontrola jednotného coding standardu bývá zpravidla součásti kontinuální integrace. V případě, kdy dojde k porušení předepsaných pravidel, CI pipeline buďto spadne, anebo dojde k použití nástroje na automatickou opravu kódu. Takový nástroj zpravidla nedokáže vyřešit všechny chyby, nicméně může např. automaticky změnit odsazení tak, aby vyhovovalo předepsaným pravidlům.

2.6 Statická analýza kódu

Statická analýza kódu je proces analýzy zdrojového kódu programu, jehož cílem je **hledání chyb a nedostatků bez jeho faktického spuštění**. Nástroje statické analýzy pomáhají vývojářům identifikovat slabiny a nedostatky, které by mohly ohrozit bezpečnost a integritu programu.[18]

Většina pokročilých statických analyzátorů staví svou analýzu nad abstraktním syntaktickým stromem (AST). Jednou z výhod statické analýzy kódu je **rychlost provedení a relativně nízká náročnost na prostředky**. Tato výhoda plyne především z faktu, že analyzovaný kód není reálně spouštěn. V důsledku toho se statická analýza často používá součástí kontinuální integrace.

Statická analýza umí odhalit řadu potencionálních problémů jako např.:

- nekompatibilní typy,
- detekce nedosažitelného kódu,
- použití nedefinovaných proměnných,
- volání neexistujících funkcí.

Výše zmíněné benefity lze využít hlavně v dynamicky typovaných programovacích jazycích, které provádí dynamickou typovou kontrolu až za běhu programu. U silně typovaných programovacích jazyků může být řada těchto problémů odhalena už při samotné kompilaci.

2.6.1 Statická analýza a code review

Studie [19] se zabývala otázkou, jak může statická analýza pomoci zredukovat čas strávený programátory na manuálním code review. Zkoumáno bylo 23 open source Java

repozitářů v rámci kterých bylo analyzováno 96 pull requestů (žádostí o spojení kódu do hlavní vývojové větve).

Výsledkem studie je zjištění, že statická analýza zpětně odhalila 16 % všech připomínek, které se objevily v rámci pull requestů. Toto zjištění tak podporuje v praxi často zaběhlé řešení, kdy dochází k vyhodnocení statické analýzy kódu v rámci kontinuální integrace ještě před vytvořením samotného pull requestu. Vývojář tak může opravit nedostatky vyplývající ze statické analýzy a tím mmj. dochází k šetření času ostatních vývojářů stráveného na code review.

2.7 Mutační testování

Mutační testování ověřuje kvalitu ostatních typů testů, jako jsou například testy jednotkové nebo integrační. Poskytují cenné informace o tom, zda ostatní testy jsou napsané vhodně a efektivně a odhaluje potencionální slabiny ostatních testů. Jak z textu vyplývá, nutnou podmínkou pro spuštění mutačních testů je implementace ostatních typů testů.

Mutační testování se týká procesu automatické mutace syntaxe programu s cílem vytvořit varianty programu, tj. generovat umělé defekty na základě předem nadefinovaných pravidel. Tyto programy s defekty se nazývají **mutanti**.

V kontextu testování vývojáře nejvíce zajímá stav mutantů. Testovací případy, které jsou schopné odlišit chování mutantních programů od chování původního nezměněného programu, splňují cíle testu. V takovém případě je **„mutant zabit — killed mutant“**. V opačném případě, když testovací případ projde, říkáme, že **„mutant unikl — escaped mutant“**.[21] Mutanti jsou generováni na základě předem definovaných mutačních operátorů. Jako příklad mutačního operátoru lze uvést záměnu binárního operátoru plus za minus nebo záměnu znaménka násobení za dělení. Více k mutačním operátorům v sekci 2.7.1.

Při spuštění mutační analýzy dojde k parsování definovaných částí kódu aplikace a vytvoření mutantů na základě mutačních operátorů. Když jsou mutanti vytvořeni, dojde ke spuštění již existujících testů a dle poměru zabitých mutantů k celkovému počtu mutantů dojde k výpočtu skóre **MSI** (Mutation score indicator).

MSI = počet zabitých mutantů / celkový počet mutantů * 100

MSI je hlavním výstupem mutačního testování. Čím je toto číslo větší, tím kvalitněji jsou ostatní testy napsané a tím méně by měl být kód náchylný k chybám (tím více chyb a nedostatků by mělo být odhaleno procesem automatického testování). V praxi se často toto číslo používá jako podmínka při odeslání revizí do vzdáleného repozitáře v rámci kontinuální integrace. Když je číslo menší než stanovená mez, vývojář musí své testy opravit.

2.7.1 Mutační operátory

Na základě mutačních operátorů dochází ke generování mutantů (vkládání chyb do kódu). Většina nástrojů pracujících s mutačními testy nabízí širokou paletu mutačních operátorů. Mutační operátory lze rozdělit na několik typů. Např.

- binární aritmetika (plus za minus, modulo za násobení, or za xor, ...),
- viditelnost funkcí (`protected` za `public`, `private` za `protected`, ...),
- zaokrouhlování (zaokrouhlení za zaokrouhlení dolů, ...),
- práce s výjimkami (odebrání bloku `finally`, záměna vyhozené výjimky, ...),
- nahrazení literálů (0 za 1, přičtení jedničky k číslu, ...).

2.7.2 Aproximační strategie

Mutační testování má všeobecně dobrou pověst jedné z **nejúčinnějších, ale zároveň nejdražších** (nejnáročnějších na zdroje) softwarových testovacích technik. Za účelem snížení nákladů na aplikaci mutace, různí vědci navrhli použití aproximačních strategií. Tyto strategie slibují snížení nákladů na testování aplikace při zachování účinnosti mutačního testování.[22]

Aproximační strategie se spoléhají hlavně na skutečnost, že většina produkovaných mutantů je nadbytečná v tom smyslu, že jsou „téměř“ vždy zabiti, když dojde k zabití jiných ekvivalentních mutantů. **Tyto strategie se snaží heuristicky produkovat mutanty, kteří nejsou redundantní**. Takové základní strategie jsou běžně označovány jako **náhodný výběr** a **selektivní mutace**. Strategie náhodného výběru náhodně použije pouze konkrétní procento z celé sady zavedených mutantů. Selektivní strategie se snaží použít pouze podmnožinu celku sady mutantních operátorů.[22]

Pokročilejší aproximační strategie jsou popsány ve studii [22]. Konkrétně se jedná o strategie:

- strategie testování mutací prvního řádu,
- strategie testování mutací druhého řádu.

Strategie testování mutací prvního řádu je obdobou strategie náhodného výběru. Strategie se opírá o empirickou studii [23], kde autor došel k závěru, že použití pouze 10 % vygenerovaných mutantů vyústí ke ztrátě detekce chyb pouze o 16 %. Použití této strategie tedy velmi významně sníží náklady na mutační testování.

Strategie testování mutací druhého řádu vychází z výše zmíněné strategie testování mutací prvního řádu a snaží se dokonce ještě o další snížení ceny mutačního testování.

Tato strategie pracuje již s omezenou množinou mutantů ze strategie prvního řádu a z této množiny tvoří dvojice, které jsou sloučeny do jednoho mutantního programu. Podmínkou je, že každý původní mutant musí být obsažen alespoň v jednom sloučeném mutantním programu. Každý nový mutantní program obsahuje tedy přesně 2 mutace. Výsledkem aplikace této aproximační strategie dojde ke snížení mutantů téměř na polovinu.

V závěru studie [22] prezentuje zjištěné poznatky:

- obě zmíněné strategie významně snižují cenu testování,
- první strategie je účinnější v detekci chyb než druhá strategie,
- druhá strategie snižuje výrazně cenu testování i v porovnání s první strategií,
- druhá strategie odstraní přibližně 80 % až 90 % ekvivalentních mutantů.

2.7.3 Předpoklad bezchybného programu

Řada vývojových týmů se spoléhá na různé indikátory kvality automatických testů. Většinou se jedná o hodnotu, která reprezentuje **procento pokrytí kódu** automatickými testy. Takových indikátorů je celá řada, mezi nejpopulárnější patří:

- pokrytí příkazů (statement coverage) testy,
- pokrytí větví (branch coverage) testy,
- pokrytí slabých mutací (weak mutation coverage) testy,
- pokrytí silných mutací (strong mutation coverage) testy.

Pokrytí příkazů a pokrytí větví jsou indikátory toho, kolik procent všech příkazů resp. větví kódu je pokryto testy. Pokrytí slabých mutací nastává tehdy, když se stav výpočtu po provedení mutantního kódu liší od původního kódu. Pokrytí silných mutací je stav, kdy původní kód a kód s mutací vykazují odlišnost ve svých výstupech.

Ve studii [24] se často pracuje s pojmem **předpoklad bezchybného kódu**. Je to předpoklad, že testovací sady jsou hodnoceny na základě pokrytí, kterého dosahují na bezchybných programech, které neobsahují žádné známé chyby.

Statement coverage a branch coverage jsou metriky, které uplatňují předpoklad bezchybného programu, narozdíl od mutačních metrik.

Výsledkem studie [24] je tvrzení, že **zvyšování statement coverage, branch coverage a weak mutation coverage má malý vliv na odhalení nových chyb**. Jedním z dalších zajímavých (a možná překvapivých) zjištění zmíněné studie je, že ve vztahu

mezi strong mutation coverage a odhalením chyb figuruje určitá mezní hodnota. To znamená, že nad určitou hranicí byla zpozorována silná **korelace mezi zvýšením pokrytí kódu a zvýšeným počtem odhalených chyb**. Pod touto prahovou hranicí je však pokrytí dosažené testovací sadou irelevantní a nijak významně nezvyšuje počet odhalených chyb.

2.8 Smoke a sanity testování

Účelem smoke (kouřového) testování je zjistit, zda je nový build softwaru stabilní, aby bylo možné build podrobit dalšímu podrobnějšímu testování. Na stabilním buildu budou dále provedeny další typy testů, které daná aplikace obsahuje. Pokud však build není stabilní, tj. smoke testy selžou, pak je sestavení odmítnuto a vývojový tým musí chyby opravit a vytvořit nový stabilní build.[25]

Smoke testy testují pouze nejzákladnější a klíčovou funkcionalitu aplikace. Neusiluje o hluboké detailní testování aplikace a hlavním cílem je co nejvčasnější detekce nestabilních buildů a jejich vrácení vývojovému týmu k opravě. Ve většině případů bývají smoke testy prováděny automaticky po sestavení programu.

Sanity testování je termín, který souvisí se smoke testy, nicméně nejsou totéž. Základní charakteristika, která oba typy testů spojuje je, že oba typy jsou používány jako kritéria pro přijetí nebo odmítnutí nové verze buildu. **Pokud testy selhávají, pak je build odmítnut a další testy se nespouštějí**. Smoke testy jsou součástí regresního testování, zatímco sanity testy jsou součástí akceptačního testování. **Kontrolují, zda nově přidaná funkce funguje správně a nezpůsobuje žádné chyby**. Obecně se smoke testy provádí na relativně nestabilním produktu, zatímco sanity testy se provádí na relativně stabilním produktu.[25]

3 Testovací nástroje

V následujících sekcích jsou popsány nástroje, které byly využity při praktické části diplomové práce. Jedná se výhradně o nástroje s otevřeným zdrojovým kódem, které jsou zdarma pro komerční využití.

3.1 PHPUnit

PHPUnit je velice rozšířený framework pro psaní testovacích scénářů v programovacím jazyce PHP. Samotný nástroj je postavený na architektuře xUnit. Frameworky implementující architekturu xUnit sdílí řadu základních komponent, které implementují. **PHPUnit poskytuje základní funkcionalitu pro psaní jednotkových nebo integračních testů.** Poskytuje bohatý výčet připravených assertů (předpokladů), které mohou vývojáři využívat. PHPUnit umožňuje testovat integraci komponent, které není možné nebo efektivní použít v testovacím prostředí. Framework pro podobné případy nabízí řešení pomocí mocků a stubů.

3.1.1 Vytvoření testu

Dle dokumentace [26], vytvoření samotného testu vyžaduje 4 jednoduché kroky. Příklad testu lze vidět na ukázce 3.1.

1. Vytvoření třídy obsahující testy. Třída s testy se zpravidla jmenuje stejně jako testovaná třída, pouze s příponou `Test`.
2. Třída s testy dědí od třídy `PHPUnit\Framework\TestCase`.
3. Samotné testovací scénáře tvoří veřejné metody s předponou `test`.
4. Uvnitř testovacích metod jsou uvedeny asserty, které ověřují, že se testovaný kód chová dle předpokladů.

Jak bylo uvedeno výše, samotný framework poskytuje celou řadu připravených assertů. Samotné asserty ověřují, zda se testovaný kód chová dle očekávání. PHPUnit nabízí např. následující asserty:

```php
<?php
use PHPUnit\Framework\TestCase;

final class StackTest extends TestCase
{
    public function testPushAndPop(): void
    {
        $stack = [];
        $this->assertSame(0, count($stack));

        array_push($stack, 'foo');
        $this->assertSame('foo', $stack[count($stack)-1]);
        $this->assertSame(1, count($stack));

        $this->assertSame('foo', array_pop($stack));
        $this->assertSame(0, count($stack));
    }
}
```

Ukázka kódu 3.1: Testovací scénář za pomoci nástroje PHPUnit[26]

- `assertTrue($hodnota)` — ověřuje, zda proměnná `$hodnota` je pravda,
- `assertSame($očekávání, $hodnota)` — ověřuje, zda proměnná `$hodnota` nabývá stejné hodnoty jako `$očekávání`,
- `assertArrayHasKey($klíč, $pole)` — ověřuje, zda pole `$pole` obsahuje klíč `$klíč`.

Samotný testovací scénář může skončit celou řadou stavů:

- úspěch — každý předpoklad je naplněn,
- selhání — alespoň jeden předpoklad není naplněn,
- chyba — při spuštění testu nastala chyba,
- risk — test neprovedl žádný assert,
- přeskok — test je za běhu označen jako „k přeskočení“,
- nekompletní — test je za běhu označen jako nekompletní.

3.1.2 Testovací dvojníci

Při psaní testů nastávají situace, kdy **vývojář nemůže nebo nechce použít službu v testovacím prostředí, která se používá v prostředí produkčním**. Může se jednat např. o API třetích stran. Většinou se testy spouští při kontinuální integraci a v případě, kdy neprojdou všechny testy, kód nelze nasadit na produkční servery. V takovém případě by výpadek služby třetí strany mohl ochromit proces nasazení vlastního kódu na produkci. V takových případech lze využít tzv. testovacích dvojníků (test doubles). Testovací dvojník se nemusí chovat do detailu přesně jako původní služba, nicméně poskytuje stejné rozhraní, které lze využít při testování. PHPUnit poskytuje podporu pro vytváření testovacích dvojníků. Konkrétně framework nabízí možnost vytvoření stubů a mocků.

Objekty, které nahrazují původní služby stejným rozhraním s vlastní, nakonfigurovanou implementací jednotlivých metod, se nazývají **stuby**. Vytvoření stubu pomocí frameworku PHPUnit lze vidět na ukázce 3.2. Dalším testovacím dvojníkem, který PHPUnit nabízí, je **mock**. Mock slouží k nahrazení objektu, kde cílem je ověřit chování, ke kterému dochází v testovaném kódu. Po vytvoření mocku je např. možné nastavit ověření, kolikrát má dojít k volání konkrétní metody a jaké mají být případné parametry v testovaném kódu.

```
<?php
use PHPUnit\Framework\TestCase;

final class StubTest extends TestCase
{
    public function testStub(): void
    {
        // Vytvoření stubu
        $stub = $this->createStub(GoogleSearchApi::class);

        // Nastavení chování metody search
        $stub->method('search')
             ->willReturn('result');

        $this->assertSame('result', $stub->search());
    }
}
```

Ukázka kódu 3.2: Vytvoření stubu pomocí PHPUnit

3.1.3 Pokrytí kódu testy

Další funkcionalitou, kterou PHPUnit poskytuje, je export týkající se pokrytí kódů testy. Export je možné nechat vygenerovat ve formátech jako HTML, XML nebo jako PHP kód. Ve výsledném exportu lze vyčíst např. celkové procento otestovaných řádků kódu nebo celkové procento otestovaných metod v kódu. Dále lze zvolit některé alternativní metriky jako jsou např. celkové pokrytí větví v kódu nebo celkové pokrytí cest v kódu testy. Každý vývojářský tým si tak může určit hlavní metriky, které určují kvalitu pokrytí kódu testy pro konkrétní projekt a jejich minimální akceptovanou hodnotu.

3.2 Selenium

Selenium je zastřešujícím projektem řady nástrojů a knihoven, které umožňují a podporují **automatizaci webových prohlížečů**. Poskytuje rozšíření pro emulaci interakce uživatele s prohlížeči, server pro škálování a infrastrukturu pro implementaci specifikace W3C WebDriver, která umožňuje psát jednotný kód pro komunikaci se všemi hlavními webovými prohlížeči.[27]

V praxi se Selenium často využívá k automatizovanému end-to-end testování. Nástroj umožňuje komunikaci se všemi nejpoužívanějšími prohlížeči v různých verzích a na různých operačních systémech. Samotné Selenium zaštiťuje pouze samotnou interakci s prohlížeči, případně škálování, neobsahuje samotné prostředky pro testování. K efektivnímu psaní testů je tak potřeba použít testovací framework, např. PHPUnit. Selenium se skládá z řady nástrojů, které budou popsány v následující sekci.

3.2.1 Komponenty

Selenium RC (remote control) je komponenta, která je v dnešní době již překonána komponentou WebDriver. Selenium RC se chová jako prostředník mezi samotnými prohlížeči a příkazy, které se mají v prohlížeči vykonat. Při spuštění testovacího scénáře, komponenta injektuje javascriptový program do prohlížeče. Díky tomuto programu je možné automaticky ovládat prohlížeč a vykonávat požadované příkazy. Selenium RC našlo své uplatnění v dobách, kdy ještě neexistoval standard, který by popisoval API, které prohlížeče implementují za účelem jednotného automatického ovládání. Právě tímto standardem je WebDriver rozhraní, které popisuje konsorcium W3C.

Komponenta **WebDriver** už pracuje nad definovaným rozhraním, které implementují jednotliví výrobci prohlížečů. Takové rozhraní umožňuje automatizované ovládání prohlížeče jako by se jednalo o reálného uživatele. Díky WebDriver komponentě tak vývojáři mohou psát end-to-end testy webových aplikací pro různé prohlížeče při zachování jednotného zdrojového kódu.

Selenium **Grid** umožňuje provádění skriptů na vzdálených počítačích (virtuálních nebo skutečných) směrováním příkazů odeslaných klientem na instance vzdáleného prohlížeče. Jeho cílem je poskytnout snadný způsob paralelního provádění testů na více počítačích. Selenium Grid pomáhá řešit především následující 2 problémy.

- Spouštění testů pro různé prohlížeče, různé verze prohlížečů a různé verze operačních systémů.
- Snížení času běhu testů jejich paralelizací.[27]

Dalším nástrojem, který Selenium poskytuje je **IDE** (integrated development environment). IDE je nástroj, který slouží k vytváření nových testovacích scénářů, bez nutnosti znalostí v oblasti programování. IDE existuje jako rozšíření do prohlížeče, které nahrává kroky uživatele a převádí je do zdrojového kódu.

3.2.2 Instalace a spuštění testů

Při reálném nasazení nástroje Selenium pro účely testování na projekt jsou potřeba následující kroky.

1. Nainstalování klientské knihovny komunikující přes rozhraní s WebDriverem. Pro jazyk PHP lze použít knihovnu `php-webdriver/php-webdriver`, kterou původně pro své účely vyvinula společnost Facebook.
2. Stažení WebDriveru konkrétního prohlížeče, ve kterém mají testy běžet. Svůj WebDriver, který implementuje W3C standard, nabízí např. prohlížeče Chrome, Firefox, Safari, Opera, Edge nebo Internet Explorer.
3. Nainstalování komponenty Selenium Grid. Tento krok není povinný, testy lze spouštět i bez Gridu. Nicméně Grid může řešit paralelizaci běhu testů nebo spouštění testů ve více prohlížečích a na různých operačních systémech.
4. Napsání samotných testů v testovacím frameworku. Příklad lze vidět na ukázce kódu 2.3.
5. Spuštění testů.

3.3 PHPStan

PHPStan je nástroj, který provádí **statickou analýzu kódu** nad kódem psaným v jazyce PHP. Jedná se open-source nástroj, který pomáhá s odhalením chybného kódu.

Mimo identifikace jednoznačně chybného kódu, který vždy skončí fatální chybou, dokáže PHPStan odhalit i potencionálně chybný kód nebo kód, který z logického pohledu nedává smysl a je zbytečný. Nástroj PHPStan pomáhá vývojářům **odhalit** výše zmíněné **problémy ještě před produkčním nasazením**.

Při psaní kódu v kompilovaných, silně typovaných jazycích, je zapotřebí uvést typ každé proměnné nebo např. návratový typ před samotným spuštěním programu. Překladač se ujistí, zda je všechno v pořádku a případně upozorní na chyby ve zdrojovém kódu, jako jsou např. volání nedefinované metody nebo předání nesprávného počtu parametrů funkci. Překladač tak funguje jako první obranná linie před zanesením chyb do aplikace. Nicméně PHP nepatří ke kompilovaným, silně typovaným jazykům a tuto obrannou linii neposkytuje. Pokud vývojář udělá chybu, aplikace selže až za běhu při vykonání inkriminovaného řádku. Při samotném testování vývojáři musí věnovat značné úsilí objevování potencionálních chyb, které by v jiných jazycích odhalil už samotný překladač a nemají tolik času věnovat se testovaní skutečné byznys logiky. Tuto vlastnost jazyka PHP se snaží suplovat nástroj PHPStan.[28]

3.3.1 Přehled hlídaných chyb

PHPStan hlídá řadu chyb a code smells (kód, který relativně funguje, ale indikuje možný hlubší problém návrhu kódu). V dokumentaci [28] se uvádí následující seznam — s poznámkou, že seznam se každou novou verzí rozšiřuje.

- Existence třídy použité v jazykových konstruktech jako `instanceof`, `catch` apod.
- Existence a viditelnost volaných funkcí a metod. Kontrola počtu předaných parametrů.
- Kontrola návratových typů metod s jejich definicemi.
- Existence a viditelnost použitých proměnných. Kontrola typu definice proměnné a jejího přiřazení.
- Správný počet předaných parametrů do metod `sprintf`, `printf` na základě formátovacího řetězce.
- Zbytečná přetypování proměnných.
- Detekce striktních porovnávacích operátorů mezi různými typy.

PHPStan je koncipovaný pro použití v rámci kontinuální integrace. Při spuštění pipeline se mimo klasických testů spustí i statická analýza kódu. V případě, kdy nástroj

identifikuje alespoň jednu chybu, pipeline spadne stejně, jako v případě nevyhovujícího jednotkového testu. Vývojář musí reportované chyby opravit a poté dojde ke spuštění další pipeline. PHPStan nabízí různé formáty chybových výstupů tak, aby byly např. kompatibilní s platformami jako GitHub nebo GitLab. Na ukázce 3.3 je uveden příklad spuštění statické analýzy na páté úrovni nad adresářem `src` s chybovým výstupem ve formátu pro GitHub.

```
vendor/bin/phpstan analyse src --level 5 --error-format=github
```

Ukázka kódu 3.3: Spuštění statické analýzy

3.3.2 Nasazení na již existující projekt

Řada nástrojů pro statickou analýzu funguje skvěle pro nové projekty, kde nástroj hlídá kvalitu kódu od samého začátku. Nicméně bývá velice náročné nástroje použít pro již existující projekt nebo dokonce pro historický projekt, který s sebou nese značný technologický dluh. Takové nástroje často při prvním spuštění odhalí tisíce chyb a odradí tak vývojáře od jejich používání. **PHPStan usnadňuje nasazení na existující projekty díky svým úrovním striktnosti**. Základní úroveň 0 kontroluje skutečně pouze ty nejzákladnější a nejtriviálnější chyby. Naproti tomu nejvyšší úroveň (aktuálně 8) kontroluje skutečně netriviální chyby, které mohou být velice těžko odhalitelné. Díky možnosti konfigurace úrovně je tak možné relativně snadno nasadit PHPStan na již existující projekt a postupem času navyšovat úrovně.

Další funkcionalita, která usnadňuje nasazení na již existující projekt, je základní linie (baseline) — ukázka 3.4. **Základní linie představuje seznam aktuálně hlášených chyb, které má PHPStan ignorovat**. V praxi se základní linie využívá v případech, kdy vývojáři vyžadují striktnější kontrolu kódu, nicméně nemají ještě opraveny všechny chyby, které PHPStan reportuje na vyšším, požadovaném levelu. Vývojář v takovém případě v konfiguraci zvedne level na požadovanou hodnotu a nechá PHPStan vygenerovat základní linii.

PHPStan nabízí ještě další možnost, jak ignorovat existující chyby. V konfiguračním souboru lze zapsat i s pomocí regulárních výrazů chyby, které nemá brát nástroj na zřetel. Příklad je uveden v ukázce 3.5.

3.3.3 Vlastní pravidla

PHPStan obsahuje celou řadu pravidel, která kontrolují kód. Nicméně poskytuje i možnost psaní vlastních pravidel, která mohou být specifická pouze pro konkrétní projekt. Vlastní pravidlo musí implementovat rozhraní `PHPStan\Rules\Rule` a následně musí být registrováno v konfiguračním souboru. Samotný PHPStan interně pracuje nad

```
parameters:
   ignoreErrors:
      -
         message: "#^If condition is always true\\.$#"
         count: 1
         path: src/Person/PersonHelper.php
      -
         message: "#^Call to an undefined method
            Grido\\\\Grid\\:\\:addFilterAsciiText\\(\\)\\.$#"
         count: 2
         path: src/Person/PersonGrid.php
```

Ukázka kódu 3.4: Příklad vygenerované základní linie

```
parameters:
   ignoreErrors:
      - '#^Property [a-zA-Z0-9\\]++::\$[a-zA-Z]*[rR]epository
         \([a-zA-Z0-9\\]++\) does not accept
         Doctrine\\ORM\\EntityRepository<[a-zA-Z0-9\\]++>.#'
```

Ukázka kódu 3.5: Ignorování chyb s pomocí regulárních výrazů

abstraktním syntaktickým stromem (AST), který reprezentuje syntaktickou strukturu kódu. Při psaní vlastních pravidel tak musí mít vývojář alespoň základní povědomí o AST. Na ukázce 3.6 je kód pravidla z oficiálního repozitáře nástroje PHPStan[29], které kontroluje, zda byla použitá konstanta v kódu dříve definována.

```php
<?php declare(strict_types = 1);

namespace PHPStan\Rules\Constants;

use PhpParser\Node;
use PHPStan\Analyser\Scope;
use PHPStan\Rules\RuleErrorBuilder;

/**
 * @implements \PHPStan\Rules\Rule<\PhpParser\Node\Expr\ConstFetch>
 */
class ConstantRule implements \PHPStan\Rules\Rule
{
   public function getNodeType(): string
   {
      return Node\Expr\ConstFetch::class;
   }

   public function processNode(Node $node, Scope $scope): array
   {
      if (!$scope->hasConstant($node->name)) {
         return [
            RuleErrorBuilder::message(sprintf(
               'Constant %s not found.',
               (string) $node->name
            ))->discoveringSymbolsTip()->build(),
         ];
      }

      return [];
   }

}
```

Ukázka kódu 3.6: Pravidlo pro kontrolu existence konstanty[29]

3.4 Rector

Nástroj Rector slouží především k automatizované úpravě kódu v repozitáři. **Nástroj podle předem vybraných pravidel upraví zdrojové soubory** a šetří tak čas strávený vývojářem při refaktoringu kódu. Rector lze využít ve dvou základních módech. Prvním je automatický jednorázový refaktoring zastaralého kódu. Druhým polem působnosti nástroje je automatické hlídání kvality kódu při kontinuální integraci.

Při spuštění za účelem jednorázového refaktoringu kódu lze nástroji nastavit buď jedno pravidlo, nebo celou sadu již předpřipravených pravidel. Po spuštění dojde k iteraci nad soubory se zdrojovými kódy a aplikaci vybraných pravidel. Předem připravených pravidel aktuálně existuje více než 600[30]. Jako příklad mohou posloužit následující:

- `RemoveAlwaysTrueIfConditionRector` — odstranění podmínek, které se vždy vyhodnotí jako pravda,
- `UnderscoreToCamelCaseLocalVariableNameRector` — přejmenování lokálních proměnných podle konvence camel case,
- `RemoveEmptyClassMethodRector` — odstranění prázdných metod,
- `RemoveUnusedAliasRector` — odstranění nepoužitých aliasů.

Řadu pravidel lze ještě dodatečně konfigurovat a přizpůsobit je konkrétnímu způsobu použití. Mimo jednotlivých pravidel mohou vývojáři využít sadu již připravených pravidel, které mají stejný cíl. Jedná se např. o sadu `CODE_QUALITY`, která integruje desítky jednotlivých pravidel za účelem zlepšení kvality zdrojového kódu. Na ukázce 3.7 je použito pravidlo `AddReturnTypeDeclarationRector`, které doplňuje metodám jejich návratový typ. Pravidlo je dále konfigurované tak, aby doplňovalo pouze typ `void` všem metodám v adresáři `tests`, které začínají klíčovým slovem `assert`.

Nástroj lze využít i dalším způsobem, a to jako součást kontinuální integrace. Stejně jako ve výše zmíněném případě, nejprve je potřeba nadefinovat pravidla, která se mají aplikovat. Poté už pouze stačí do CI pipeline přidat spuštění nástroje Rector s přepínačem *–dry-run*3.8. V případě, kdy nástroj aplikuje pravidlo a došlo by ke změně kódu, pipeline neprojde a vývojář musí svůj kód opravit.

```php
<?php

return static function (ContainerConfigurator
    $containerConfigurator): void {
   $parameters = $containerConfigurator->parameters();
   $parameters->set(Option::PATHS, [__DIR__ . '/tests']);

   $services = $containerConfigurator->services();
   $services->set(AddReturnTypeDeclarationRector::class)
      ->call('configure', [[
         AddReturnTypeDeclarationRector::METHOD_RETURN_TYPES =>
             ValueObjectInliner::inline([
            new AddReturnTypeDeclaration(
               '*',
               'assert*',
               new \PHPStan\Type\VoidType()
            ),
         ]),
      ]]);
};
```

Ukázka kódu 3.7: Pravidlo pro doplnění návratových typů

```
vendor/bin/rector process --dry-run
```

Ukázka kódu 3.8: Spuštění nástroje Rector v CI

3.5 Infection

Nástroj Infection slouží k provedení mutačního testování. Jak již bylo uvedeno v teoretické části, hlavním důvodem pro provádění mutačního testování je **ověření kvality ostatních typů testů**. Stejně jako celá řada testovacích nástrojů, i Infection interně pracuje s abstraktním syntaktickým stromem.

Nástroj pracuje se třemi hlavními metrikami. Primární metrikou je indikátor skóre mutace — **MSI**. Výpočet skóre MSI již byl nastíněn v teoretické čáti2.7. Další poskytovanou metrikou je pokrytí kódu mutacemi — **MCC**. Tato hodnota by se měla shodovat s hodnotou pokrytí kódu testy. Poslední významná metrika se nazývá **indikátor skóre kódu pokrytého mutacemi — Covered Code MSI**. Ta je hlavním indikátorem efektivity testů. Výpočet hodnoty probíhá podobně jako u metriky MSI. Nicméně MSI vyjadřuje procento počtu zabitých mutantů nad celým kódem, kdežto Covered Code

MSI udává procento zabitých mutantů pouze nad kódem, který je pokrytý testy.

3.5.1 Konfigurace a spuštění

Infection nabízí bohatou možnost konfigurace. Nástroj umožňuje např. nastavení frameworku, ve kterém jsou napsané jednotkové testy. Aktuálně podporuje PHPUnit, PhpSpec a Codeception. Dále je možné nastavit minimální akceptovanou hodnotu indikátoru MSI a Covered Code MSI (lze vidět na ukázce 3.9). Taková možnost má své využití zejména při použití mutačního testování v rámci kontinuální integrace. V případě, kdy jedna ze sledovaných metrik vyjde menší než požadované minimum, sestavení aplikace spadne. Další konfigurovatelnou položkou jsou samotné mutátory. V nastavení lze vyjmenovat jednotlivé mutátory, které se mají aplikovat. Další možností, jak použít pouze část mutátorů, je využití tzv. profilů. V dokumentaci jsou uvedeny různé profily, které vždy obsahují sadu souvisejících mutátorů. Jako příklad mohou posloužit např. profily `@arithmetic` (sada mutátorů souvisejících s aritmetickými operacemi) nebo `@boolean` (sada mutátorů souvisejících s booleovskými operacemi).

```
./infection.phar --min-msi=50 --min-covered-msi=75
```

Ukázka kódu 3.9: Spuštění mutačních testů

4 Implementace testů

Cílem praktické části této diplomové práce bylo zajistit větší stabilitu, dostupnost a menší chybovost webové aplikace z oblasti e-commerce. Hlavním prostředkem k zajištění výše zmíněných benefitů je zvýšení kvality automatizovaných testů běžících v rámci kontinuální integrace. Autor diplomové práce již řadu let pracuje jako vývojář na tomto projektu a v rámci praktické části budou popsány následující kroky, které byly implementovány za výše zmíněnými cíli.

- Rozšíření sady jednotkových a integračních testů,
- zavedení end-to-end testů,
- zavedení mutačních testů,
- zavedení statické analýzy,
- automatizace kontroly kvality kódu.

4.1 Rozšíření sady jednotkových a integračních testů

Repozitář již obsahoval řadu jednotkových a integračních testů, nicméně nebyly ani zdaleka pokryty všechny kritické části aplikace. Většina nových testů na již existující kód byla psána při úpravě existujícího kódu v rámci reálného úkolu, který autor práce dostal. Nejprve byly napsány testy na již existující funkcionalitu. Poté, co testy procházely, došlo k implementaci požadované nové funkcionality a následného ověření, zda testy stále procházejí. Nové testy na starý kód tak byly psány v rámci celého roku 2020 a jejich přidávání pokračuje i v roce 2021.

Jedním ze zásadních bloků, které bránily poctivějšímu psaní testů v minulosti, bylo složité **testování funkcionalit, kde hrají roli databázové entity**. Projekt využívá objektově relačního mapování, které poskytuje framework Doctrine 2. Doctrine převádí data, která jsou uložena v relační databázi, do podoby objektů (tříd) v PHP, které poskytují objektově orientovaný přístup. Při psaní testů je často nutností využití těchto objektů. Instanci takového objektu si může vývojář buďto vytvořit sám pomocí objektově orientovaného přístupu nebo může nechat tyto objekty vytvořit framework Doctrine 2 z relační databáze. Samotné vytvoření relační databáze a následná migrace smysluplných

dat může integrační testování velice zkomplikovat a velice prodloužit dobu nutnou pro provedení testů.

Tento problém řeší balíček `nelmio/alice`, což je generátor tzv. fixtures. Fixtures jsou podstrčená, nakonfigurovaná data, která se uloží do lokální relační databáze a umožňují pracovat s databázovými entitami. Pokud je použita vhodná abstrakce, kterou např. poskytuje framework Doctrine 2, databáze, kde jsou uložena data související s testy se může lišit od produkčního databázového stroje. Projekt, který je objektem zájmu diplomové práce běží na databázi PostgreSQL, nicméně testovací data se ukládají do SQLite. Ukázka konfigurace fixtures je vidět na ukázce 4.1. Na ukázce jsou nakonfigurovány 2 entity. První reprezentuje objekt administrátora a druhá webovou stránku. Balíček `nelmio/alice` poskytuje řadu předpřipravených funkcí pro ulehčení při generování dat. Na ukázce lze např. vidět užití zápisu `'<email()>'`, kdy balíček vygeneruje validní e-mailovou adresu. Poměrně snadno lze generovat i datum z určitého rozsahu, jak je vidět zde: `'<dateTimeBetween("-10 days", "now")>'`. Balíček řeší i vzájemné propojení entit mezi sebou (cizí klíče v relačních databázích). Na ukázce je vidět, že entita administrátora se odkazuje na druhou entitu pomocí zápisu `'@website1'`.

```
App\Model\Entity\Core\Admin:
   "admin1":
      id: 1
      username: admin
      firstName: Radek
      lastName: Meduna
      email: '<email()>'
      password: "$2y$10$arltcXe9dNa9Ke/cOkn1UeAmJi5oDVMsad5asd45asS"
      role: admin
      website: ['@website1']

App\Model\Entity\Core\Website:
   "website1":
      name: radekmeduna.cz
      locale: cs
      country: @country1
      code: radekmeduna_cz
      createdAt: '<dateTimeBetween("-10 days", "now")>'
```

Ukázka kódu 4.1: Konfigurace fixtures

Po konfiguraci entit je nutné uložit data do databáze. Samotný balíček poskytuje roz-

hraní, díky kterému je tato operace velice snadná. Na ukázce 4.2 je vidět celý postup při využití frameworků Nette, Doctrine a balíčku `nelmio/alice`. Nejprve je potřeba získat cesty ke konfiguračním souborům. Na základě konfiguračním souborů dojde k vytvoření entit, které se následně jednoduchým způsobem uloží do relační databáze.

Za využití fixtures lze velice efektivně testovat kód závislý na databázi pomocí předkonfigurovaných vstupních dat. Samotné nahrání dat do databáze, které se děje na pozadí, je velice rychlé a běh jednotkových testů příliš nebrzdí.

Další výzvu představovala standardizace **testování API třetích stran** v rámci projektu. Ne vždy je ideální testovat API třetích stran na produkčním endpointu. Hlavním důvodem může být stabilita samotného API. Když testy získávají data z API, je nutné, aby API fungovalo, jinak neprojdou testy. Jinak je možné dostat se do situace, kdy jedno dočasně nefungující API zabrání nahrání dat na produkci, protože neprocházejí testy. Dalším důvodem může být rychlost provedení testů, získávání dat z externího API může výrazně brzdit testovací proces. V praxi se tak často při testování používají jako odpovědi API třetích stran staticky definované odpovědi v odpovídajícím formátu.

Tento problém lze velice elegantně řešit v PHP frameworku Nette. Framework řeší získávání služeb pomocí dependency injection (DI). „Podstatou dependency injection (DI) je odebrat třídám zodpovědnost za získávání objektů, které potřebují ke své činnosti (tzv. služeb) a místo toho jim služby předávat při vytváření.“[31] V praxi princip DI funguje tak, že vývojář vytvoří službu a zaregistruje ji do DI kontejneru pomocí zápisu do konfiguračního souboru. Kdekoli chce tuto službu využít, DI kontejner ji vloží do konstruktoru třídy. V případě služby komunikující s externím API, která má být zároveň podrobena testování lze postupovat následovně.

1. Vytvoření rozhraní služby, které bude implementováno službou komunikující s API. Ukázka 4.3.
2. Vytvoření služby implementující rozhraní, které komunikuje s API. Ukázka 4.4.
3. Zaregistrování služby do produkčního DI kontejneru. Ukázka 4.5.
4. Vytvoření služby implementující rozhraní, která vrací statické odpovědi. Ukázka 4.6.
5. Zaregistrování služby do testovacího DI kontejneru. Ukázka 4.7.

Jelikož produkční a testovací prostředí může mít svůj vlastní DI kontejner, problém za vývojáře vyřeší samotný framework. Při použití služby vývojář vždy využívá rozhraní `WeatherInfoApi`. DI kontejner poskytne implementaci na základě toho, zda je služba vyžadována v produkčním nebo testovacím prostředí. Běh testů a jejich výsledek tak není závislý na externím API.

Další úprava jednotkových a integračních testů se zaměřovala na to, aby **nezáleželo na pořadí testů, ve kterém jsou spouštěny**. To je jeden z bodů principu F.I.R.S.T1.7. Tento princip byl často porušen v integračních testech, které využívají napojení na databázi. Pokud integrační test mění nebo aktualizuje data, měl by po provedení assertu vrátit data do původní podoby. Další užitečný nástroj pro vrácení aplikace do původního stavu před testem poskytuje samotný testovací framework PHPUnit. PHPUnit poskytuje metodu `tearDown()`, která je zavolána po každém spuštění testu z dané třídy a má za úkol vrátit aplikaci do původního stavu před spuštěním testu.

```php
<?php

use Doctrine\ORM\EntityManagerInterface;
use Nelmio\Alice\Fixtures\Loader;
use Nette\Utils\Finder;

final class AliceLoader
{

   private Loader $aliceLoader;

   private EntityManagerInterface $entityManager;

   public function __construct(
        Loader $aliceLoader,
        EntityManagerInterface $entityManager
    ) {
      $this->aliceLoader = $aliceLoader;
      $this->entityManager = $entityManager;
   }

   public function loadData(string $folderPath): void
   {
      // získání cest všech konfiguračních souborů s fixtures
      foreach (Finder::find('*.neon')->from($folderPath) as $file) {
         $filePath = $file->getPathname();
         // vytvoření entit z konfiguračních souborů pomocí balíčku
             nelmio/alice
         $entities = $this->aliceLoader->load($filePath);
         foreach ($entities as $entity) {
            // připravení entit k budoucímu uložení do databáze
            $this->entityManager->persist($entity);
         }
      }

      // perzistentní uložení entit do relační databáze
      $this->entityManager->flush();
   }
}
```

Ukázka kódu 4.2: Načtení testovacích dat do databáze

```php
<?php

interface WeatherInfoApiInterface
{
   function getWeatherInfo(): string;
}
```

Ukázka kódu 4.3: Rozhraní pro komunikaci s API

```php
<?php

final class WeatherInfoApi implements WeatherInfoApiInterface
{

   public function getWeatherInfo(): string
   {
      $client = new GuzzleHttp\Client();
      // požadavek na API
      $response = $client->request(
         'GET',
         'https://api.weather.com/weather-info',
         ['auth' => ['user', 'pass']]
      );
      return (string) $response->getBody();
   }
}
```

Ukázka kódu 4.4: Služba komunikující s API

```
services:
   weatherInfoApi.client: WeatherInfoApi
```

Ukázka kódu 4.5: Registrace služby do DI kontejneru

```php
<?php

final class WeatherInfoApiTest implements WeatherInfoApiInterface
{
   public function getWeatherInfo(): string
   {
      // vrácení statické odpovědi bez odeslání požadavku na API
      return '
      {
         "weatherInfo":{
            "temperature":"25",
            "humidity":"29",
            "forecast":{
               "temperature":[{"monday":"23"}, {"tuesday":"26"}]
            }
         }
      }';
   }
}
```

Ukázka kódu 4.6: Služba poskytující statickou odpověď bez napojení na API

```
services:
   weatherInfoApi.client: WeatherInfoApiTest
```

Ukázka kódu 4.7: Registrace služby do DI kontejneru pro testy

4.2 Zavedení end-to-end testů

End-to-end testy jsou velice komplexní testy, které otestují aplikaci jako celek. Dle dříve zmíněného schématu testovací pyramida1.1 je nutné komplexní testy volit velice pečlivě, neboť jsou ze všech druhů testů také těmi nejdéle běžícími. Pro potřeby zavedení end-to-end testů došlo k sepsání testovacích scénářů, které pokrývají pouze ty nejkritičtější průchody aplikací.

- Přihlášení uživatele do aplikace.
- Vložení produktu do košíku a dokončení objednávky.
- Kontrola viditelnosti důležitých požadovaných prvků na hlavní stránce.

Každý z těchto scénářů se skládá z jednotlivých kroků. V tabulce 4.1 je detailněji popsán testovací scénář „kontrola viditelnosti důležitých požadovaných prvků na hlavní stránce“. První 3 kroky jsou implementovány na ukázce 4.8. V rámci ukázky jsou použity principy konceptu objektového modelu stránky, které byly zmíněny v teoretické části práce2.3.1. Objektový model stránky pomáhá s dlouhodobou udržitelností testovacích scénářů. Ukázka jednoduché třídy poskytující objektové rozhraní po vzoru objektového modelu stránky lze vidět na ukázce 4.9.

Tabulka 4.1: Dílčí kroky testovacího scénáře

krok	**akce**
1	navigace na URL hlavní stránky
2	kontrola viditelnosti loga aplikace
3	kontrola viditelnosti košíku
4	kontrola viditelnosti produktů
5	kontrola viditelnosti bannerů
6	kontrola viditelnosti článků
7	kontrola viditelnosti patičky

End-to-end testy byly psány pro prohlížeč Google Chrome. Při implementaci end-to-end testů pomocí rodiny nástrojů Selenium lze optimalizovat běh samotných testů různými způsoby. Jedním z nejzásadnějších faktorů může být použití prohlížeče v tzv. headless módu. Při standardním spuštění Selenium testů se na pozadí spustí instance prohlížeče, ve které se provádí naprogramované jednotlivé kroky podobně, jako kdyby je naklikal uživatel. Jednotlivé úkony lze většinou při spuštění testů dokonce vidět (ve

velice zrychleném průběhu). Nicméně **prohlížeč spuštěný v headless módu je speciálně určený pro automatizaci a nevykresluje žádné grafické rozhraní**. Ve výsledku tak může dojít ke **značné časové úspoře** (v tomto konkrétním případě téměř poloviční). Testy používají pro psaní assertů testovací framework PHPUnit, který je využit i na jednotkové a integrační testy.

```php
<?php

use Facebook\WebDriver\Remote\RemoteWebDriver;

final class HomepageTest
{

   private RemoteWebDriver $webDriver;

   public function __construct(RemoteWebDriver $webDriver)
   {
      $this->webDriver = $webDriver;
   }

   private function testHomepage(): void
   {
      // navigace na URL hlavní stránky
      $this->webDriver->get(HomePage::getUrl());
      $logo = $this->webDriver->findElement(
         HomePage::getLogoWebDriverBy()
      );
      $logoImg = $logo->findElements(
         HomePage::getLogoImageWebDriverBy()
      );
      // kontrola viditelnosti loga aplikace
      $this->assertEquals(1, count($logoImg));

      $basketPanel = $this->webDriver->findElement(
         BasketPanelControl::getBasketWebDriverBy()
      );
      // kontrola viditelnosti košíku
      $this->assertTrue($basketPanel->isDisplayed());
   }
```

Ukázka kódu 4.8: Implementace prvních tří kroků testovacího scénáře

```php
<?php

use Facebook\WebDriver\WebDriverBy;

final class BasketPanelControl
{

   private const BASKET_LOCATOR = '#basket-section';

   public static function getBasketWebDriverBy(): WebDriverBy
   {
      return WebDriverBy::cssSelector(self::BASKET_LOCATOR);
   }

}
```

Ukázka kódu 4.9: Jednoduchá implementace objektového modelu stránky

4.3 Zavedení mutačních testů

Pro potřebu mutačního testování byl zvolen nástroj Infection. Narozdíl od předchozích typů testů, pro zavedení mutačního testování není potřeba psát žádný vlastní PHP kód. Pro správné fungování je třeba nainstalovat nástroj Infection, nástroj generující pokrytí kódu kompatibilní s Infection (Xdebug, phpdbg nebo pcov) a správná konfigurace.

Při prvním spuštění nástroj nabídne možnost interaktivní konfigurace. Výsledkem je konfigurační soubor `infection.json.dist`, který ovlivňuje chování samotného nástroje. Konfigurační soubor nabízí možnost nastavení např. cest, které podléhají mutační analýze, cestu k souboru s logy nebo nastavení mutátorů. Jednoduchý konfigurační soubor z projektu je uveden na ukázce 4.10. Zcela zásadní je konfigurační hodnota `minMsi`. Při použití mutačního testování v rámci kontinuální integrace je tato hodnota určující pro fakt, zda mutační testy projdou nebo selžou. Pokud je výsledná vypočtená hodnota MSI mutačního testování nižší než nakonfigurované minimum, spuštěná pipeline selže a vývojář musí opravit zdrojový kód.

Další zajímavou možností je konfigurace použitých mutátorů. Pod klíčem `mutators` je možné vybírat mezi různými profily. Každý profil obsahuje sadu mutátorů, které spolu souvisí. Existuje např. profil `@arithmetic`, který obsahuje mutátory, týkající se aritmetických operací. V projektu byl nastaven výchozí profil `@default`, který používá většinu existujících mutátorů.

Při nasazení mutačního testování byla nastavena minimální hodnota MSI na velice malé číslo, protože testy nebyly v moc dobré kondici. Postupem času se minimální hodnota pomalu navyšuje tak, aby se zvedala minimální kvalita vznikajících testů.

```
{
    "source": {
        "directories": [
            "app",
            "packages"
        ]
    },
    "logs": {
        "text": "infection.log"
    },
    "mutators": {
        "@default": true
    },
    "minMsi": 50,
    "phpUnit": {
        "customPath": "vendor\/bin\/phpunit"
    },
    "bootstrap":"./tests/bootstrap.php"
}
```

Ukázka kódu 4.10: Jednoduchý konfigurační soubor nástroje Infection

4.4 Zavedení statické analýzy

Pro potřeby statické analýzy zdrojového PHP kódu byl zvolen nástroj PHPStan. PHPStan velice dobře funguje jako součást kontinuální integrace. Jednou z velkých výhod nástroje je možnost konfigurace úrovně striktnosti, na které kontroluje zdrojový kód. Tento aspekt je zejména velice důležitý pro nasazení na již existující projekt, což je i případ tohoto projektu.

Při nasazení PHPStanu na projekt byl jako první zvolen level 0, který odhalil jednotky chyb, které byly rychle opraveny. Při postupném zvedání levelů se objevovalo více a více chyb. PHPStan nabízí možnost vytvoření baseline(3.4), což je soubor, který uchovává chyby, které při statické analýze ignoruje. Této funkce se dá využít při přechodu na vyšší level, aniž by byly opraveny všechny existující chyby.

Na nižších levelech docházelo k manuálnímu opravování chyb, nicméně při snaze o přechod na vyšší levely (v tomto případě konkrétně na level 6) může být tento postup zdlouhavý a značně neefektivní. Řadu chyb lze automaticky vyřešit vhodným použitím nástroje Rector.

První okamžik, kdy bylo takřka nemožné řešit všechny identifikované chyby manuálně, nastal při přechodu na šestou úroveň. V předchozích přechodech vždy PHPStan

identifikoval maximálně stovky chyb, přičemž řada z nich spolu souvisela a vyřešením jednoho problému zmizelo více chyb. Nicméně při snaze o přechod na level 6 PHPStan hlásil téměř 18 tisíc chyb. Drtivá většina chyb souvisela s tím, že PHPStan na levelu 6 kontroluje, zda má každá metoda nadefinovaný svůj návratový typ. Úryvek reportu chyb je vidět na ukázce 4.11.

Obrázek 4.1: Počet chyb při první snaze o přechod na PHPStan level 6

V aplikaci chyběla definice návratového typu zejména u metod, které nevracejí nic (přesněji vracejí `void`). Po provedení analýzy byl vyvozen závěr, že většina metod, kde chybí definice návratového typu spadá do jedné z následujících kategorií.

- Jedná se o metodu, jejíž název začíná na `action` nebo `render`, která se nachází ve třídě, která končí názvem `Presenter`. To jsou metody, které jsou typické pro framework Nette a jejich úkolem je vykreslení obsahu do šablony. V jejich případě nikdy nedává smysl jiný návratový typ než `void`.
- Jedná se o metodu s názvem `save` nebo `delete`, která se nachází ve třídě s názvem končícím na `Repository`. To jsou metody typické pro repozitáře spravující entity frameworku Doctrine. V jejich případě nikdy nedává smysl jiný návratový typ než `void`.
- Jedná se o metodu, jejíž název začíná na `set` a které se nachází v adresáři `Entity`. To jsou settery, které jsou typické pro entity frameworku Doctrine. V jejich případě nikdy nedává smysl jiný návratový typ než `void`.

Pro automatické doplnění návratového typu `void` na základě výše uvedených pravidel byl použit nástroj Rector, konkrétně pravidlo `AddReturnTypeDeclarationRector`. Výsledná konfigurace tohoto pravidla dle tří výše zmíněných bodů lze vidět na ukázce 4.12.

Rector lze pustit ve dvou různých módech. Nejprve je možnost spustit nakonfigurované pravidlo nanečisto. Rector v konzoli zobrazí změny, které by provedl při ostrém spuštění a kód nijak neupravuje (obrázek 4.2). Pokud je vývojář se změnami spokojený a odpovídají jeho představám, spustí Rector v ostrém módu. V takovém případě Rector upraví zdrojové kódy dle nadefinovaných pravidel.

Další velkou výhodu, kterou PHPStan poskytuje a která se v projektu velice osvědčila, je využití generik v jazyce PHP. Jazyk PHP nepodporuje používání generik ani pomocí nativních typů ani pomocí standardizovaných anotací. Generika umožňují např.

```
 ------------------------------------------------------------------------
  Line   packages/api/src/Response/AddContactResponse.php
 ------------------------------------------------------------------------
  29     Method Api\Response\AddContactResponse::process() has no
     return typehint specified.
 ------------------------------------------------------------------------

 ------------------------------------------------------------------------
  Line   packages/api/src/Response/OptOutContactResponse.php
 ------------------------------------------------------------------------
  29     Method Api\Response\OptOutContactResponse::process() has no
     return typehint specified.
 ------------------------------------------------------------------------

 ------------------------------------------------------------------------
  Line   packages/api/src/Response/SendEmailResponse.php
 ------------------------------------------------------------------------
  31     Method Api\Response\SendEmailResponse::process() has no
     return typehint specified.
 ------------------------------------------------------------------------
```

Ukázka kódu 4.11: Úryvek chybového výstupu nástroje PHPStan

návrat datového typu na základě předaného argumentu. PHPStan tak poskytuje další míru kontroly nad rámec samotného jazyka a poskytuje tím vývojářům další nástroj, který jim pomáhá psát funkční a bezpečný kód. Příklad rozhraní využívající generika pomocí zápisu čitelného pro PHPStan lze vidět na ukázce 4.13.

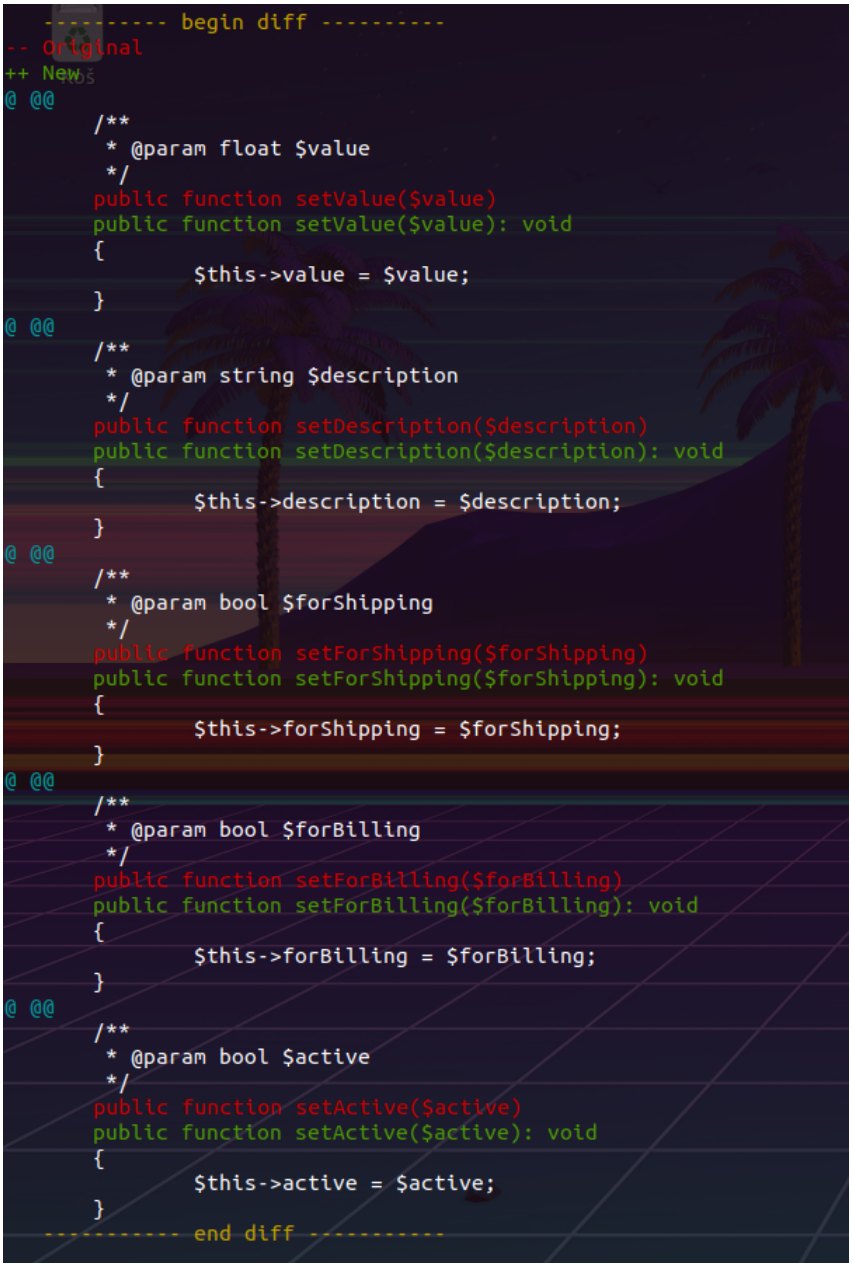
```php
<?php

// use statementy vynechány pro udržení přehlednosti

return static function (
    ContainerConfigurator $containerConfigurator
): void {
    $parameters = $containerConfigurator->parameters();
    $parameters->set(Option::PATHS, [__DIR__ . '/app']);

    $services = $containerConfigurator->services();
    $services->set(AddReturnTypeDeclarationRector::class)
        ->call('configure', [
            [
                AddReturnTypeDeclarationRector::METHOD_RETURN_TYPES =>
                    ValueObjectInliner::inline(
                    [
                        new AddReturnTypeDeclaration(
                            '*Entity*',
                            'set*',
                            new VoidType
                        ),
                        new AddReturnTypeDeclaration(
                            '*Presenter',
                            'action*',
                            new VoidType
                        ),
                        new AddReturnTypeDeclaration(
                            '*Presenter',
                            'render*',
                            new VoidType
                        ),
                        new AddReturnTypeDeclaration(
                            '*Repository',
                            'save',
                            new VoidType
                        ),
                        new AddReturnTypeDeclaration(
                            '*Repository',
                            'delete',
                            new VoidType
                        )
                    ]
                ),
            ],
        ]);
};
```

Ukázka kódu 4.12: Doplnění návratového typu `void` podle stanovených pravidel



Obrázek 4.2: Výstup nástroje Rector při spuštění nanečisto

```
/**
 * @template T
 * @param class-string<T> $className
 * @return T
 */
function getModuleByClassName(string $className): ModuleInterface;
```

Ukázka kódu 4.13: Typické rozhraní metod využívající generika

5 Automatizace

Pro potřeby automatizace spouštění výše uvedených nástrojů zajišťujících kvalitu kódu byl použit nástroj GitLab CI/CD. Jedná se o nástroj, který je přímo zabudovaný v platformě GitLab a podporuje metody kontinuálního vývoje softwaru.

GitLab CI/CD se konfiguruje pomocí souboru s názvem `.gitlab-ci.yml` umístěného v kořenovém adresáři repozitáře. Tento soubor vytvoří pipeline, která se automaticky spustí při změnách v repozitáři. Pipeline se skládá z jedné nebo více fází, které běží v definovaném pořadí a každá může obsahovat jednu nebo více úloh, které běží paralelně. Tyto úlohy jsou prováděny GitLab runnery.[32]

V dokumentaci [32] jsou popsány základní koncepty, které se týkají GitLab CI/CD.

- Pipelines — prostřednictvím pipeline dochází k vytvoření struktur CI/CD procesu.
- Job — zakladní element souboru `.gitlab-ci.yml`. Každý job vykonává definované operace.
- Stage — definují skupiny jobů. Každý job musí náležet nějaké stage. Stage definuje pořadí vykonávání jobů.
- CI/CD proměnné — znovupoužitelné hodnoty založené na konceptu klíč hodnota.
- Artifakt — znovupoužitelný výstup některého jobu.

Na zkrácené ukázce 5.1 souboru `.gitlab-ci.yml`, který je v plné podobě součástí přílohy k diplomové práci, lze vidět automatizaci spouštění nástrojů souvisejících s hlídáním kvality kódu. V souboru jsou definovány 3 fáze:

- build,
- code quality,
- deploy.

Ve zkrácené ukázce jsou popsány joby týkající se pouze fáze *code quality*. Tato fáze sdružuje následující joby:

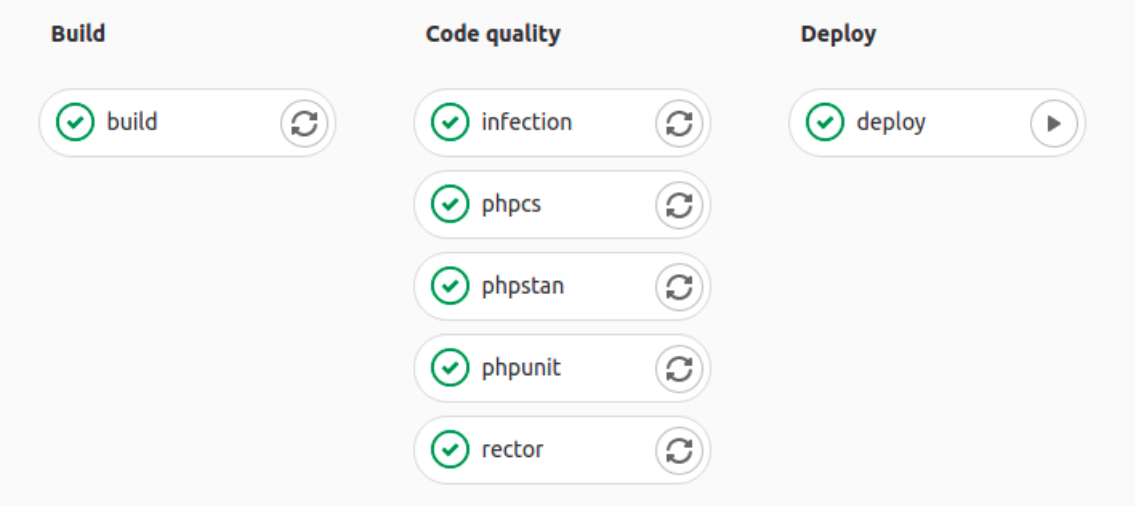
- infection — spuštění mutačních testů pomocí nástroje Infection. V konfiguračním souboru v kořenu repozitáře je uvedena minimální hodnota MSI 90. Pokud je výsledná hodnota nižší, pipeline neprojde.
- phpstan — spuštění statické analýzy pomocí nástroje PHPStan. V případě návratu chybového kódu nástroj uloží svůj výstup do souboru `phpstan-report.xml` ve formátu `junit`.
- phpunit — spuštění testů využívajících testovací framework PHPUnit. V případě návratu chybového kódu nástroj uloží svůj výstup do souboru `phpunit-report.xml` ve formátu `junit`.
- phpcs — spuštění kontroly definovaného coding standardu. V případě návratu chybového kódu nástroj uloží svůj výstup do souboru `phpcs-report.xml` ve formátu `junit`.
- rector — spuštění kontroly kvality kódu pomocí nástroje Rector. V konfiguračním souboru v kořenu repozitáře jsou uvedeny sady pravidel, které hlídají striktní kvalitu kódu a detekují mrtvý kód.

Výsledná pipeline je graficky zobrazena na obr. 5.1. Po fázi zajišťující kvalitu kódu je nadefinovaná fáze nasazení. Tato fáze je definována jako manuální, nespouští se tedy automaticky, ale samotné nasazení musí iniciovat vývojář kliknutím na tlačítko. Fáze nasazení je dostupná pouze v případě, kdy všechny předchozí joby proběhly v pořádku. Pokud tedy selže alespoň jedna kontrola kvality kódu, nasazení není možné a je nutné chybu opravit.

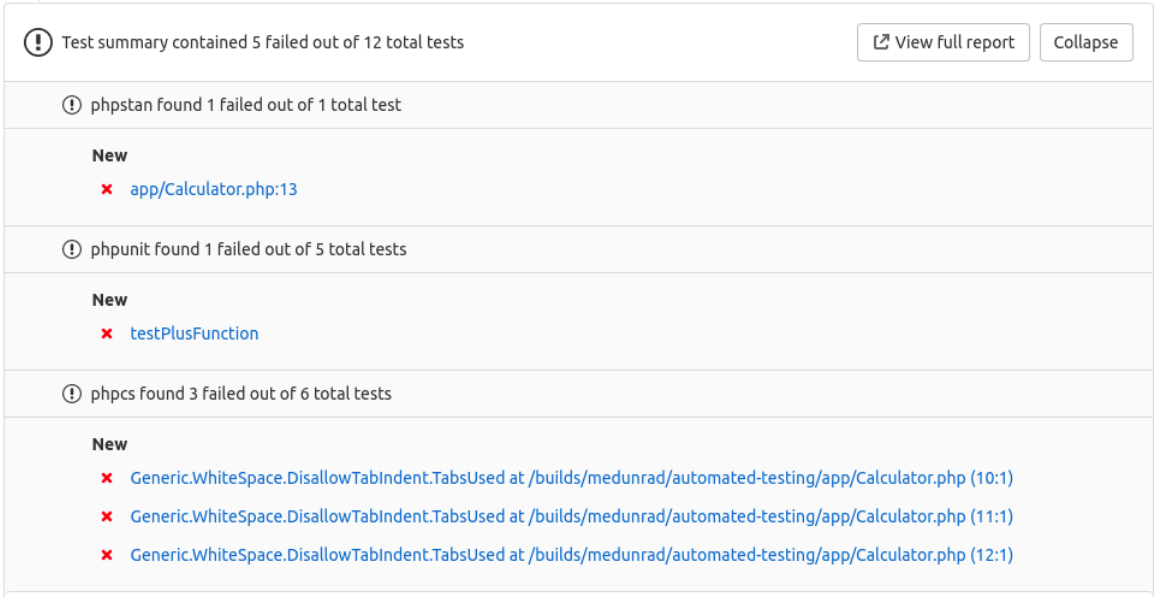
GitLab hezky spolupracuje s nástroji, které poskytují chybový výstup ve formátu `junit`. V takovém případě GitLab na detailu žádosti o spojení kódu zobrazí přehledný widget (obr. 5.2), který zobrazuje chyby, které brání integraci kódu. Zároveň GitLab přidá do detailu pipeline záložku se souhrnem provedených testů (obr. 5.3). Z dat je možné vyčíst kolik testů bylo celkem spuštěno nebo jaká byla procentuální chybovost testů.

```
image: php:7.4-fpm
stages:
    - build
    - code quality
    - deploy
infection:
    stage: code quality
    before_script:
        - pecl install pcov > /dev/null && docker-php-ext-enable pcov
    script:
        - bin/infection.phar
phpstan:
    stage: code quality
    script:
        - vendor/bin/phpstan analyse --error-format=junit
            --no-progress > phpstan-report.xml
    artifacts:
        expire_in: 1 week
        reports:
            junit: phpstan-report.xml
phpunit:
    stage: code quality
    script:
        - vendor/bin/phpunit --log-junit phpunit-report.xml
    artifacts:
        expire_in: 1 week
        reports:
            junit: phpunit-report.xml
phpcs:
    stage: code quality
    script:
        - vendor/bin/phpcs --standard=PSR2 app tests --report=junit
            > phpcs-report.xml
    artifacts:
        expire_in: 1 week
        reports:
            junit: phpcs-report.xml
rector:
    stage: code quality
    script:
        - vendor/bin/rector process app tests --dry-run
```

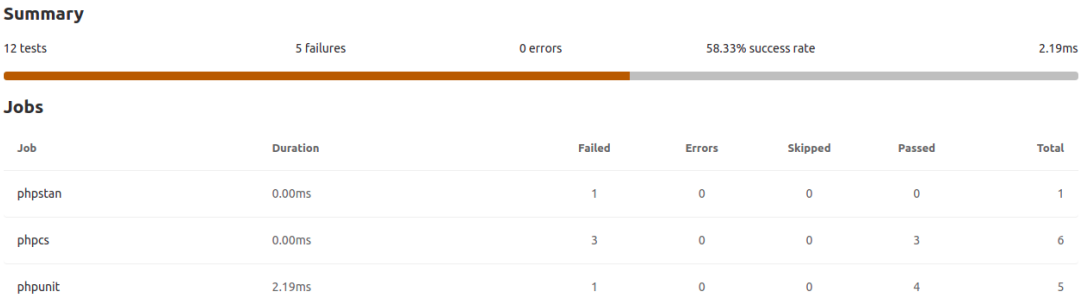
Ukázka kódu 5.1: Zkrácená ukázka souboru `.gitlab-ci.yml`



Obrázek 5.1: Grafická podoba výsledné pipeline



Obrázek 5.2: Widget zobrazující chyby bránící integraci kódu



Summary

12 tests | 5 failures | 0 errors | 58.33% success rate | 2.19ms

Jobs

Job	Duration	Failed	Errors	Skipped	Passed	Total
phpstan	0.00ms	1	0	0	0	1
phpcs	0.00ms	3	0	0	3	6
phpunit	2.19ms	1	0	0	4	5

Obrázek 5.3: Souhrn provedených testů

6 Závěr

Diplomová práce prezentovala různé techniky testování a zajišťování kvality kódu softwarových aplikací. V teoretické části byly představeny různé možnosti zajišťování kvality kódu, popsány jejich hlavní výhody a motivace k jejich nasazení. V praktické části došlo k představení konkrétních nástrojů specifických pro programovací jazyk PHP. Popsané techniky byly implementovány v konkrétních open-source nástrojích a ukázky uvedeny v diplomové práci. Dále byla popsána automatizace spouštění vybraných testovacích nástrojů v rámci kontinuální integrace. Tato automatizace byla implementována pomocí nástroje GitLab CI/CD a uvedena v příloze.

V rámci diplomové práce došlo k rozšíření sady jednotkových a integračních testů, zavedení end-to-end testů, zavedení mutačních testů a zavedení statické analýzy v reálném e-commerce projektu. Všechny tyto techniky jsou spouštěny automatizovaně na platformě GitLab. Dopad implementace výše zmíněných praktik je relativně složité kvantitativně měřit. Nicméně ze subjektivního pohledu došlo ke snížení času stráveného na code review. Spoustu neduhů je detekováno automaticky ještě před samotným code review. Zavedení statické analýzy na relativně striktní level a doplnění sady testů pomáhá s odhalením chyb v raných fázích vývoje a zajišťuje tak větší stabilitu samotného produktu. Zavedení statické analýzy také zlepšilo programovací návyky řady vývojářů. Objektivně lze pozorovat, že v prvních dnech po zvýšení levelu striktnosti velice často docházelo k detekci chyb týkajících se statické analýzy. Po pár dnech si vývojáři zvykli na nové, striktnější kontroly a počet chyb začal klesat.

Co se týče výhledu do budoucnosti, určitě je dobrý nápad pokračovat ve zvyšování striktnosti statické analýzy a v pečlivém psaní testů. Dalším krokem by mohla být definice minimálního pokrytí kódu testy, kontrolovaná v rámci kontinuální integrace a postupné navyšování této hodnoty na rozumnou mez.

Literatura

[1] TASSEY, Gregory. The economic impacts of inadequate infrastructure for software testing. National Institute of Standards and Technology, 2002.

[2] DOOLEY, John. Software development and professional practice. Apress, 2011.

[3] FOWLER, Martin. The Practical Test Pyramid. Martin Fowler [online]. 2018 [cit. 2020-11-21]. Dostupné z: https://martinfowler.com/articles/practical-test-pyramid.html

[4] Fitting Frontend Design Testing into the Test Pyramid [online]. In: . [cit. 2020-11-21]. Dostupné z: https://warhol.io/blog/frontend-design-testing-test-pyramid

[5] FOWLER, Martin; FOEMMEL, Matthew. Continuous integration. 2006.

[6] KUMAR, Pankaj. Aspect-Oriented Software Quality Model: The AOSQ Model. Advanced Computing: An International Journal (ACIJ), 2012, 3.2: 105-118.

[7] MARTIN, Robert C. Clean code: a handbook of agile software craftsmanship. Pearson Education, 2009.

[8] FOURNIER, Greg. Essential software testing: a use-case approach. CRC Press, 2008.

[9] JORGENSEN, Paul C. Software testing: a craftsman's approach. CRC press, 2018.

[10] BERGMANN, Sebastian. Getting Started with PHPUnit 9. PHPUnit [online]. [cit. 2021-01-09]. Dostupné z: https://phpunit.de/getting-started/phpunit-9.html

[11] COPELAND, Lee. A practitioner's guide to software test design. Artech House, 2004.

[12] BOSE, Shreya. What is End to End Testing? Browserstack [online]. 2020 [cit. 2020-11-21]. Dostupné z: https://www.browserstack.com/guide/end-to-end-testing

[13] RICCA, Filippo; LEOTTA, Maurizio; STOCCO, Andrea. Three open problems in the context of E2E web testing and a vision: NEONATE. In: Advances in Computers. Elsevier, 2019. p. 89-133.

[14] FOWLER, Martin. PageObject. Martin Fowler [online]. 2013 [cit. 2020-11-17]. Dostupné z: https://martinfowler.com/bliki/PageObject.html

[15] BATH, Graham, Rex BLACK, Alexander PODELKO, Andrew POLLNER a Randy RICE. Foundation Level Specialist Syllabus Performance Testing. International Software Testing Qualifications Board [online]. 2018 [cit. 2020-12-30]. Dostupné z: https://www.istqb.org/documents/ISTQB%20CTFL-PT%20Syllabus%202018%20GA.pdf

[16] WALTON, Philip. Web Vitals. Web.dev [online]. 2020, 2020-04-21 [cit. 2020-12-30]. Dostupné z: https://web.dev/vitals/

[17] PAL, Sayan Kumar. Coding Standards and Guidelines. GeeksforGeeks [online]. 2019 [cit. 2020-12-29]. Dostupné z: geeksforgeeks.org/coding-standards-and-guidelines/

[18] KOC, Ugur, et al. Learning a classifier for false positive error reports emitted by static code analysis tools. In: Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages. 2017. p. 35-42.

[19] SINGH, Devarshi, et al. Evaluating how static analysis tools can reduce code review effort. In: 2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). IEEE, 2017. p. 101-105.

[20] Sandwich Testing | Software Testing. GeeksforGeeks [online]. 2019 [cit. 2020-12-12]. Dostupné z: https://www.geeksforgeeks.org/sandwich-testing-software-testing/

[21] PAPADAKIS, Mike, et al. Mutation testing advances: an analysis and survey. In: Advances in Computers. Elsevier, 2019. p. 275-378.

[22] PAPADAKIS, Mike; MALEVRIS, Nicos. An empirical evaluation of the first and second order mutation testing strategies. In: 2010 Third International Conference on Software Testing, Verification, and Validation Workshops. IEEE, 2010. p. 90-99.

[23] WONG, Weichen Eric. On mutation and data flow. 1993. PhD Thesis. Purdue University.

[24] CHEKAM, Thierry Titcheu, et al. An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption. In: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE). IEEE, 2017. p. 597-608.

[25] CHAUHAN, Vinod Kumar. Smoke Testing. Int. J. Sci. Res. Publ, 2014, 4.1: 2250-3153.

[26] BERGMANN, Sebastian. 2. Writing Tests for PHPUnit. PHPUnit [online]. [cit. 2021-02-06]. Dostupné z: https://phpunit.readthedocs.io/en/9.5/writing-tests-for-phpunit.html

[27] The Selenium Browser Automation Project. Selenium [online]. [cit. 2021-02-14]. Dostupné z: https://www.selenium.dev/documentation/en/

[28] MIRTESOVÁ, Petra. PHPStan: Find Bugs In Your Code Without Writing Tests! PHPStan [online]. 2016, 2016-12-04 [cit. 2021-01-23]. Dostupné z: https://phpstan.org/blog/find-bugs-in-your-code-without-writing-tests

[29] MIRTES, Ondřej. PHPStan - PHP Static Analysis Tool. GitHub [online]. [cit. 2021-01-24]. Dostupné z: https://github.com/phpstan/phpstan-src/blob/master/src/Rules/Constants/ConstantRule.php

[30] VOTRUBA, Tomáš. Rector - Speedup Your PHP Development. GitHub [online]. [cit. 2021-02-01]. Dostupné z: https://github.com/rectorphp/rector

[31] GRUDL, David. Dependency Injection. Nette [online]. [cit. 2021-02-27]. Dostupné z: https://doc.nette.org/cs/3.1/dependency-injection

[32] GitLab CI/CD. GitLab Docs [online]. [cit. 2021-04-03]. Dostupné z: https://docs.gitlab.com/ee/ci/

Seznam obrázků

Seznam tabulek

Seznam ukázek kódu

Příloha č. 1

Elektronická příloha obsahuje kód, který automaticky spouští nástroje zajišťující kvalitu kódu (PHPStan, Infection, Rector, PHP_CodeSniffer, PHPUnit) v rámci kontinuální integrace na platformě GitLab CI/CD. Obsah přílohy:

```
automated-testing
├── app
├── bin
├── scripts
├── tests
├── .gitlab-ci.yml
├── bootstrap.php
├── composer.json
├── composer.lock
├── infection.json.dist
├── phpstan.neon
├── phpunit.xml
└── rector.php
```

Univerzita Hradec Králové
Fakulta informatiky a managementu

Zadání diplomové práce

Autor: **Bc. Radek Meduna**

Studium: I1800328

Studijní program: N1802 Aplikovaná informatika

Studijní obor: Aplikovaná informatika

Název diplomové práce: **Automatizované testování webových aplikací psaných v PHP**

Název diplomové práce AJ: Automated testing of web applications written in PHP

Cíl, metody, literatura, předpoklady:

Cíl práce

Cílem diplomové práce je popis aktuálních trendů psaní automatických testů pro webové aplikace psané v programovacím jazice *PHP*. Práce uvede základní typy testů a doporučení pro dlouhodobé udržení kvality při vývoji webových aplikací, popíše jejich hlavní výhody a následně popíše jejich implementaci a automatizaci v rámci *continuous integration*.

Osnova

1. Úvod do testování
2. Druhy testů a programovací konvence
3. Testovací nástroje
4. Implementace testů
5. Automatizace
6. Závěr

1. ANAND, Azeem Uddin1 Abhineet. Importance of Software Testing in the Process of Software Development.
2. PAPADAKIS, Mike, et al. Mutation testing advances: an analysis and survey. In: Advances in Computers. Elsevier, 2019. p. 275-378.
3. CHEKAM, Thierry Titcheu, et al. An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption. In: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE). IEEE, 2017. p. 597-608.
4. FOURNIER, Greg. Essential software testing: a use-case approach. CRC Press, 2008.

Garantující pracoviště: Katedra informatiky a kvantitativních metod, Fakulta informatiky a managementu

Vedoucí práce: doc. Ing. Filip Malý, Ph.D.

Datum zadání závěrečné práce: 14.1.2018