Univerzita Hradec Králové
Fakulta informatiky a managementu
Katedra informačních technologií

Automatizované testování e-shopu

Diplomová práce

Autor: Bc. Ondřej Habr
Studijní obor: Aplikovaná informatika

Vedoucí práce: Mgr. Daniela Ponce, Ph.D.

Hradec Králové duben 2023

Prohlášení:

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

vlastnoruční podpis

V Hradci Králové dne 23.4.2023 Bc. Ondřej Habr

Poděkování:

Děkuji vedoucí diplomové práce Mgr. Daniele Ponce, Ph.D. za metodické vedení práce, cenné rady, připomínky a doporučení při tvorbě této práce.

Anotace

Tato diplomová práce se zaměřuje na implementaci automatizovaného testování pro vybraný e-shop vyvinutý v Nette Frameworku. Tato práce je rozdělena do dvou částí. V první části je představeno softwarové testování a různé metody testování, včetně jednotkového, integračního, funkčního, testování uživatelského rozhraní a výkonového testování. Následně je uvedeno automatizované testování, jeho strategie, metodologie a nástroje, které lze pro jednotlivé druhy testů použít. V druhé části je popsán vybraný e-shop včetně jeho architektury a požadavků na automatizace a také jsou popsány nástroje, které byly použity pro implementaci testů a jejich samotná implementace. V závěru práce je zhodnocena provedená implementace a jsou navržena její možná další rozšíření.

Annotation

Title: Automated testing of e-shop

This thesis focuses on the implementation of automated testing for a selected e-shop developed in the Nette Framework. This work is divided into two parts. The first part introduces software testing and various testing methods, including unit, integration, functional, user interface testing, and performance testing. Subsequently, automated testing, its strategies, methodologies, and tools that can be used for individual types of tests are presented. The second part describes the selected e-shop, including its architecture and requirements for automation, and describes the tools that were used for the implementation of the tests and their implementation itself. At the end of the work, the implemented implementation is evaluated, and its possible further extensions are suggested.

Obsah

Seznam obrázků

Seznam tabulek

Seznam zdrojových kódů

1 Úvod

Automatizované testování je dnes nezbytnou součástí softwarového vývoje, a to i v oblasti e-commerce. E-shopy se staly velmi populárními a dnes jsou pro mnoho firem hlavním zdrojem příjmů. Správné fungování e-shopu je proto klíčové pro úspěch podnikání.

V teoretické části této práce je přiblížena problematika, proces a cíle softwarového testování a dále jsou uvedeny jednotlivé typy softwarového testování. Poté je popsáno automatizované testování, známé strategie použití a metodologie softwarového testování. Závěrem této části jsou uvedeny výhody a nevýhody automatizovaného testování a shrnuty testy, které lze automatizovat.

Praktická část této diplomové práce je věnována implementaci automatizovaného testování pro e-shop, který byl vyvinut v Nette Frameworku. Jsou zde popsány různé nástroje pro provádění testování, včetně jednotkového, integračního, funkčního, testování uživatelského rozhraní a výkonového testování. Následně je popsán e-shop, do kterého je automatizované testování implementováno, včetně jeho architektury a požadavků co má být automatizováno. Dále jsou v práci popsány nástroje, které byly použity pro implementaci testů včetně toho, jak byly do aplikace implementovány.

2 Cíl práce

Cílem této diplomové práce je analyzovat, navrhnout a vhodně implementovat automatizované testování na vybraném e-shopu. Tedy analyzovat možnosti, co a jak lze automatizovat, nalézt vhodné nástroje pro automatizované testování a ukázat samotnou implementaci navrhovaného řešení.

3 Metodika zpracování

Analýza automatizovaného testování je provedena za pomocí literatury, která je uvedena v seznamu literatury, kdy je zkoumáno softwarové testování a co je možné testovat a jakým způsobem, kdy následně je provedena analýza automatizovaného testování a je zkoumáno jaké existují metodiky automatizovaného testování a jaké testy lze těmito způsoby automatizovat.

Vývojovým týmem zvoleného e-shopu byly zadány požadavky na automatizované testování, které má být prováděno v oblastech jednotkových a integračních testů, funkčních testů z pohledu uživatele a testů výkonnosti a stability. Pro automatizaci testování existuje velké množství nástrojů a jsou tak vybrány a podrobeny analýze pouze nástroje ve vybraných oblastech testování se zaměřením na specifikaci vybraného e-shopu. Z těchto nástrojů jsou vybrány konkrétní nástroje pro každou oblast, která má být testy pokryta, a tyto jsou podrobněji zkoumány a jsou blíže popsány jejich vlastnosti a možnosti použití.

Pro tyto nástroje je navržen způsob implementace pro vybraný e-shop, jsou stanoveny oblasti, na které lze aplikovat tyto nástroje a samotná implementace je ukázána na konkrétních příkladech, které lze použít i na další odpovídající části aplikace, ale pro svoji rozsáhlost a malý přínos nejsou v této práci zařazeny. Implementace je popsána tak, aby bylo možné využít dané postupy i pro jiné aplikace, které mají obdobnou strukturu.

Výsledky by měly ukázat přínosy automatizovaného testování a jednotlivých nástrojů, zhodnocení jejich efektivity a časové náročnosti na implementaci vůči časové úspoře po nasazení. Přínos je hodnocen mimo časové efektivity také zejména na odhalených nedostatcích a chybách, které nejsou před nasazením nástrojů známé, nalezených za použití jednotlivých nástrojů.

Cíle, kterých mělo být dosaženo, budou zhodnoceny v závěru práce vůči získaným výsledkům. Z výsledků a poznatků, které z práce vyplynou, budou uvedeny

doporučení dalšího výzkumu, který by mohl na práci navazovat z nově vzniklých otázek.

4 Testování softwaru

Testování softwaru je jako součást vývoje softwaru velmi různorodé téma. Proto tato práce obsahuje úvod do testování softwaru. Tato kapitola shrnuje, proč se software testuje, které termíny jsou důležité, co je testování softwaru, jak lze testování softwaru provádět a jak je organizováno. Dále jsou zmíněny testovací standardy a druhy testování.

Struktura kapitoly je v souladu se standardní literaturou o testování softwaru [2–7]. V literatuře neexistuje jednotné pořadí, ve kterém jsou témata uváděna, ale pracování s různými zdroji pomáhá uspořádat adekvátní přehled. Zatímco tato kapitola poskytuje ucelený obraz testování, zvláštní pozornost je věnována aspektům důležitým pro témata této práce. Kromě uvedení pozadí pro následující kapitoly může být tato kapitola použita jako stručný úvod do testování softwaru a jako reference pro testovací techniky.

4.1 Definice softwarového testování

Softwarové testování lze definovat jako proces, který se používá k ověřování, zda softwarový produkt splňuje určené požadavky a specifikace. Jedná se o proces, který se věnuje ověřování, zda produkt funguje tak, jak by měl, a zda neobsahuje žádné chyby, nedostatky nebo nekonzistence. Cílem softwarového testování je minimalizace rizik spojených s používáním softwaru a zajištění vysoké kvality produktu.

Testování softwaru spočívá v dynamickém ověřování chování programu na konečné sadě testovacích případů, vhodně vybraných z obvykle nekonečné domény provádění, proti specifikovanému očekávanému chování. Klíčovými problémy jsou při identifikaci oblasti znalostí testování softwaru zejména oblasti:

- **Dynamické**: tento termín znamená, že testování vždy znamená spuštění programu na (hodnotných) vstupech. Pro větší přesnost, samotná vstupní hodnota není vždy dostatečná k určení testu, protože komplexní,

nedeterministický systém může na stejný vstup reagovat různým chováním v závislosti na stavu systému. V následujícím textu však bude zachován termín „vstup“ s implicitní konvencí, že zahrnuje také specifikovaný vstupní stav v těch případech, kdy je to potřeba. Od testování se liší a doplňují se s ním techniky statické analýzy, jako je peer review a inspekce (které jsou někdy nesprávně označovány jako „statické testování“); tyto nejsou považovány za součást této oblasti znalostí (ani spouštění programu na symbolických vstupech nebo symbolické vyhodnocení).

- **Konečné**: i pro jednoduché programy je teoreticky možných tolik testovacích případů, že provedení vyčerpávajícího testování může vyžadovat i roky. To je důvod, proč lze v praxi celou sadu testů obecně považovat za nekonečnou. Ale počet provedení, které lze reálně pozorovat při testování, musí být samozřejmě konečný. Je zřejmé, že by mělo být provedeno „dostatečné“ testování, aby poskytlo přiměřenou jistotu. Testování ve skutečnosti vždy znamená kompromis mezi omezenými zdroji a harmonogramy a ze své podstaty neomezenými požadavky na testování: tento konflikt ukazuje na dobře známé problémy testování, jak technické povahy (kritéria pro rozhodování o přiměřenosti testu), tak manažerské povahy (odhad úsilí testovat).
- **Vybrané**: mnoho navrhovaných testovacích technik se zásadně liší ve způsobu výběru (konečné) testovací sady a testeři si musí být vědomi, že různá výběrová kritéria mohou přinést značně odlišnou účinnost. Jak určit nejvhodnější výběrové kritérium za daných podmínek je velmi složitý problém; v praxi se uplatňují techniky analýzy rizik a zkušební inženýrství.
- **Očekávané**: musí být možné (i když ne vždy snadné) rozhodnout, zda jsou pozorované výsledky provádění programu přijatelné nebo ne, jinak by úsilí o testování bylo zbytečné. Pozorované chování lze zkontrolovat podle očekávání uživatele nebo podle specifikace. Rozhodnutí o úspěšnosti/neúspěchu testu se v literatuře o testování běžně označuje jako problém orákula, který lze řešit různými přístupy, například lidskou kontrolou výsledků nebo srovnáním se stávajícím referenčním systémem. V některých situacích může být očekávané chování specifikováno pouze

částečně, to znamená, že pouze některé části skutečného chování musí být porovnány s některým uvedeným tvrzením.

Podle standardu IEEE 610.12-1990 (IEEE Standard Glossary of Software Engineering Terminology) lze testování definovat takto (IEEE Standard Glossary of Software Engineering Terminology., 1990) (IEEE Standard Glossary of Software Engineering Terminology (HyperCard Stack), 1990):

1. „Proces provozování systému nebo součásti za specifikovaných podmínek, pozorování nebo zaznamenávání výsledků a provádění hodnocení některých aspektů systému nebo součásti.“
2. „Proces analýzy softwarové položky za účelem zjištění rozdílů mezi existujícími a požadovanými podmínkami (tj. chyb) a vyhodnocení vlastností softwarových položek.“

První definice naznačuje, že nalezení selhání může být doprovázeno hodnocením. To platí například pro testování výkonu programu. Identifikace neuspokojivého výkonu je také selháním, protože systém nefunguje tak, jak je specifikováno. Myslet na testování výkonu, při snaze najít selhání, však není samozřejmé. Druhá definice obdobně uvádí, že cílem testování je nalezení rozdílů – tedy chyb – mezi požadovaným chováním a skutečným chováním částí softwaru a dále vyhodnocení, tohoto chování pro poskytnutí zpětné vazby o stavu softwaru.

4.2 Důležitost softwarového testování

Testování softwaru je důležitou fází životního cyklu vývoje softwaru. Představuje konečný přehled specifikace, designu a kódování. Hlavním cílem návrhu testovacího případu je odvodit sadu testů, které dokážou zjistit chyby v softwaru. Důkladně testovaný software udržuje kvalitu. Kvalitní produkt je to, co každý chce a ocení. Testování softwaru také poskytuje objektivní, nezávislý pohled na software, který umožňuje podniku ocenit a pochopit rizika při implementaci softwaru. Testovací techniky zahrnují – ale nejsou omezeny pouze na – proces spouštění programu nebo aplikace se záměrem najít softwarové chyby. Rozsah testování softwaru často zahrnuje zkoumání kódu i provádění tohoto kódu v různých prostředích a

podmínkách a také zkoumání aspektů kódu: dělá to, co má dělat, a dělá to, co má dělat. V současné kultuře vývoje softwaru může být testovací organizace oddělena od vývojového týmu (PEZZE & YOUNG, 2007).

Následující body shrnují hlavní důvody, proč je testování softwaru důležité (PEZZE & YOUNG, 2007) (RUBIN, 2008) (LEVENTHAL & BARNES, 2007).

- **Pomáhá šetřit peníze –** Testování softwaru má širokou škálu výhod. Nákladová efektivita projektu je jedním z hlavních důvodů, proč společnosti využívají služeb testování softwaru.
  Testování softwaru se skládá z řady projektů. V případě, že se najdou nějaké chyby v raných fázích, jejich oprava stojí menší množství peněz. Proto je nezbytným předpokladem provést testování co nejdříve. Když se najímají kvalitní analytici nebo testeři, kteří mají bohaté zkušenosti a prošli technickým vzděláním pro projekty, jsou to investice a projekt z toho bude profitovat.
- **Bezpečnost –** Je to další zásadní bod, proč by se nemělo brát testování softwaru na lehkou váhu. Je považována za nejzranitelnější a nejcitlivější část. Existuje spousta situací, kdy jsou informace a podrobnosti o uživatelích ukradeny a jsou zneužívány. To je jedním z hlavních důvodů, proč uživatelé hledají osvědčené a spolehlivé produkty.
  Jelikož konkrétní produkt prochází testováním, uživatel si může být jistý, že obdrží spolehlivý produkt a osobní údaje uživatele budou v bezpečí. Uživatelé mohou získat produkty bez zranitelnosti pomocí testování softwaru.
- **Kvalita produktu –** Sledování potřeb produktu je nezbytným předpokladem, protože pomáhá dosáhnout požadovaných výsledků. Produkty by měly uživateli sloužit za všech podmínek. Je nutné, aby to přineslo hodnotu, která byla slíbena. Proto by měl fungovat úplným způsobem pro zajištění efektivní zákaznické zkušenosti. Je také nutné zkontrolovat kompatibilitu zařízení. Například v případě, že se plánuje

spuštění aplikace, je nutné ověřit její kompatibilitu v široké řadě operačních systémů a zařízení.

- **Spokojenost zákazníka –** Prvořadým cílem vlastníka produktů je nabídnout co nejlepší spokojenost zákazníků. Důvodem, proč je nutné se rozhodnout pro testování softwaru, je skutečnost, že nabízí předpoklad pro lepší uživatelskou zkušenost. Rozhodnutí pro testování softwaru bude přinášet dlouhodobé výhody. Získání důvěry klienta jistě není snadný úkol, především v případě, že se zjistí, že produkt pokaždé nefunguje a je nestabilní. V dnešní době lze vybírat z velkého množství aplikací a první dojem je tak opravdu důležitý. Pokud se nepodaří uživatele oslnit, uživatelé si najdou jiný produkt, který bude splňovat všechny požadavky.
- **Zlepšení procesu vývoje –** S pomocí oddělení zajišťování kvality se může najít široká škála scénářů a chyb pro reprodukci chyby. Je to opravdu jednoduché a vývojáři mohou opravit chyby během pár okamžiků. Kromě toho by testeři softwaru měli pracovat paralelně s vývojovým týmem, což je užitečné pro urychlení procesu vývoje.
- **Snadné přidávání nových funkcí –** Čím je kód propojenější a starší, tím je jeho změna obtížnější. Testy působí proti tomuto sklonu ke srůstání kódu tím, že umožňují vývojářům s jistotou přidávat nové funkce. Pro nového vývojáře může být změna starších částí kódu děsivá, ale s testy bude alespoň vědět, jestli něco důležitého přestalo fungovat. To pomáhá při vytváření náskoku softwaru na trhu a poražení konkurence.
- **Zlepšení výkonu softwaru –** Pokud se pracuje se softwarem nebo aplikací, která má nízký nebo snížený výkon, zjistí se, že snižuje pověst společnosti na trhu. Podle odborníků tento bod není tak důležitý, ale v případě uvedení na trh jakéhokoli softwaru bez testování softwaru, kdy poté výkon softwaru nesplňuje očekávání nebo požadavky klientů, bude přesvědčování lidí obtížné.

4.3 Cíle softwarového testování

„Testovat program znamená snažit se, aby selhal“ (MEYER, 2008). Nalezení chyb je tedy hlavním cílem testování (CLEFF, 2010). Navzdory své okouzlující

jednoduchosti lze v definicích testování softwaru identifikovat podřízené cíle. Testování bez cílů lze považovat za ztrátu času a peněz (WALLMÜLLER, 2001). Pominou-li se nadbytečné argumenty a cíle, které se téměř rovnají hledání neúspěchů, literatura jen stěží zmiňuje cíle podřízené. Ve skutečnosti mnoho autorů cíle testování vůbec explicitně nedefinuje, ale buď je berou jako dané (například (PERRY, 2003)), nebo začleňují testování do kontextu organizační optimalizace (např. (BLACK, 2009)). Jiní autoři popisují cíle úkolů, do kterých je testování zabudováno (jako je řízení kvality (WALLMÜLLER, 2001)), nebo cíle odvozené z rolí v procesu testování (např. analytik testu (OLAN, 2003)). Kromě toho je běžné diskutovat o cílech kvality, kterých by mělo být dosaženo pomocí testování softwaru (Software Quality: Testing, analyzing and Verifying Software, 2009) (PEZZE & YOUNG, 2007). Tyto cíle zahrnují bezpečnost, použitelnost, kontinuitu, ovladatelnost, funkčnost, použitelnost, integrovatelnost, výkon a efektivitu (POL, KOOMEN, & SPILLNER).

Pokud se diskutuje o podřízených cílech, nachází se v kontextu provádění testu spíše než v kontextu motivace, proč se testování provádí. Tyto cíle jsou obvykle obecnými cíli vývoje softwaru. K jejich dosažení nebo k odstranění mezery mezi současným stavem a plněním cílů lze použít testování softwaru.

Takové cíle jsou:

- snížení nákladů na odstranění závad a chyb,
- snížení nákladů způsobených chybami (v důsledku selhání systému),
- posuzování kvality softwaru,
- dodržování norem a zákonů,
- vyhýbání se soudním sporům a problémům souvisejícím se zákazníky,
- zvýšení důvěry ve vyvíjený software nebo ve společnost, která vyvíjí softwarové produkty (CLEFF, 2010) a
- podpora co nejlepšího ladění (ZELLER, 2006).

Tyto cíle mají ekonomické pozadí. Buď se snaží snížit náklady nebo zvýšit příjmy, a to jak přímo, tak s ohledem na dlouhodobé efekty (jako je image společnosti). Z hlediska testování lze všechny cíle zahrnout do hledání selhání. Lze však namítnout, že cílem testování je zvýšit hodnotu softwaru. Odstranění závad (a aby toho bylo možné, nalezení chyb) je nástrojem k dosažení vyšší kvality a tím i hodnoty (MYERS & SANDLER, 2004). Zvyšující se hodnota musí být udržována ve vztahu k testovacímu úsilí, aby byl cíl zachován. Kromě cílů testování z vnější perspektivy by mělo mít testování jako úkol při vývoji softwaru také své cíle. Tyto cíle by měly být „objektivně měřitelné“ (WALLMÜLLER, 2001) a obvykle se snaží dosáhnout kvantitativních kritérií, jako je pokrytí nebo úprava číselných údajů o složitosti.

4.4 Proces softwarového testování

Softwarové testování je klíčovou částí vývojového cyklu softwaru. Jeho hlavním cílem je ověřit, že software splňuje specifikace a požadavky a že funguje správně. Proces testování je obvykle rozdělen do několika fází, které se liší podle stupně detailu a složitosti testování. V této kapitole jsou uvedeny základní fáze procesu softwarového testování.

Plánování testování

Prvním krokem v procesu softwarového testování je plánování testování. Během této fáze určí testovací tým, jaké testy budou provedeny, jak se budou provádět a kdy budou prováděny. Plánování testování může být náročné, zejména pokud se jedná o velký projekt. Musí se zvážit, jaké funkce softwaru se budou testovat, jaké bude mít testování cíle a jakým způsobem se testování bude provádět (MEYER, 2008).

Návrh testování

Po plánování testování se tým zabývá návrhem testů. To znamená, že se rozhodne, jaké testovací scénáře a případy se budou používat k ověření, zda software funguje správně. Tento krok může zahrnovat vytvoření testovacích dat, testovacích procedur a dalších dokumentů (LEVENTHAL & BARNES, 2007).

Implementace testování

Implementace testování je proces, během kterého se testovací scénáře a případy provádějí. Tým provádí testování v souladu s plánem a dokumentací testování. V této fázi se také mohou objevit chyby, které je nutné zaznamenat a opravit (CARTER, 2018).

Hodnocení výsledků testování

Po dokončení testování se tým zabývá hodnocením výsledků. Zde se analyzují výsledky testování, aby se zjistilo, zda software splňuje specifikace a požadavky. Pokud byly nalezeny chyby, tým je zaznamená a popíše, jak je opravit (SEARS & JACKO, 2008).

Oprava chyb

V této fázi se tým soustředí na opravu nalezených chyb. Tým zajišťuje, že všechny chyby byly odstraněny a software funguje tak, jak by měl.

Opakování testování

Po opravě chyb je třeba testování opakovat, aby se ověřilo, že chyby byly opraveny a software pracuje, jak má.

5 Typy softwarových testů

5.1 Statické testování

Statická analýza nespouští uvažovaný program a nevyžaduje nutně nástrojovou podporu (LIGGESMEYER, 2009). Kód je analyzován a interpretován (SNEED & WINTER, 2002). Důrazně se však doporučuje použití nástroje, protože techniky statické analýzy jsou vhodné pro automatizaci. Zahájení statických analýz je možné v raném bodě vývoje. Obecně však nejsou nalezeny žádné vady a chyby, ale pouze jejich náznaky.

Analýza stylu (neformálně známá také jako kontrola programových pokynů (MEYER, 2008)) je technika, která kontroluje, zda zdrojový kód vyhovuje předem definovaným konvencím. Tyto konvence mohou být směrnicemi pro celou společnost i osvědčenými postupy pro programovací jazyky (LIGGESMEYER, 2009). Díky dostupnosti zásuvných modulů lze analýzu stylu bez problémů integrovat do integrovaného vývojového prostředí (IDE) (MAJCHRZAK, Improving the technical aspects of software testing in enterprises, 2010). Analýza stylu je zvláště doporučena pro vestavěné systémy, ale také obecně doporučena (PEZZE & YOUNG, 2007); navíc může být dobrou volbou udržovat tzv. legacy systémy (WALLMÜLLER, 2001). Dá se očekávat, že programátoři si po chvíli zvyknou na povinný styl programování. Kromě vytvoření podnikové směrnice pro programování, která harmonizuje styly programování (MAJCHRZAK, Improving the technical aspects of software testing in enterprises, 2010), by se měly dodržovat styly poskytované vývojáři programovacích jazyků (např. (Oracle, nedatováno)) a v učebnicích pro odborníky (např. (SUTTER & ALEXANDRESCU, 2004)). Obsáhlý úvod podává Kellerwessel (KELLERWESSEL, 2002). Zde je uvedeno, že stylová analýza by měla být kombinována s používáním moderního programovacího jazyka, který – pokud je správně používán – snižuje riziko programování ve špatném stylu (WALLMÜLLER, 2001).

Navzdory skutečnosti, že dělení (slicing) lze provádět jak staticky (WEISER, 1981), tak dynamicky (KOREL & LASKI, 1988), je připisováno technikám statické analýzy

(SUTTER & ALEXANDRESCU, 2004). Dělení se používá k automatickému zjišťování vzájemných vztahů v rámci programů, například s ohledem na vliv příkazů na proměnné. To pomáhá usoudit, která část nebo dokonce který příkaz programu způsobuje nežádoucí akci. Lze identifikovat vzory defektů (ZELLER, 2006). Zatímco statické dělení nevyžaduje provádění programu, dynamické dělení bere v úvahu informace, které jsou dostupné pouze během provádění (POL, KOOMEN, & SPILLNER). Dělení lze použít pro lepší pochopení programu a pro podporu hledání defektů (SEARS & JACKO, 2008). Existují různé strategie dělení (ZELLER, 2006); přehled starších a současných výzkumů je uveden v (XU, QIAN, ZHANG, WU, & CHEN, 2005).

Analýza řídicího toku staticky kontroluje kód, aby odhalila anomálie. Cílem je odhalit kód, který nelze ze sémantických důvodů provést (mrtvý kód) a cykly, které nelze po zadání opustit (nekonečný cyklus – endless loop) (BLACK, 2009). Grafy volání posouvají tento koncept na vyšší úroveň. Dokumentují, jak jsou moduly mezi sebou propojeny a pomáhají plánovat testy (BLACK, 2009).

Analýza toku dat (anomálie) – nebo detekce anomálií toku dat – se používá k detekci částí programů, které se odchylují od očekávání. Podobně jako práce kompilátoru se snaží detekovat zvláštní části programů; teoreticky by tyto části mohly být identifikovány i pečlivě pracujícími lidmi (LIGGESMEYER, 2009). Typickým pokusem je zkontrolovat proměnné, zda jsou definovány, odkazovány (tj. čteny) a dereferencovány (tj. nastaveny jako nedefinované). Po kontrole programu lze detekovat anomálie, jako je předefinování proměnných bez jejich čtení, jejich definování a okamžité dereferencování nebo dereferencování a následné pokusy o načtení proměnné (CLEFF, 2010). Důrazně se doporučuje podpora kompilátoru pro tento druh analýzy; explicitní vyvolání analýzy datového toku se uvádí jako poměrně neobvyklé (JAN, NGUYEN, & BRIAND, 2016).

Kontrolní tok a analýzu toku dat lze kombinovat s nástroji, které automaticky skenují kód na anomálie. Mohou být založeny na různých technikách hodnocení a mohou také nabízet vizuální podporu (ZELLER, 2006).

Inspekce softwaru a recenze kódu jsou rozsáhlou oblastí s řadou technik. Je známá již od začátků testování (FAGAN, 1976). Obecnou myšlenkou je ruční posouzení zdrojového kódu. Nehledají se pouze vady, ale recenzenti posuzují, zda bylo dosaženo požadované úrovně kvality a zda jsou splněny konvence (WALLMÜLLER, 2001). Dobře provedené revize jsou velmi nákladově efektivní a významně přispívají k celkovému zajištění kvality softwaru (CLEFF, 2010). Liggesmeyer rozlišuje mezi třemi hlavními technikami (LIGGESMEYER, 2009): komentování recenzí, neformální revizní sezení a formální kontroly. Komentování recenzí je nejjednodušší technika; programy, moduly nebo dokumentace jsou přiděleny recenzentům, kteří je kontrolují a komentují. Neformální recenzní schůzky jsou běžně organizovány jako strukturovaný návod. Vývojáři ukazují svou práci recenzentům, kteří je přímo komentují. Formální kontroly jsou nejsofistikovanější technikou. Nejsou pouhými návody, ale mají formálně definované fáze a role. Jsou vysoce strukturované, což zvyšuje úsilí potřebné k jejich vedení, ale výrazně zvyšuje výsledek.

Úspěch recenze je založen na účastnících a jejich souhře (WALLMÜLLER, 2001). Typicky je počet účastníků na recenzi mezi čtyřmi a deseti; účastníci zastávají různé role a obvykle je jedním z nich moderátor (SPILLNER, ROSNER, WINTER, & LINZ, 2008). Lze tvrdit, že recenze jsou přínosem pro pracovní atmosféru a přispívají ke vzdělávání zaměstnanců (PEZZE & YOUNG, 2007). Firemní kultura však musí být pro recenze adekvátní (BÉRARD, a další, 2001).

Podrobnější podtřídy pro revize navrhují např. Bath a McKay, kteří (mimo jiné) zmiňují technické revize, revize managementu, audity a revize designu (BATH & MCKAY, 2010). Audity jsou obvykle méně technicky zaměřené než revize (WALLMÜLLER, 2001). Uvádí se zde, že revize mohou být efektivnější při odstraňování defektů než (dynamické) testování (BLACK, 2009) a vést k okamžitým vylepšením kódu (WALLMÜLLER, 2001). Podle Liggesmeyera jsou recenze kódu široce používány odborníky z praxe (LIGGESMEYER, 2009).

Technikou, která je podobná recenzím a kterou lze s recenzemi kombinovat, je identifikace tzv. anti vzorů (KOENIG, 1995). Zatímco návrhové vzory popisují osvědčené strategie pro implementaci opakujících se funkcí softwaru (GAMMA, HELM, JOHNSON, & VLISSIDES, 1995), anti vzory popisují, jak funkcionalitu neimplementovat (BROWN, MALVEAU, MCCORMICK, & MOWBRAY, 1998). Anti vzory se také označují jako „zápachy" kódu, tj. části zdrojového kódu, které mají „špatný zápach" kvůli potenciálním problémům (FOWLER, 1999). Odpovídající konstrukce lze nalézt jak ručně, tak s podporou nástrojů. Kód by pak měl být refaktorován, tedy změněn na lepší styl při zachování jeho funkčnosti (OPDYKE & JOHNSON, 1990). Ke změně kódu, aby se držel vzorů, se doporučuje použít refaktoring (KERIEVSKY, 2004). Některé „pachy" kódu lze také nalézt výše popsanými technikami. Mrtvý kód lze například identifikovat analýzou řídicího toku.

Metriky popisují klíčové údaje, které jsou generovány ze zdrojového kódu, zprostředkujícího kódu, programů, dokumentace programu nebo systémů, které spravují data související s testováním. Metriky agregují data a umožňují vyvozovat závěry nebo odhadovat trvání nebo časové požadavky. Používají se k posouzení kvality programu, ke zvládnutí složitosti, ke kontrole procesu vývoje a ke kontrole, zda jsou splněny standardy (LIGGESMEYER, 2009). Dále jsou užitečné k odhadu úsilí, nákladů a trvání, k proaktivně identifikovat problémy z důvodu srovnání a posoudit efektivitu nově zaváděných nástrojů a metod (LIGGESMEYER, 2009). Mohou také pomoci najít problematické oblasti v kódu (BATH & MCKAY, 2010). Kód, který se stává příliš složitým kvůli vnořeným příkazům, metodám, které jsou velmi dlouhé, a podobným kritériím naznačují potenciální problémy.

Některé metriky jsou také důležité pro fungování testovacích nástrojů. Například lze vypočítat procento pokrytých hran grafu řídicích toků a řetězců def-use (MAJCHRZAK & KUCHEN, Automated test case generation based on coverage analysis, 2009). Metriky obecně však mají při testování pouze podřízenou (nebo spíše pomocnou) roli. Proto není uveden žádný úvod do různých druhů metrik a

jejich použití. Viz (LIGGESMEYER, 2009) nebo (SPILLNER, ROSNER, WINTER, & LINZ, 2008).

Kromě výše uvedených technik lze ruční kontrolu diagramů považovat za statickou techniku (LIGGESMEYER, 2009). To platí např. pro grafy regulačních toků, Nassi–Shneidermannovy diagramy (NASSI & SHNEIDERMAN, 1973) a strukturní diagramy.

5.2 Dynamické testování

Dynamické techniky jsou založeny na spouštění kódu za účelem detekce selhání. Obsahují velké množství technik, které lze rozdělit do šesti kategorií. Popisy jsou uvedeny v následujících částech.

5.2.1 Strukturně orientované

Orientace na strukturu znamená, že program je testován proti vlastnímu kódu (SNEED & WINTER, 2002). Nutí testery podrobně kontrolovat kód. Úsilí a požadovaná zručnost zaměstnanců pro strukturně orientované techniky je vysoká, ale je možné efektivně odhalit nedostatky (BATH & MCKAY, 2010). Hlavní kritikou je, že samotná struktura může být nesprávná, například v důsledku nesprávných požadavků. Zatímco jiné techniky by to mohly alespoň pomoci odhalit, nelze to na základě struktury detekovat. Rady ohledně výběru funkčně orientovaných technik poskytují Bath a McKay (BATH & MCKAY, 2010).

Existuje řada strukturně orientovaných technik, které jsou velmi jednoduché. Vzhledem k tomu, že jsou ve výsledcích horší než testování kontrolního průtoku, nebudou dále podrobněji rozebrány. Pro testy příkazů se vytvoří tolik testovacích případů, kolik je potřeba k pokrytí každého příkazu (zdrojového kódu) alespoň jednou. Oborové testy vyžadují, aby každá větev – tedy každý přechod z příkazu, který zahrnuje rozhodnutí – byla pokryta alespoň jednou. Různé druhy testů podmínek vyžadují pokrytí atomických podmínek nebo jejich kombinací (BATH & MCKAY, 2010). Testy podmínek, které jsou odvozeny ze zdrojového kódu, mohou také vzít v úvahu vyhodnocení příkazů kompilátory, aby se snížil počet testovacích případů (CLEFF, 2010). Například ve většině objektově orientovaných

programovacích jazyků se bitové (operátor |) podmínky vyhodnocují úplně, zatímco logika (operátor ||) nevyžaduje úplné vyhodnocení, pokud je první operand pravdivý.

Testování řídicího toku se snaží pokrýt každou cestu programem. Pro tento účel je sestaven graf řídicího toku (CFG) (LIGGESMEYER, 2009). Nastavuje se kontrolou každého příkazu programu ve zdrojovém kódu nebo instrukci ve stroji nebo „bajtkódu“. Každý příkaz/instrukce tvoří uzel v grafu, zatímco přechody tvoří hrany. Jednoduchý příkaz jako x = 2 má pouze přechod na další příkaz; podmínky mají dva (např. pokud (…)) nebo více (např. přepínač (…)) přechodů. Technika vytváření CFG pro Javu je popsána v (MAJCHRZAK & KUCHEN, Automated test case generation based on coverage analysis, 2009). S daným CFG se vytvářejí testovací případy, které pokrývají každý přechod alespoň jednou. Soubor testovacích případů se snaží být co nejmenší. Alternativně lze definovat práh pro požadovanou míru pokrytí (BATH & MCKAY, 2010). Rozumného prahu lze často dosáhnout s mnohem menším počtem testovacích případů, než bylo potřeba pro plné pokrytí. To je způsobeno testovacími případy, které pokrývají více cest programem, zatímco některé přechody vyžadují přizpůsobené testovací případy, aby je pokryly. Volba prahu může být dokonce nutností. Kvůli sémantickým omezením nemusí být všechny hrany pokrytelné (PEZZE & YOUNG, 2007).

Testování řídicích toků je možné dále rozdělit, např. na testování cesty a testování cyklu. Poslední zmíněné je také známé jako lineární kódová sekvence a skok (LCSAJ) (HENNELL, WOODWARD, & HEDLEY, 1976) nebo cesta skoku ke skoku (JJ-cesta) (WOODWARD & HENNELL, 2006). To se však v současné literatuře nedělá. Některé méně známé techniky jsou vysvětleny v (ROITZSCH, 2005); různá kritéria pokrytí jsou diskutována v (THALLER, 2002). Srovnání technik založených na řízení toku je uvedeno v (PEZZE & YOUNG, 2007). Někdy se testování cesty používá také jako synonymum pro testování řídicího toku (např. v (BATH & MCKAY, 2010)).

Testování toku dat využívá informace o tom, jak program zpracovává data. Za tímto účelem se kontroluje při čtení a zápisu proměnných (LIGGESMEYER, 2009). To lze

provést na úrovni zdrojového kódu, ale také pro zprostředkující kód nebo strojový kód. Pro zprostředkující nebo strojový kód, který je spuštěn na zásobníku, se stává složitějším, protože načtení proměnné do zásobníku neznamená, že je čtena.

5.2.2 Funkčně orientované

Při testování funkcí programu se berou v úvahu jeho příčiny a následky. Zatímco strukturně orientované techniky se zaměřují na samotný program, funkčně orientované techniky mají pohled na vyšší úrovni (SNEED & WINTER, 2002). Testeři čtou dokumenty se specifikacemi a navrhují odpovídající testovací případy (LIGGESMEYER, 2009). Funkční orientaci lze proto také nazvat orientací specifikace (BATH & MCKAY, 2010). Testovací případy jsou odvozeny spíše systematickým způsobem než zvolením přístupu hrubé síly (PEZZE & YOUNG, 2007).

Testování pomocí funkčně orientovaných technik nemusí nutně vést k úplnému pokrytí testováním, ale pomáhá zajistit, aby byly splněny specifikace (BATH & MCKAY, 2010). Funkčně orientované techniky sice nejsou primárním cílem, ale také mohou odhalit nekonzistence nebo dokonce chyby ve specifikaci (CLEFF, 2010). Obecně platí, že funkčně orientované testy modulů by neměli provádět vývojáři, kteří je implementovali (THALLER, 2002). Rady ohledně výběru funkčně orientovaných technik a diskuzi o jejich praktickém významu poskytují Bath a McKay (BATH & MCKAY, 2010).

Nejjednodušší funkčně orientovanou technikou kromě náhodného testování je testování funkčního pokrytí. Snaha je pokrýt každou funkci jedním testovacím případem (ROITZSCH, 2005). Funkční pokrytí je nejen neefektivní, ale také čelí vážným omezením při testování programů pro objektovou orientaci (ROITZSCH, 2005).

Takzvané případy užití popisují scénáře, ve kterých se program skutečně používá (BATH & MCKAY, 2010). Používají se při analýze a specifikaci požadavků. Některé specifikace již obsahují připravené případy užití (CLEFF, 2010). Unified Modeling

Language (UML) poskytuje diagramy případů použití pro vizualizaci (BOOCH, REMBAUGH, & JACOBSON, 2005). Každý případ použití se skládá z několika cest programem. Pro každý případ použití musí být vytvořeno tolik testovacích případů, kolik je potřeba k pokrytí každé odpovídající cesty alespoň jednou (BATH & MCKAY, 2010). Může být také organizováno jako testování vláken, kde testeři hodnotí funkčnost v pořadí, které by si koncoví uživatelé obvykle vybrali (WATKINS, 2001). Use case testování je orientováno na zákazníka, ale je velmi pravděpodobné, že některé testy jsou vynechány (BATH & MCKAY, 2010). Funkčnost nebude testována, pokud se použití programu liší od záměru vývojářů.

Rozhodovací tabulky jsou speciální formy kombinatorických testů. Jsou užitečné pro popis obchodních pravidel (BLACK, 2009). Rozhodovací tabulky shrnují pravidla v přehledných tabulkách. Každé pravidlo se skládá z podmínek a akcí. Každá kombinace podmínek spouští řadu akcí. Podmínky obvykle nabývají hodnot „platí“, „neplatí“ nebo „irelevantní“ (MAJCHRZAK, Improving the technical aspects of software testing in enterprises, 2010). K akcím může dojít (nebo nemusí). Je potřeba jeden test na sloupec tabulky (BLACK, 2009). Je však možné plné pokrytí pravidel. V důsledku toho je snadné testovat s danou rozhodovací tabulkou a tato technika je velmi oblíbená; vytvoření tabulky je však náročné nebo dokonce téměř nemožné, pokud jsou k dispozici pouze nepřesné požadavky na program (BATH & MCKAY, 2010). Kromě toho tabulky exponenciálně rostou v počtu podmínek (LIGGESMEYER, 2009). Rozhodovací tabulky se používají pro analýzy příčin a následků, které se snaží snížit počet požadovaných testovacích případů (CLEFF, 2010). Pomáhá také graficky znázornit graf příčin a následků (ROITZSCH, 2005), který je sestaven na základě specifikace programu (WALLMÜLLER, 2001).

Rozdělení ekvivalence (ve starší literatuře také nazývané vstupní testování (ALPER, 1994)) je velmi běžný přístup. Je téměř univerzálně použitelný (LIGGESMEYER, 2009). Namísto náhodného testování se testovací případy zarovnají s třídami, které mají být testovány. To se děje jak v makro, tak v mikro perspektivě (BLACK, 2009). Například části programu jsou vybírány pro testování na základě jejich vlastností. V modulu programu jsou vytvořeny třídy ekvivalence, které přímo oslovují jeho

prvky. Existují dva druhy tříd ekvivalence: platné a neplatné. To si lze představit jako textové pole, které má hodnoty pro celé hodiny 24hodinového času. −1 a 31 jsou příklady neplatných tříd. 0−23 je třída platných hodnot. Pro testování se bere jedna reprezentativní hodnota z každé třídy (CLEFF, 2010). Testování lze provést třemi testovacími případy pro −1, 5 a 28. Samozřejmě jsou možné i jiné hodnoty. Rozdělení ekvivalence je účinná technika, ale náchylná k vynechání testů (BATH & MCKAY, 2010). To je obvykle způsobeno náhodným vytvořením tříd, které zahrnují více než jednu třídu. Je třeba také poznamenat, že třídy ekvivalence obecně nejsou disjunktní (PEZZE & YOUNG, 2007).

Použití číselných hodnot (zejména intervalů) je někdy popisováno jako analýza okrajových hodnot, zatímco rozdělení ekvivalence používá pouze textové hodnoty nebo hodnoty vlastností (BLACK, 2009). Bylo zjištěno, že hraniční hodnoty souvisí s poruchami (THALLER, 2002). Pro nastavení hraničních hodnot musí být třídy předem rozděleny (BATH & MCKAY, 2010). Rozdělení ekvivalence pomocí alfanumerických hodnot lze také nazvat testování syntaxe (ALPER, 1994).

Doporučuje se testovat speciální hodnoty spolu s analýzou hraničních hodnot (ROITZSCH, 2005). Hodnoty jako nula, maximální hodnota proměnné, prázdné datové struktury, nulové odkazy, takzvaná magická čísla a vyhrazené příkazy mohou spustit speciální rutiny nebo vést k dalším problémům. Seznam typických speciálních hodnot je uveden například v (ROITZSCH, 2005).

Vzhledem k vysokému počtu kombinací ve velkých tabulkách byly nalezeny přístupy, které zvyšují efektivitu. Testování domén se používá ke kontrole hranic tříd ekvivalence a ke snížení počtu testovacích případů potřebných k testování všech (zbývajících) tříd. Používají nejen třídy funkční ekvivalence, které byly odvozeny ze specifikace, ale také třídy strukturní ekvivalence (ROITZSCH, 2005). Několik příkladů pro testování domén uvádí Black (BLACK, 2009) a Roitzsch (ROITZSCH, 2005); příklady jsou testy domény cesty a testy kódu, který naznačuje defekty. Další techniky pro stejný účel jsou ortogonální pole a tabulky všech párů (BLACK, 2009). Obojí znamená snížit počet testovacích případů potřebných k

uspokojivému testování s třídami ekvivalence. Obecně to lze provést s úspěchem, ale existuje riziko vynechání požadovaných testovacích případů (BATH & MCKAY, 2010). Grafickou podporu pro tento úkol zajišťují kresby klasifikačních stromů. Vizualizují párování, které může zvýšit přehlednost – nebo alternativně vést k obrovským stromům s opačným efektem (BATH & MCKAY, 2010).

Dalším přístupem ke kombinatorickému testování je vzít rozdělení ekvivalence jako základní myšlenku a odvodit z něj konkrétní techniky. Rozdělení kategorií, jak je popsáno Pezzè a Youngem (PEZZE & YOUNG, 2007), v zásadě připomíná výše uvedený popis rozdělení ekvivalence. Navrhují také techniky pro optimalizaci, jako je párové testování (také známé jako všechny páry) a testování založené na katalogu (PEZZE & YOUNG, 2007). Proto je třeba s ohledem na literaturu poznamenat, že kombinatorické techniky jsou představeny v různých rozlišeních a s různými názvy.

Partition Analysis (analýza oddílů) je přístup, který kombinuje různé myšlenky, konkrétně ověřování a testování (RICHARDSON & CLARKE, 1985). Lze jej zařadit do různých kategorií (LIGGESMEYER, 2009). Analýza oddílů používá symbolické provádění a definuje tři kritéria pro porovnání specifikace a implementace programu: kompatibilitu, ekvivalenci a izomorfismus. Pokud jejich ověření částečně selže, provedou se dodatečné zkoušky (LIGGESMEYER, 2009). Více informací je uvedeno v (RICHARDSON & CLARKE, 1985); novější přístup je popsán v (PODGURSKI & YANG, 1993).

Technika, která převádí myšlenku strukturovaného testování na testy černé skříňky, je stavově orientované testování. Někdy je to připisováno technikám založeným na modelu (PEZZE & YOUNG, 2007). Bez znalosti kódu programu se zjistí jeho možné stavy. Cílem testování je pokrýt každý přechod mezi stavy alespoň jednou. To je snadné a efektivní pro programy, které mají známé stavy a malý počet přechodů (PEZZE & YOUNG, 2007). Na rozdíl od jiných funkčních testů se bere v úvahu paměť programu (tedy jeho stav) (LIGGESMEYER, 2009). Pokud lze program popsat jako konečný stroj, je k dispozici podpora nástrojů (CLEFF, 2010). Manuální úsilí však nelze zcela obejít (PEZZE & YOUNG, 2007).

Stavově orientované testování nelze použít, pokud je neznámý počet stavů nebo pokud je třeba pokrýt vysoký počet (řekněme alespoň několik stovek) přechodů. Exploze stavového prostoru se rychle stává problematickou pro lidské testery, ale ani stroje nedokážou udržet kontrolu nad stavy, která obecně roste exponenciálně s velikostí programu (LIGGESMEYER, 2009). Proto se používá především k testování softwaru pro vestavěné systémy (BATH & MCKAY, 2010). Na základě testů se stavovými automaty byly vyvinuty statistické stavově orientované testy. Berou v úvahu pravděpodobnost, že je funkce použita, a snaží se nejprve najít ty defekty, které se s větší pravděpodobností vyskytnou při skutečném použití programu (ROITZSCH, 2005).

Není možné pouze testovat funkčně orientovaným způsobem, ale testovat funkce programu. Toto se nazývá funkční testování; tento termín je však potenciálně zavádějící. Technika má za cíl ověřit, zda software dodržuje svou specifikaci; tento úkol se provádí zcela ručně. Kromě toho lze tvrdit, že program je spíše kontrolován, aby se zjistilo, zda vyhovuje požadavkům, místo aby byl testován. Typickými technikami jsou kontrola správnosti, kontrola souměřitelnosti a testování interoperability; podrobný úvod je uveden v (BATH & MCKAY, 2010).

5.2.3 Dynamická analýza

Techniky dynamické analýzy mohou odhalit vady, které je téměř nemožné najít jinými technikami. Druh selhání, se kterým se lze setkat, může být kritický a zároveň obtížně reprodukovatelný. Problémy se zdroji pravděpodobně nastanou po dokončení softwaru. Oprava takových defektů je velmi nákladná. Kromě hledání defektů lze dynamickou analýzu použít k vizualizaci výkonu programu a k odvození informací za běhu. Dynamická analýza může zpomalit provádění a je závislá na použitém programovacím jazyce (BATH & MCKAY, 2010).

Úniky paměti popisují stav, kdy programy nepřetržitě přidělují více paměti, než potřebují. Ve většině případů je paměť správně alokována, ale není zcela uvolněna poté, co již není potřeba. V závislosti na programu mohou být úniky paměti nepříjemné až fatální. V některých případech nejsou zaznamenány, v jiných

zpomalují provádění nebo dokonce vedou ke zhroucení prováděného programu. Je obtížné je reprodukovat, protože může trvat dlouho (v extrémních případech měsíce), než budou mít následky, a obvykle je těžké je napravit. Jsou běžnější v programech, které byly vyvinuty pomocí programovacích jazyků, které nepoužívají garbage collection. Podpora nástroje je vysoce doporučena pro detekci úniků paměti (MARIANI, HAO, SUBRAMANYAN, & ZHU, 2017).

Existují další techniky dynamické analýzy, které se snaží odhalit problémy programování, které nelze detekovat staticky. Příkladem poměrně často popisované techniky je detekce „divokých" ukazatelů. Říká se jim také visící ukazatele. Ukazatele naznačují adresy paměti, které se používají ke čtení nebo ukládání dat nebo jako adresy skoků. Nesprávné použití ukazatele a aritmetika ukazatele může způsobit různé problémy. Nalezení (a odstranění) divokých ukazatelů se může některým z nich vyhnout. Dalším příkladem je lockset analýza, která se snaží odhalit závodní podmínky. Závodní podmínky mohou nastat, když programy s více vlákny používají sdílené proměnné; je těžké je odhalit a bývají nedeterministické (FOWLER, 1999).

5.2.4 Testování výkonu

Výkonnostní testy lze také nazvat testy účinnosti. Používají se k posouzení chování za běhu s ohledem na využití zdrojů a doby odezvy. Přiměřený výkon není pouze faktorem použitelnosti, ale je důležitý i pro celkovou kvalitu programu. Provádějí se tak výkonnostní testy, aby se zajistilo, že program bude fungovat za nepředvídatelných budoucích podmínek. Existují dvě hlavní testovací techniky a řada dalších technik, které spíše měří, než testují (ROITZSCH, 2005).

Zátěžové testy se používají k posouzení chování programu při zatížení. Zátěž, tj. součet vstupů do programu, se podobá typické zátěži, která se pro program odhaduje. Mezi cíle zátěžových zkoušek patří následující (BÉRARD, a další, 2001).

- Zkontrolovat, jak programy zpracovávají paralelní požadavky a kolik požadavků mohou paralelně zpracovat,

- zjistit, jak funguje správa relací programu, tj. jak je řešeno bezproblémové paralelní použití,
- k posouzení doby odezvy systému za různých podmínek,
- sledování využití zdrojů ve vztahu k zatížení programu.

Zátěžové testy lze také nazvat objemovými. Obecně by se zátěžové testy měly odlišovat od benchmarků, které měří výkon programu a snaží se generovat klíčové údaje (THALLER, 2002).

Zátěžové testy mají jiný cíl. Namísto použití typických zátěží jsou programy testovány zátěží, která přesahuje očekávanou zátěž. Ve skutečnosti je zatížení dokonce nad maximem uvedeným ve specifikaci, aby bylo možné pochopit, jak se program chová při takovém zatížení. Obecně by se programy měly chovat elegantně. Namísto náhlého selhání by měl být program schopen uložit otevřené soubory, zachovat relace a zajistit, aby nebyla poškozena žádná data. V ideálním případě by se měl program přepnout do režimu minimálního využití zdrojů. Pokud je například zaplaven požadavky, program by se je mohl snažit přijímat co nejdéle a odpovídat na ně, jakmile se počet příchozích požadavků sníží. Ve webovém prostředí však může být tato strategie fatální; požadavky, které nebudou okamžitě zodpovězeny, nebudou – kvůli krátkým časovým limitům – stejně přijaty a klienti budou posílat další požadavky. Proto by v tomto případě měly být nezodpověditelné požadavky zahozeny, dokud se zatížení nesníží. Na rozdíl od toho je pád nežádoucí, protože restartování programu bude chvíli trvat. Pokud je havárie nevyhnutelná, mělo by jít o ladnou degradaci, tedy nouzové vypnutí, a ne zastavení v náhodném bodě (BROWN, MALVEAU, MCCORMICK, & MOWBRAY, 1998).

Dále lze zátěžové testy použít k nalezení úzkých míst v programech. To platí zejména pro programy, které jsou součástí větších systémů. Například systém, který poskytuje data ze souborů na vyžádání, pravděpodobně stěží využije CPU a paměť, ale vytvoří velké I/O zatížení na velkokapacitních paměťových systémech. Pokud přijde množství požadavků, které je těžké zpracovat, neměl by se snažit doručit všechny soubory najednou, což by snížilo výkon velkokapacitního úložiště, a

nakonec vedlo k propustnosti blízké nule. Místo toho by měl využívat velké množství paměti a ukládat požadavky. Mohou být zpracovány, jakmile je I/O zatížení nižší. Kromě toho mohou zátěžové testy naznačit implementaci mezipaměti pro často požadované soubory (KELLERWESSEL, 2002).

Zátěžové testy mohou dodržovat různé strategie. Testy odrazu se liší mezi nominálním a mimořádným zatížením, aby se zjistilo, zda program přizpůsobuje své využití zdrojů. Tvrdilo se, že zátěžové testy mohou být také testy, které nevyvolávají vysokou zátěž, ale pouze porušují kvantitativní specifikaci programu. Příkladem je testování programu, který by měl přijmout 16 paralelních uživatelů se 17 uživateli. Jedná se však spíše o funkční než zátěžový test (BLACK, 2009).

Řada dalších technik je založena spíše na hodnocení výkonu programu než jeho testování. Testy škálovatelnosti se nazývají testy, ale jejich cílem je zjistit, zda se systémy škálují. Škálování znamená přizpůsobení se změněným vzorům používání. Škálovatelný program by se mohl jednoduše přizpůsobit zvýšené pracovní zátěži, mohl by provádět více požadavků na upgradovaném hardwaru nebo být připraven na vytvoření více instancí na různých počítačích. To se kontroluje s ohledem na odhadovaný budoucí požadavek. Kontroly využití zdrojů a měření efektivity se používají ke sledování výkonu programu a k odvození klíčových údajů z jeho provádění. Klíčovými údaji mohou být doby odezvy a také údaje o propustnosti a podobná data. Pomáhají naučit se, jak program funguje a které jeho části by mohly vyžadovat zlepšení. Jedním z problémů, který se týká zejména monitorování programů, je vyhnout se zkreslení vyvolanému jejich skutečným monitorováním. Výsledky měření mohou být značně změněny samotným aktem měření (HENNELL, WOODWARD, & HEDLEY, 1976).

Kromě testování výkonu je možné zkontrolovat spolehlivost programu. Tyto testy spolehlivosti zahrnují prvky ze zátěžových testů a ručního hodnocení. Jsou používány k tomu, zda program splní zamýšlené úkoly bez rizika výpadků nebo nežádoucího chování. Zatímco testy výkonu se zabývají skutečným výkonem, testy spolehlivosti, například kontrolují, zda rutiny pro přepnutí při selhání fungují

správně, zálohy se vytvářejí podle plánu a je možná obnova dat. Posledně jmenovaný může být také nazýván testem zotavení. Spolehlivost lze hodnotit jak z hlediska funkčnosti, tak z hlediska časových údajů. Pokud jde o funkci převzetí služeb při selhání, lze zkontrolovat, zda je program odolný vůči selhání. V ideálním případě by chyby neměly narušovat základní funkčnost, ale měly by být programem zpracovány co nejlépe. Běžně používanými metodami jsou analýza režimu a účinků poruchy (FMEA) (GEORGIEVA, 2010) a analýza režimu a účinků a kritičnosti poruchy (FMECA) (NEUFELDER, 1992).

6 Automatizované testování

Co je automatizace testování? Automatizace testování jakékoli softwarové aplikace má sekvenci činností, procesů a nástrojů, které budou zpracovány za účelem spuštění testu na softwaru. Výsledky těchto běhů lze zaznamenat a uložit. Následující aktivity popisují proces testování automatizace:

- Testovací plán
- Návrh nebo implementace testu
- Provádění testů
- Vyhodnocení a analýza testů.

Automatizace testování může být 2 typů (SNEHA & MALLE, 2017):
Testování založené na kódu – využívá již existující rozhraní, knihovny, třídy a moduly k testování s *n* počtem vstupů a může ověřit a validovat, zda jsou výsledky správné nebo ne.

Testování založené na grafickém uživatelském rozhraní (GUI) – Stisky kláves a kliknutí myší v rozhraní mohou být simulovány a používány frameworkem k detekci změn a k ověření, zda je výkon programu správný nebo ne. Bez automatizovaného testování by ruční testování vyžadovalo přísnou práci a spoustu práce.

6.1 Strategie softwarového testování

Strategii testování lze definovat jako osnovu, která znázorňuje testovací přístup v SDLC (Software Development Life Cycle). Představuje širokou škálu metodologií návrhu softwarových testovacích případů. Jsou rozpracovány do řádné série kroků, které povedou k úspěšnému testování softwaru. Strategie testování softwaru jsou velmi důležité pro proces testování. Tyto strategie jsou většinou vyvíjeny testovacím týmem, projektovými manažery, funkčním testovacím týmem a softwarovým inženýrem. Níže jsou uvedeny testovací strategie (CHAUHAN & SINGH, 2014) (SHARMA, 2012).

6.1.1 Jednotkové testování

Jednotkové testování znamená testování prováděné na nejnižší úrovni nebo velmi podrobné úrovni. Tyto testy se používají k testování zcela základní jednotky softwarové aplikace, kterou může být modul, metoda nebo komponenta. Jednotka je základní modul (nejmenší možná sada řádků kódu, které lze testovat). Tento přístup se používá pro průběžné testování a revizi. Normálně vývojáři zapisují tyto testy a refaktorují kód aplikace, dokud neprojdou všechny testy jednotek. Testování jednotek spadá pod white box testování.

6.1.2 Integrační testování

Integrační testování přichází na řadu, když jsou kombinovány více než dva moduly a je třeba je testovat jako celek. Testování integrace se provádí až po testování jednotky. To lze provést jak metodou shora dolů, tak zdola nahoru. Hlavním cílem integračního testování je ověřit a zjistit, jak fungují všechny softwarové moduly, když jsou všechny malé moduly zkombinovány dohromady. Toto testování se provádí na strukturách ve velkém měřítku a na rozhraních.

6.1.3 Systémové testování

Testování systému znamená testování celého systému jako jednotky, jak název napovídá. Toto testování se provádí za účelem testování kvality celého systému. Zde hrají velmi důležitou roli funkční a nefunkční atributy jako spolehlivost, udržovatelnost a bezpečnost a specifikace požadavků systému. Všechny požadavky jsou kontrolovány. Pokud zde testy selžou, bude systém převeden zpět do SDLC a proces bude pokračovat.

6.1.4 Akceptační testování

Toto testování bude provedeno při předání systému uživatelům/zákazníkům pro tento systém ze strany vývojářů. Hlavním cílem akceptačního testování je ověřit, že systém funguje, spíše než nacházet chyby. Hlavním cílem smoke testování je otestovat dominantní funkce softwaru, ale ne do hloubky testovat všechny funkce. Pokud testy projdou, bude testování pokračovat do další úrovně/fáze, pokud ne,

bude testování zastaveno a budou informováni vývojáři s žádostí o nové sestavení s opravou aktuálního selhání.

6.2 Metodologie softwarového testování

Níže jsou uvedeny metodiky testování softwaru, které jsou popsány v pracích (AKTAŞ, YAĞDERELİ, & SERDAROĞLU, 2021) (SUPRIYONO, 2020) (ARUMUGAM, 2019).

6.2.1 White Box testování

V tomto typu testování jsou testy vyvíjeny na základě znalosti kódu, tj. interních implementačních detailů a struktury kódu. Tento typ testování je tedy vysoce produktivní při odhalování chyb a řešení problémů způsobených těmito chybami. White box testování je také známé jako clear box testování, white box analýza nebo clear box analýza. V této strategii musí mít tester jasnou znalost toho, co jednotlivé komponenty programu spojují a spolupracují při hledání chyb. Pro použití této metodologie testování je nutná znalost kódu, tato metoda se v praxi používá jen zřídka.

Různé typy technik white box testování jsou následující:

- Pokrytí rozhodování
- Testování primární dráhy
- Testování toku dat
- Pokrytí výpisu
- Kontrola průtoku
- Upravené pokrytí podmínek/rozhodnutí
- Testování větví
- Testování cesty

6.2.2 Black Box testování

Název „Black box“ (neboli „Černá skříňka“) označuje vnitřní detaily, práci, strukturu atd., nebo nic není viditelné nebo známé žádnou osobou, která to vidí. Testování černé skříňky je také založeno na stejném konceptu „černé skříňky“. Při tomto typu

testování nebude mít tester žádné znalosti o vyvinutém kódu nebo způsobu, jakým je navržen. Tester bude mít znalosti o požadovaných specifikacích a výstupech očekávaných produktem. Hlavním cílem je otestovat, jak se systém chová pro různé vstupy. Testuje pouze hlavní požadavek produktu. Ověří výstupy pro různé typy vstupů a zkontroluje, zda systém u některého typického vstupu selže.

Různé typy technik Black box testování jsou následující:

- Testování rozhodovací tabulky
- Odhadování chyb
- Testování přechodu stavu
- Testování všech párů
- Ekvivalenční rozdělení
- Graf příčiny a následku
- Analýza okrajových hodnot
- Testování případu použití

6.2.3 Grey Box testování

V posledních letech byla vyvinuta nová metodika testování, tzv. Grey box testování. Je také známá jako Gray box analýza. Grey box testování stojí mezi Black box a White box testováním. Testeři pracující s touto metodou budou mít malé znalosti o vyvinutém kódu, ale ne všechny, tj. jak komponenty systému interagují. Tito testeři by také měli přemýšlet a psát testy jako Black box testeři. Používá se pro penetrační testování.

6.3 Výhody a nevýhody automatizovaného testování

Regresní testování. Automatizované testování je pro tento typ testování nejběžnější. Regresní testování je typ testování zaměřený na kontrolu změn provedených v aplikaci nebo prostředí (ladění, sloučení kódu, migrace na jiný operační systém, databázi, webový server), aby se potvrdilo, že stávající funkce stále fungují. Regrese

mohou být funkční i nefunkční testy. Regresní testování obvykle používá testovací případy napsané v raných fázích vývoje a testování. To znamená, že regresní automatizované testy se provádějí v předem stanoveném časovém intervalu. Obvykle se stahují po každé úspěšné kompilaci (v malých projektech) nebo každý večer nebo každý týden. Tím je zaručeno, že změny v nové verzi programu nepoškodí již existující funkcionalitu. V článku (FAHAD & NADEEM, 2008) autoři diskutují o výhodách a nevýhodách použití UML diagramů pro regresní testování a analyzují, že UML model pomáhá efektivně identifikovat změny pro výběr regresního testu. Dále jsou uvedeny zásadní výhody automatizovaného testování.

- **Rychlost provedení**. Automatizované ověřovací skripty mohou nějakou dobu trvat. Jejich dokončení však zabere méně času, než kdyby měl člověk tyto kontroly provádět ručně. Proto automatizované testy pomáhají poskytnout rychlou zpětnou vazbu vývojovému týmu.
- **Úspora času pro testery**. Automatizace testování ušetří testerům čas. Proto se mohou více soustředit na zkoumání nových funkcí. Automatické kontroly mohou být spuštěny automaticky s minimálním dohledem, bez jakéhokoli dohledu nebo ručně.
  Obvykle, když se nepoužívají automatizované testy, dochází k cyklické situaci nedostatku času.
- **Schopnost vytvářet automatizované testy vývojáři**. Automatizované testy jsou obvykle napsány ve stejném jazyce jako testovaný produkt. V důsledku toho se odpovědnost za psaní, provádění a provádění testů stává sdílenou odpovědností. Ke kvalitě softwaru může přispět každý ve vývojovém týmu, nejen testeři.

Naopak nevýhody automatizovaného testování jsou následující.

- **Falešný pocit kvality produktu.** Z tohoto důvodu stojí za to věnovat zvláštní pozornost úspěšně absolvovaným automatizovaným testům. To je důležité zejména pro testování funkčnosti na uživatelské úrovni (UI) nebo na úrovni

systému. Automatizovaný test kontroluje pouze to, co je naprogramováno k testování. Všechny automatizované testy v testovací sadě mohou projít, ale některé chyby nemusí být identifikovány. Důvodem je, že tento test nebyl navržen tak, aby detekoval tyto specifické poruchy (OLIINYK & OLEKSIUK, 2019).

- **Nedostatečná spolehlivost**. Automatické kontroly nemusí být úspěšné kvůli mnoha faktorům. Automatické kontroly lze přerušit například změnou uživatelského rozhraní, vypnutím služby nebo problémem se sítí. Tyto problémy přímo neovlivňují testovaný program, ale mohou ovlivnit výsledek automatických testů.
- **Potřeba podpory**. Je třeba si uvědomit, že automatizované testy vyžadují údržbu. Automatické kontroly jsou krátkodobé. Pokud nejsou aktualizovány, dojde k selhání. Je také možné, že některé kontroly již nejsou relevantní nebo že neodpovídají novým implementacím softwaru. Tato selhání mohou ovlivnit výsledky testu. Psaní automatizovaného testovacího případu není jednorázové úsilí. Pokud se chce z automatických testů vytěžit maximum, je třeba je aktualizovat a mít je aktuální. To obvykle vyžaduje čas, úsilí a zdroje.
- **Odhalení méně chyb než ručním testováním.** Většina chyb je odhalena „náhodně“ nebo během průzkumného testování. Tento typ testování zahrnuje simultánní studium softwarového produktu, návrh testů a jejich provedení. Jeho specifičnost spočívá v tom, že na každém průzkumném testovacím sezení je možnost otestovat aplikaci různými způsoby. Na druhou stranu automatické kontroly vždy sledují nastavenou cestu, někdy se stejnou testovací datovou sadou. Zde lze zmínit jeden z principů testování – „paradox pesticidu“. Spočívá v tom, že opakovaným prováděním stejných testů čelíme skutečnosti, že nacházejí méně a méně chyb. Je to dáno vývojem systému, v důsledku čehož je mnoho nalezených závad opraveno a staré testovací případy již nefungují. To zase snižuje pravděpodobnost nalezení nových vad na produktu (UMAR, 2019).
- **„Automatizace testování není vždy testováním“.** Zde se rozumí, že testování je výzkumná činnost. Testování vyžaduje specifické znalosti, cílevědomou mysl a ochotu naučit se aplikaci. Bohužel se mnoho lidí mýlí o

důležitosti automatizace testování. Dostanou nástroj pro automatizaci testování a chtějí se zbavit všech „ručních testerů“. To však není možné, protože testování není jen o provedení sady předem definovaných testovacích kroků a porovnání skutečných výsledků s očekávanými výsledky. Nejnovějšími úkoly jsou automatické kontroly.

- **Schopnost seskupovat selhání do shluků**. V článku (MARIANI, HAO, SUBRAMANYAN, & ZHU, 2017) je uveden přístup k automatické detekci procházejících a neúspěšných spouštění pomocí detekce anomálií na bázi clusteru na datech dynamického provádění. Klíčovou hypotézou, která je základem tohoto přístupu, je, že selhání se seskupují do malých shluků, zatímco procházející provedení se seskupují do větších.

6.4 Jaké testy lze automatizovat

Softwarové komponenty a procesy, které lze automatizovat, jsou popsány v práci (OLIINYK & OLEKSIUK, 2019) následovně.

- Těžko dostupná místa v systému. Například procesy na pozadí, protokolování souborů, vstup do databáze.
- Často používaná funkcionalita tam, kde je vysoké riziko chyb: platební systémy, registrace atd. Automatizace kontrol kritické funkčnosti zajišťuje rychlé chyby, protože test trvá v průměru několik minut.
- Zatěžovací testy, které testují funkčnost systému s velkým počtem požadavků.
- Operace se šablonami, včetně vyhledávání dat, zadávání formulářů s mnoha poli, kontrola jejich zachování.
- Ověřování vstupů – vyplnění pole s nesprávnými údaji a kontrola správnosti výstupu.
- Dlouhé scénáře typu end-to-end. Například scénář internetového obchodu, který zahrnuje: registraci uživatele, stránku s podrobnostmi o produktu, nákupní košík, nákup produktu a potvrzení nákupu.
- Ověřování údajů vyžadujících přesné matematické výpočty, např. účetní nebo analytické zpracování.

- Kontrola správnosti vyhledávání dat.

Přestože se objevují novější nástroje pro automatizované testování, je stále obtížné otestovat funkčnost uživatelského rozhraní. Bez alespoň jednoho ručního testovacího procesu také není možné automatizovat testování nové funkce. Proto říci, že je možné vše zautomatizovat, je prozatím přehnané.

7 Přehled nástrojů pro automatizované testování webových aplikací

Tato kapitola obsahuje vybrané nejznámější nástroje pro automatizované testování, pro jednotlivé kategorie testů.

7.1 Jednotkové testování

7.1.1 JUnit

JUnit je open-source knihovna pro automatizované testování aplikací naprogramovaných v jazyce Java. Je jedním z nejstarších a nejpopulárnějších nástrojů pro automatizaci testů v Java, který se často používá v různých oblastech, jako jsou e-commerce, finance, telekomunikace a bezpečnost. JUnit umožňuje vývojářům automatizovat jednotkové testy, integrační testy a funkční testy. JUnit poskytuje také nástroje pro sledování a analýzu testů, jako jsou grafy, statistiky a možnosti exportu dat.

JUnit je jednoduchý a snadno použitelný nástroj pro automatizaci testů, který umožňuje vývojářům snadno vytvořit a spustit testy. JUnit také poskytuje širokou podporu a dokumentaci od komunity, což umožňuje vývojářům snadno se naučit používat tento nástroj. Implementace a údržba testů v JUnit může být časově náročná. Testy jsou psané ručně a mohou vyžadovat další nástroje pro automatizaci a sledování. Navíc, když dochází ke změnám v aplikaci, testy se mohou stát zranitelnými a vyžadují tak častou údržbu.

JUnit je silný a flexibilní automatizovaný testovací nástroj pro aplikace naprogramované v jazyce Java. I když má JUnit své nevýhody, jako je potenciální náročnost na implementaci a údržbu testů, stále se jedná o velmi užitečný nástroj pro automatizaci testů a jeho popularita a dlouhá historie ukazuje, že je pro mnoho vývojářů stále nepostradatelným nástrojem pro kvalitní vývoj softwaru (OLAN, 2003).

7.1.2 PHPUnit

PHPUnit je open-source testovací framework pro PHP. Umožňuje automatizovat jednotkové testy, integrační testy a funkční testy, které ověřují, že kód funguje tak, jak má. PHPUnit je jedním z nejpopulárnějších testovacích nástrojů pro PHP a používá se pro testování mnoha různých typů aplikací, včetně e-shopů. Je založen na JUnit, což je knihovna pro automatizované testování v jazyce Java. PHPUnit umožňuje vývojářům automatizovat jednotkové testy, integrační testy a funkční testy, což umožňuje ověřit, že kód aplikace funguje správně a že se chová tak, jak se očekává.

PHPUnit poskytuje širokou škálu funkcí pro automatizaci testů, včetně podpory pro testy jednotlivých tříd a metod, podpory pro testy více vláken, podpory pro testy s různými daty a podpory pro testy s různými konfiguracemi. PHPUnit také poskytuje nástroje pro sledování a analýzu testů, jako jsou výkresy, statistiky a možnosti exportu dat. PHPUnit je velmi populární nástroj pro automatizaci testů v PHP, protože je snadno použitelný, flexibilní a poskytuje širokou škálu funkcí pro automatizaci testů. PHPUnit se často používá pro automatizaci testů v různých oblastech, jako jsou e-commerce, finance, telekomunikace a bezpečnost. PHPUnit má také velkou a aktivní komunitu, která poskytuje podporu a dokumentaci pro vývojáře (ZANDSTRA, 2016).

Jednou z nevýhod PHPUnit je, že testy mohou být náročné na implementaci a údržbu. Testy musí být napsány ručně a mohou vyžadovat další nástroje pro automatizaci a sledování. Testy také mohou být zranitelné ke změnám v rozhraní aplikace, což může vést k nutnosti časté údržby (CARTER, 2018).

7.1.3 Nette Tester

Nette Tester je testovací nástroj pro PHP, který je součástí frameworku Nette. Tento testovací nástroj lze použít k automatizovanému testování jednotek, k automatizovanému testování integrace, k automatizovaným funkčním testům. Díky tomuto podporuje různé testovací scénáře. Jeho hlavními výhodami jsou:

- Snadnost použití: Nette Tester je jednoduchý a intuitivní testovací nástroj, který je snadno integrovatelný do existujících projektů nebo aplikací.
- Podpora různých testovacích scénářů: Nette Tester podporuje různé typy testů, jako jsou jednotkové testy, integrační testy nebo funkční testy, což umožňuje plně testovat celou aplikaci.
- Kompatibilita s PHPUnit: Nette Tester je kompatibilní s PHPUnit, což znamená, že může být snadno použit s již existujícími projekty nebo knihovnami, které využívají PHPUnit.
- Různé formáty výstupu: Nette Tester umožňuje výstup v různých formátech, jako jsou text, HTML nebo JUnit XML, což umožňuje snadno integrovat testy do dalších nástrojů pro sledování chyb nebo kontinuální integraci.
- Kompatibilita s Nette Framework: Nette Tester je součástí Nette Framework, což znamená, že je snadno integrovatelný do projektů, které využívají tento framework.
- Dokumentace a komunita: Nette Tester má dostupnou a kvalitní dokumentaci a má aktivní komunitu, která může pomoci při řešení problémů nebo nastavení nástroje.

Z tohoto důvodu může být Nette Tester dobrou volbou pro automatizované testování e-shopu nebo aplikace používající Nette Framework.
Jednou z nevýhod Nette Tester je, že je specifický pro framework Nette, a tak není kompatibilní s jinými frameworky. To znamená, že vývojáři, kteří používají jiný framework než Nette, nemohou použít tento nástroj pro automatizaci svých testů, další výhodou je, že každý test je klasický PHP skript a lze jej tak samostatně spouštět, což umožňuje snadno test spustit a zjistit, zda neobsahuje například programátorskou chybu a pokud ano, je možné test snadno krokovat ve zvoleném IDE a chybu nalézt (GRUDL, Začínáme s Nette Tester, 2023) (GRUDL, Proč používám Nette Tester, 2014).

Srovnání PHPUnit a Nette Tester:

- PHPUnit je univerzální testovací framework, který lze použít pro testování různých typů aplikací v PHP, zatímco Nette Tester je speciálně určen pro aplikace využívající Nette Framework.
- PHPUnit má širší podporu v různých komunitách a projektech, zatímco Nette Tester je méně rozšířený.
- PHPUnit má více funkcí a možností nastavení, zatímco Nette Tester je jednodušší na použití a snadněji se integruje do projektů postavených na Nette Framework.

7.2 Testování uživatelského rozhraní

Existuje mnoho nástrojů pro automatizované testování webových aplikací. Některé z nejpopulárnějších jsou:

- Selenium: Selenium je open-source automatizovaný testovací nástroj, který podporuje různé programovací jazyky, včetně Java, C#, Python, Ruby a JavaScript. Selenium umožňuje automatizovat testy pro webové aplikace a prohlížeče, jako jsou Chrome, Firefox, Safari a Edge. (RAMYA, SINDHURA, & SAGAR, 2017)
- TestComplete: TestComplete je komerční automatizovaný testovací nástroj, který podporuje různé programovací jazyky a webové prohlížeče. Má širokou škálu funkcí pro automatizaci testů, včetně podpory pro různé typy testů, jako jsou jednotkové testy, integrační testy a funkční testy. (KAUR & KUMARI, 2011)
- Testim je SaaS (Software as a Service) automatizovaný testovací nástroj, který umožňuje automatizovat testy webových aplikací. Testim se zaměřuje na automatizaci testů pomocí strojového učení, což umožňuje rychle a efektivně automatizovat testy webových aplikací bez nutnosti ručně psát testovací skripty. Testim podporuje různé webové prohlížeče, jako je

Chrome, Firefox, Safari a Edge, a různé programovací jazyky, jako jsou JavaScript, TypeScript, Python a Java. (RODRIGUEZ, 2022)

7.2.1 Selenium

Selenium je open-source automatizovaný testovací nástroj pro webové aplikace, který umožňuje automatizovat testy pro webové prohlížeče, jako jsou Chrome, Firefox, Safari a Edge. Tento nástroj je kompatibilní s různými programovacími jazyky, jako jsou Java, C#, Python, Ruby a JavaScript, což umožňuje vývojářům používat jejich oblíbený jazyk pro automatizaci testů.

Selenium se skládá ze dvou hlavních částí: Selenium WebDriver a Selenium IDE. Selenium WebDriver je knihovna, která umožňuje programovat testy v různých jazycích a automatizovat je na webové prohlížeče. Selenium IDE je integrované vývojové prostředí, které umožňuje vytvářet testy pomocí grafického rozhraní a záznamu akcí uživatele. Selenium je velmi populární nástroj pro automatizaci testů webových aplikací, protože umožňuje testovat aplikace na různých webových prohlížečích a různých operačních systémech. To znamená, že se vývojáři mohou ujistit, že aplikace funguje správně na různých platformách a prohlížečích. Selenium také umožňuje automatizovat testy v různých úrovních, od jednotkových testů až po funkční testy, což umožňuje vývojářům ověřit, že aplikace funguje tak, jak má (RAMYA, SINDHURA, & SAGAR, 2017).

Selenium také poskytuje různé nástroje pro sledování a analýzu testů, jako jsou grafy a statistiky. To umožňuje vývojářům snadno identifikovat problémy a najít možnosti jejich řešení. Selenium má také velkou a aktivní komunitu, která poskytuje podporu a dokumentaci pro vývojáře, kteří se chtějí naučit používat tento nástroj. Kromě toho existuje mnoho doplňků a rozšíření pro Selenium, které umožňují rozšířit jeho funkce a zjednodušit práci s ním.

Selenium dále podporuje automatizaci testů pro mobilní aplikace prostřednictvím Appium, což umožňuje vývojářům automatizovat testy pro aplikace na různých operačních systémech, jako jsou iOS a Android.

Jedna z nevýhod Selenium je, že testy mohou být časově náročné na implementaci a údržbu. Testy musí být napsány ručně a mohou vyžadovat další nástroje pro automatizaci a sledování, jako je např. Selenium Grid. Testy také mohou být zranitelné k změnám v rozhraní aplikace, což může vést k nutnosti časté údržby (VILA, NOVAKOVA, & TODOROVA, 2017).

Lze říci, že Selenium je silný a flexibilní automatizovaný testovací nástroj pro webové aplikace. Umožňuje automatizovat testy na různých webových prohlížečích a operačních systémech, což umožňuje vývojářům ověřit, že aplikace funguje tak, jak má. Selenium má také širokou podporu a dokumentaci od komunity, což umožňuje vývojářům snadno se naučit používat tento nástroj. Je však třeba brát v úvahu, že Selenium může být časově náročný na implementaci a údržbu a může být zranitelný ke změnám v rozhraní aplikace.

7.2.2 TestComplete

TestComplete je komplexní nástroj pro automatizované testování softwaru, který umožňuje vývojářům automatizovat jednotkové testy, integrační testy a funkční testy. Tento nástroj je schopen automatizovat testy nejen pro webové aplikace, ale i pro desktopové aplikace, aplikace mobilních zařízení a aplikace pro automatizaci průmyslu. TestComplete je velmi flexibilní, umožňuje používat různé jazyky pro scriptování, jako je Python, JavaScript, VBScript nebo C#, což poskytuje vývojářům možnost volby toho, který je pro ně nejvhodnější. TestComplete také podporuje různé operační systémy, včetně Windows, MacOS a Linux, což umožňuje testovat aplikace na různých platformách.

TestComplete je také snadno použitelný nástroj, nabízí intuitivní grafické rozhraní, které umožňuje vývojářům snadno vytvářet a spouštět testy. Také má rozsáhlou dokumentaci a podporu od komunity. Jednou z nevýhod TestComplete je, že může být pro vývojáře náročné naučit se používat tento nástroj, zejména pokud nejsou zkušení s automatizací testů. TestComplete má také vyšší cenu než některé jiné

nástroje pro automatizaci testů, což může být pro některé společnosti nebo projekty limitující faktor. (KAUR & KUMARI, 2011)

7.2.3 Testim

Testim je nástroj pro testování webových aplikací, který umožňuje vývojářům snadno automatizovat své testy. Testim se odlišuje od jiných nástrojů pro automatizaci testů tím, že poskytuje jednoduchou a intuitivní metodu pro automatizaci testů, která je snadno pochopitelná i pro vývojáře bez zkušeností s automatizací testů. Testim umožňuje vývojářům automatizovat jednotkové testy, integrační testy a funkční testy, což umožňuje ověřit, že kód aplikace funguje správně a že se chová tak, jak se očekává. Nástroj poskytuje grafické rozhraní, které umožňuje vývojářům snadno vytvářet a spouštět testy, aniž by museli psát kód. Testim také poskytuje nástroje pro sledování a analýzu testů.

Jednou z nevýhod Testim je, že některé funkce mohou být omezené v porovnání s jinými nástroji pro automatizaci testů. Testim se zaměřuje především na automatizaci testů webových aplikací a méně na automatizaci testů desktopových aplikací a mobilních aplikací. Navíc, někteří vývojáři mohou najít omezení v kompatibilitě s různými prohlížeči nebo operačními systémy.

Testim má také integrovanou podporu pro umělou inteligenci (AI). Použití umělé inteligence v Testimu umožňuje vývojářům vytvořit inteligentní testy, které se automaticky přizpůsobují změnám v aplikaci. Jedním z hlavních způsobů, jak Testim využívá umělou inteligenci, je v automatickém vytváření a údržbě testovacích skriptů. Testim používá algoritmy strojového učení k analýze kódu aplikace a automaticky vytváří testovací skripty, které ověřují funkčnost aplikace. Tyto skripty se pak automaticky aktualizují v případě změn v aplikaci, což umožňuje vývojářům snížit čas potřebný k údržbě testů. Také používá umělou inteligenci k rozpoznávání elementů v aplikaci a k automatickému vytváření testů pro různé rozměry zařízení. Toto umožňuje vývojářům snadno testovat své aplikace na různých zařízeních a rozlišeních obrazovky. (RODRIGUEZ, 2022) (Testim.Io Tutorial: Speed-Up Authoring And Executino Of Automated Tests, 2023)

Zatímco Testim poskytuje řadu užitečných funkcí pro automatizaci testů s pomocí umělé inteligence, je třeba si uvědomit, že AI není všehoschopné a může mít své omezení a chyby, například:

- Náročnost na nastavení: Použití umělé inteligence v Testimu může vyžadovat další nastavení a konfiguraci, než je tomu u klasických testů. Vývojáři by tedy měli být ochotni investovat další čas a úsilí do naučení se používání tohoto nástroje.
- Chyby v AI: Umělá inteligence může být ovlivněna chybami nebo nedostatky v tréninkových datech, což může vést k chybným výsledkům nebo nesprávnému rozpoznání elementů v aplikaci.
- Omezená flexibilita: Použití AI v Testimu může omezovat možnosti vývojářů při nastavování testů nebo vytváření vlastních testovacích skriptů.
- Náklady: Použití AI v Testimu může zvyšovat náklady na licenci nebo používání této funkce.

I přes tyto omezení, Testim s AI poskytuje vývojářům širokou škálu užitečných funkcí pro automatizaci testů a umožňuje efektivněji testovat webové aplikace s menším úsilím a časem.

7.3 Testování výkonu

7.3.1 WebLOAD

Nástroj pro testování zátěže a výkonu pro webové aplikace na podnikové úrovni. WebLOAD je nástroj pro podniky s velkou uživatelskou zátěží a složitými požadavky na testování. Umožňuje provádět zátěžové testování na jakékoli internetové aplikaci generováním zátěže z cloudu i lokálních počítačů. Silnými stránkami WebLOAD jsou jeho flexibilita a snadné použití – umožňuje rychle definovat testy, které jsou potřeba, pomocí funkcí, jako je nahrávání/přehrávání založené na DOM, automatická korelace a skriptovací jazyk JavaScript.

Tento nástroj poskytuje jasnou analýzu výkonu webové aplikace, určuje problémy a úzká místa, která mohou stát v cestě dosažení potřebných požadavků na zatížení a

odezvu. WebLOAD podporuje stovky technologií – od webových protokolů po podnikové aplikace a má vestavěnou integraci s Jenkins, Selenium a mnoha dalšími nástroji, které umožňují nepřetržité testování zátěže pro DevOps. (SRIVASTAVA, KUMAR, & SINGH, 2021)

7.3.2 LoadNinja

LoadNinja od SmartBear umožňuje rychle vytvářet sofistikované zátěžové testy bez skriptů, zkracuje dobu testování o 50 %, nahrazuje emulátory zátěže skutečnými prohlížeči a získává použitelné metriky založené na prohlížeči. Je možné snadno zachytit interakce na straně klienta, ladit v reálném čase a okamžitě identifikovat problémy s výkonem. LoadNinja umožňuje týmům zvýšit pokrytí testů bez obětování kvality odstraněním zdlouhavého úsilí s dynamickou korelací a překladem skriptů. S LoadNinja se mohou inženýři, testeři a produktové týmy více soustředit na vytváření aplikací, které se škálují, a méně se soustředit na vytváření skriptů pro zátěžové testování. (KOŁTUN & PAŃCZYK, 2020)

Funkce:

- Vytváření a přehrávání zátěžových testů bez skriptů pomocí rekordéru InstaPlay.
- Skutečné provedení zátěžového testu prohlížeče ve velkém měřítku.
- VU Debugger – ladění v reálném čase.
- VU Inspector – správa aktivity virtuálních uživatelů v reálném čase.
- Hostováno v cloudu, není potřeba žádný server ani údržba.
- Sofistikované metriky založené na prohlížeči s funkcemi analýzy a vytváření sestav.

7.3.3 HeadSpin

HeadSpin nabízí svým uživatelům nejlepší možnosti testování výkonu v oboru. Uživatelé mohou optimalizovat své digitální zážitky pomocí možností testování výkonu platformy HeadSpin identifikací a řešením problémů s výkonem napříč aplikacemi, zařízeními a sítěmi. (KOŁTUN & PAŃCZYK, 2020)

Funkce:

- Monitorujte a optimalizujte výkon v rámci celé cesty uživatele
- HeadSpin poskytuje skutečná data z reálného světa a odstraňuje nejednoznačnost z tisíců zařízení, sítí a umístění.
- Uživatelé mohou využít pokročilé schopnosti umělé inteligence k automatické identifikaci problémů s výkonem během testování dříve, než budou mít dopad na uživatele

7.3.4 ReadyAPI

SmartBear nabízí vše v jednom – automatizovanou testovací platformu API s názvem ReadyAPI. Obsahuje různé nástroje jako Swagger & SwaggerHub, SoapUI NG, ReadyAPI Performance, Secure Pro, ServiceV a AlertSite. ReadyAPI Performance je API nástroj pro zátěžové testování. Tento nástroj pro testování API zajistí, že API mohou fungovat kdekoli. Umožní nainstalovat agenty zatížení na jakýkoli server nebo cloud, stejně jako on-premise. Poskytuje pokročilé metriky výkonu pro běhy zátěžových testů.

SoapUI NG je nástroj pro funkční testování a pro testování výkonu lze použít tyto případy použití funkčního testování navržené v SOAPUI. Tento nástroj pro testování zátěže pomůže s testováním rychlosti, škálovatelnosti a výkonu rozhraní API, serverů a síťových zdrojů. Má funkce flexibilního generování zátěže, paralelní testy zátěže API, monitorování serveru a předem vytvořené šablony zátěže. (JAN, NGUYEN, & BRIAND, 2016)

7.3.5 LoadView

LoadView je plně spravovaný nástroj pro testování zátěže na vyžádání, který umožňuje úplné bezproblémové testování zátěže. Na rozdíl od mnoha jiných nástrojů pro zátěžové testování provádí LoadView testování ve skutečných prohlížečích (nikoli v bezhlavých fantomových prohlížečích), které poskytují extrémně přesná data a úzce napodobují skutečné uživatele. LoadView je 100%

cloudový, škálovatelný a lze jej nasadit během několika minut. Pokročilé funkce zátěžového testování zahrnují skriptování Point and Click, globální cloudovou infrastrukturu, reálné testování prohlížeče. (LEE, 2023)

8 Implementace automatizovaného testování

8.1 Představení vybraného e-shopu a testovaných funkcí

Vybraný e-shop, který bude implementovat automatizované testování, se zaměřuje na prodej elektroniky v české republice, ale i zahraničí, a jedná se tak o rozsáhlou aplikaci, které implementace automatizovaného testování může velmi usnadnit budoucí vývoj.

Backend aplikace běží na linuxovém serveru a je vyvíjen v jazyce PHP a používá český framework Nette. Nette je open-source framework navržený pro tvorbu webových aplikací v jazyce PHP, který využívá návrhový vzor MVP neboli model-view-presenter. Byl vytvořen s vizí pomoci vývojářům snadno a rychle vytvářet kvalitní a bezpečné webové aplikace. Vyznačuje se zejména jednoduchostí, efektivitou a flexibilitou a poskytuje řadu užitečných nástrojů a funkcí, jako jsou formulářové komponenty, šablony, aplikační kontejner, autentifikaci a autorizaci atd. Tento framework je napsán v čistém PHP bez žádných dalších závislostí na jiných knihovnách, což umožňuje vývojářům snadno integrovat Nette do již existujícího projektu nebo jej použít jako základ pro nový projekt.

Frontendová část aplikace je velmi úzce spojena s backendovou částí, jelikož využívá jazyka PHP pro dynamické zobrazení stránky pomocí skriptů a doplňuje tak značkovací jazyk HTML 5 a styly využívající CSS. Samozřejmostí je využívání JavaScriptu pro spouštění skriptů na straně klienta. Je zde využit systém šablon Nette Latte, který se využívá pro webové aplikace v rámci frameworku Nette. Nette Latte umožňuje vývojářům oddělit logiku aplikace od její pohledové části, což poskytuje lepší strukturu a snadnější údržbu aplikace. Nette Latte používá jednoduchý syntax pro tvorbu šablon a umožňuje vývojářům vložit logiku do šablon pomocí speciálních značek a funkcí. Tyto značky a funkce slouží vývojářům pro jednoduché vygenerování HTML, CSS a JavaScript kódu na straně klienta. Nette Latte také podporuje řadu výkonných funkcí, jako je filtrování dat, řízení stavu aplikace a výpočty. Všechny tyto funkce jsou navrženy tak, aby umožňovaly vývojářům rychle a efektivně vytvářet a upravovat šablony pro své webové aplikace.

Aplikace využívá relační databázový model systému MySQL, kde je jako formát uložiště (storage engine) InnoDB, kdy se jedná o jedno z nejpoužívanějších implementací transakčního uložiště v databázích MySQL, která nabízí široké spektrum funkcí pro správu dat, včetně podpory transakcí, referenční integritu, zálohování a obnovu dat, zápis do více tabulek současně a řadu dalších funkcí. Také poskytuje vysokou úroveň výkonu a škálovatelnosti.

Pro tuto aplikaci byly vytvořeny požadavky na implementaci automatizovaného testování. První zásadní oblastí, kterou je třeba testovat jsou jednotkové a integrační testy. Jedná se o testování základních prvků aplikace, kde je třeba zajistit integritu a funkčnost, a to i zpětně během neustálého vývoje. Další podstatnou částí aplikace, kterou je potřeba testovat je uživatelské rozhraní. Pro úspěšnost aplikace je důležité, aby se uživatelům zobrazoval obsah správně, prvky se nepřekrývaly a byly spolehlivé i po funkční stránce. Poslední podstatnou oblastí, která je pro provoz aplikace a uživatelský zážitek důležitá je výkon a s tím související testování zátěže a výkonnosti aplikace za různých podmínek od běžného provozu k neočekávanému vytížení, aby se dalo zmapovat chování aplikace za těchto podmínek a případně se jim pokusit předcházet.

8.2 Výběr nástrojů

Vzhledem k tomu, že serverová část e-shopu je naprogramována v jazyce PHP za využití frameworku Nette, byl pro implementaci jednotkových, integrační a funkčních testů zvolen nástroj Nette Tester, který je tímto frameworkem podporován a je s ním úzce spojen, což umožnuje snadněji testovat specifika právě tohoto frameworku. I díky tomu, že je tento nástroj od stejných autorů jako framework a značně se podobá, bude pro vývojáře jednodušší se v něm zorientovat a pokračovat ve vytváření a v automatizaci testů podle zde vypracovaných typových příkladů.

Nette Tester umožňuje provádět testování spouštěním jednoduchých PHP skriptů, ale i vytváření testovacích tříd, kde metody s prefixem „*test*“ jsou jednotlivě spouštěny a měly by obsahovat testování jedné konkrétní věci. Testovací třídy jsou

dále popsány v implementaci tohoto nástroje. Co se týče samotného testování a vyhodnocování správného fungování kódu, Nette Tester poskytuje třídu *Tester\Assert*, která umožňuje používání asercí a tím validaci, že daná hodnota skutečně odpovídá hodnotě očekávané. V této třídě je definováno 23 metod aserce, které je možno použít, zde budou vyjmenovány jen ty využité v implementaci a nejčastěji využívané. Metody *Assert::same($ocekavana, $aktualni)* a *Assert::notSame($ocek, $akt)* slouží k ověření, že jsou hodnoty, předávané jako parametry, stejné, jako při využití PHP operátoru *===,* u druhé metody se samozřejmě očekává opačný stav a to, že se hodnoty liší. Tomu je podobná i funkce *Assert::equal($ocek, $akt),* která ale neřeší identitu objektů, hraničně odlišná desetinná čísla a pořadí dvojic klíč-hodnota v polích. K této metodě je i metoda opačná *Assert::notEqual($ocek, $akt).* Dále je zde dvojice metod *Assert::true($value)* a *Assert::false($value)*, kdy tyto porovnávají hodnotu zda je *true,* v druhém případě *false.* K tomuto ještě existují podobné metody *Assert::truthly($value)* a *Assert::falsey($value)*, kde hodnota musí být pravdivá (u druhé metody nepravdivá), což odpovídá podmínce *if ($value)*, respektive *if (!$value).* Také je zde dvojce metod, které testují zda je hodnota rovna *null* či nikoli – *Assert::null($value), Assert:notNull($value).* Zajímavé metody jsou také *Assert::type($type, $value)* a *Assert::exception($callable, $class)*, kdy první metody testuje hodnotu *$value* na její typ, který má být shodný s proměnnou $*type*, za kterou lze doplnit řetězcem *array, list, bool, callable, float, int, null, object, resource, scalar, string*, případně názvem třídy, kdy *$value* musí být objektem této třídy. Druhá metoda testuje vyhození výjimky. Používá k tomu dva parametry, prvním je funkce, která je testována, a druhý je třída výjimky, která má být ve funkci vyhozena. Pro testování řetězců dle vzorů, například při testování zda je řetězec email, telefon či jiný řetězec definovaného formátu, je k dispozici metoda *Assert::match($pattern, $value)*. Proměnná *$pattern* je určitý vzor, který lze definovat pomocí regulárních výrazů nebo zástupných znaků, kdy při použití regulárních výrazů je nutné tento výraz ohraničit znaky ~ nebo #.

Další užitečné metody a možnosti Nette Tester lze nalézt v jeho dokumentaci (GRUDL, Začínáme s Nette Tester, 2023), kde lze nalézt například možnosti anotací testů, pomocné třídy, průběžné testování s Travis a další.

Přesto že Nette Tester dokáže testovat vykreslení šablony a jednotlivých elementů, nedokáže komplexněji testovat uživatelské interakce a kompatibilitu pro různé zařízení a prohlížeče. Z tohoto důvodu je pro tuto oblast testování potřeba zvolit jiný nástroj. V 7. kapitole je uvedeno několik vhodných nástrojů pro tuto problematiku, ze kterých byl vybrán nástroj **Selenium**, konkrétně Selenium IDE. Byl zvolen z důvodů, že se jedná o open-source nástroj a je tedy dostupný zdarma, dále pro jeho použití není třeba znalost programování a vytvářet testy pomocí tohoto nástroje zvládne i běžný uživatel v internetovém prohlížeči, čímž se ušetří práce vývojářům. Zároveň tento nástroj umožňuje převést vytvořené testy do zdrojového kódu ve zmíněných jazycích, či je v nich přímo vytvářet a lze tak provádět i složitější testy. Jedná se také o velmi rozšířený nástroj, takže je možné dohledat spoustu informací a návodů.

Selenium IDE je nástroj pro záznam interakcí prohlížeče pro testovací případy. Tento testovací nástroj s otevřeným zdrojovým kódem se snadno instaluje a spustí: stačí si jej stáhnout jako plugin do webového prohlížeče, jako je Chrome nebo Firefox. Pomocí tohoto nástroje je možné vytvářet jednoduché nebo složité testovací scénáře. Stejně jako ostatní vývojová IDE je zde možnost nastavit body přerušení, pozastavit a zkontrolovat proměnné, když dojde k chybě. Navíc lze svůj testovací případ znovu použít v jiném testovacím případu, což usnadňuje a urychluje opakované testování. Selenium IDE rozumí podmíněným příkazům jako if a while. Toto logické zpracování umožňuje rozšířit a pokrýt více různých testovacích scénářů. (KRISHNA & GOPINATH, 2021)

Kromě těchto funkcí může Selenium IDE také provádět následující. Manipulace s dalšími komponentami uživatelského rozhraní: při interakci s webovou stránkou je možné narazit na další informační okna, oznámení, výstrahy a další. Do testovacích skriptů lze přidat podmínky pro zpracování dalších komponent uživatelského

rozhraní. Současné spouštění testovacích skriptů: Testovací případy mohou být spouštěny současně a ušetřit tak celkový čas testování. Integrace do CI: díky Selenium Command Line Runner je k dispozici integrace testování Selenium do procesu CI. Díky tomu tak lze automaticky spouštět testy, když vývojáři sloučí svůj kód. Pluginy: funkce Selenium IDE lze dále rozšířit instalací pluginů nebo lze vytvářet vlastní pluginy, které vyhovují konkrétním potřebám. (UPPAL & CHOPRA, 2012)

Jednotlivé kroky testu se skládají z příkazů. Selenium IDE má k dispozici mnoho příkazů. Každý krok obsahuje příkaz, cíl, na který se příkaz aplikuje (target) a případně i hodnotu (value) – například pro vyplnění pole. V tabulce níže jsou vypsány nejčastěji používané příkazy včetně příkladů využití.

Tabulka 1 – Selenium IDE příkazy

Příkaz	Popis	Příklad použití
open	Otevření webové stránky	target: eshop.cz
type	Vloží text do pole	target://input[@name='nazev_pole'] value: vyplnena_hodnota
click	Provede kliknutí na prvek	target://input[@id='koupit']
pause	Čeká určitý počet sekund	target: 1000
waitForText	Čeká na text	target://input[@name='nazev_pole'] value: ocekavany_text
selectWindow	Přesun do nového okna	target: null

Zdroj: (What Is Selenium IDE? + 7 Important Selenium IDE Tips, 2021)

Jelikož požadavky obsahují testování výkonu a stability aplikace a žádný z dosud použitých nástrojů se touto problematikou nezabývá a neumožňuje tyto testy provádět, je třeba použít další nástroj. Z uvedených nástrojů pro tuto oblast byl nakonec zvolen nástroj **LoadView,** který umožňuje provádět zadané testy a navíc

poskytuje 30denní zkušební dobu zdarma**.** LoadView je cloudová platforma pro testování výkonu webových aplikací a služeb. Tento nástroj poskytuje pokročilé funkce pro simulování reálných uživatelských scénářů, analýzu výkonu a monitorování zátěže v reálném čase. Následuje několik důvodů, proč byl vybrán nástroj LoadView pro testování výkonu této webové aplikace:

- Simulace reálných uživatelských scénářů: LoadView umožňuje vytvářet testovací scénáře, které simulují reálné uživatelské interakce s aplikací, včetně klikání na odkazy, zadávání formulářů a dalších akcí.
- Škálovatelnost: LoadView je cloudová platforma, která umožňuje snadno škálovat zátěž testů v závislosti na potřebách aplikace. To znamená, že je možné simulovat různé úrovně provozu a zátěže a zjistit, jak aplikace bude reagovat.
- Analýza výkonu: LoadView poskytuje pokročilé funkce pro analýzu výkonu aplikace, včetně přesného měření času odezvy a latence, kontrolu využití zdrojů a detekce přetížení serverů.
- Monitorování v reálném čase: LoadView umožňuje sledovat výsledky testů v reálném čase, což umožňuje rychle identifikovat a řešit problémy, jakmile se objeví.
- Integrovatelnost: LoadView je snadno integrovatelný s dalšími nástroji pro testování, jako jsou například Jenkins nebo Selenium.

Celkově lze říci, že LoadView je velmi užitečný nástroj pro testování výkonu webových aplikací a služeb, protože poskytuje přesné a spolehlivé výsledky testů, umožňuje simulovat reálné uživatelské scénáře a umožňuje škálovat zátěž testů podle potřeb aplikace. Navíc je tento nástroj nabízen ve zkušební verzi zdarma na 30 dní a je tak možné si vyzkoušet jeho funkce před samotnou koupí licence. (LEE, 2023)

8.3 Implementace vybraných nástrojů

8.3.1 Nette Tester

Nejzákladnější oblastí v testování jsou jednotkové testy, proto byl pro implementaci jako první zvolen nástroj Nette Tester. Pomocí tohoto nástroje budou také implementovány integrační a funkční testy, které tento framework podporuje a je tedy efektivní ho použít, nežli pro každý druh testů využívat různé frameworky a nástroje, jelikož by se vývojáři museli s každým nástrojem zvlášť seznamovat a přizpůsobit se jim, kdežto použitím jednoho nástroje na co nejvíc možných případů ušetří vývojářům čas a usnadní orientaci a přehlednost aplikace.

Jednotkové testování je důležité, jelikož se testují jednotlivé elementární části kódu, zde konkrétně třídy. Většina vytvořených testů je založena na black box testování (jen pohled z venku, tedy na veřejné vlastnosti – typicky metody), ale může být i white box (znalost vnitřního stavu testovaného objektu). Důležitost jednotkových testů spočívá v tom, že umožňují rychle a efektivně odhalit chyby v kódu a zlepšit jeho kvalitu již v rané fázi vývoje. Tím se snižuje riziko vzniku chyb a zvyšuje se spolehlivost celého systému. Jednotkové testy nelze ničím přímo nahradit, avšak jsou zde integrační testy, které testují interakce mezi jednotlivými částmi jako celek. Neposkytují však tak detailní výsledky jako testy jednotkové a najít zdroj chyby stojí více času. Výhodné je použití obou druhů testů, kde integrační testy jsou zde využity pro testování presenterů a současně i pro vykreslování šablon, které tento nástroj jinak testovat nedokáže.

Nette Tester je nejprve nutné do projektu nainstalovat, kdy minimální vyžadovaná verze PHP je v současnosti 7.1. Doporučený způsob instalace je za použití nástroje Composer. Nette Tester se nainstaluje tak, že se v adresáři projektu spustí příkazový řádek, ve kterém se spustí následující kód:

```
composer require --dev nette/tester
```

Zdrojový kód 1 – Instalace Nette Tester
Zdroj: [autor]

Po instalaci je možné hned začít psát testy a následně je spouštět a vyhodnocovat. Aby ale bylo vytváření testů co nejjednodušší a zamezilo se zbytečnému opakování stejného kódu při vytváření jednotlivých testů, je třeba přistoupit k této problematice komplexněji a vytvořit základní skripty a třídy, které lze dále využít v jednotlivých testech a ušetřit tak vývojářům čas, umožnit lepší orientaci v kódu, ale i zpětnou úpravu testů i od jiných vývojářů, kdy díky sjednocenému a ucelenému modelu budou testy udržitelné.

První soubor, který je třeba vytvořit by měl obsahovat základní PHP skripty pro načtení závislostí a nastavení prostředí, ve kterém se testy budou spouštět. Už z tohoto je zřejmé, že se jedná o skripty, které je nutné spouštět u každého skriptu, proto je zásadní tyto skripty oddělit a v každém testu pouze implementovat jako závislost. Pokud bude potřeba tyto skripty v budoucnosti modifikovat, bude stačit upravit pouze tento jeden soubor, nikoli všechny testovací soubory jednotlivě zvlášť.

Zdrojový kód 2 ukazuje soubor *bootstrap.php*, který plní funkci právě zmíněného počátečního souboru se skripty. Jak je vidět nejprve se načtou závislosti ze souboru *autoload.php* z adresáře *vendor*, který se nachází v adresáři aplikace a obsahuje závislosti knihoven, které byly nainstalovány pomocí Composeru a také samotného frameworku Nette. Druhý soubor *autoload.php,* je umístěn v adresáři, kde je i soubor *bootsrap.php.* Ten byl vytvořen pro načítání závislostí ze samotné aplikace a také pomocné soubory vytvořené pro testování – zde například trait *ConnectionT*, která bude popsána dále – a jeho zdrojový kód ukazuje Zdrojový kód 3 – autoload.php.

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

require __DIR__ . '/../vendor/autoload.php';
require __DIR__ . '/autoload.php';

Tester\Environment::setup();
date_default_timezone_set( timezoneId: 'Europe/Prague');


define('TMP_DIR', __DIR__ . '/temp');
@mkdir( directory: TMP_DIR);
```

Zdrojový kód 2 – Soubor bootstrap.php
Zdroj: [autor]

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

require __DIR__ . '/ConnectionT.php';
require __DIR__ . '/../app/model/Products.php';
require __DIR__ . '/../app/model/entity/Product.php';
```

Zdrojový kód 3 – Soubor autoload.php
Zdroj: [autor]

Dále je v *bootsrap.php* nastaveno testovací prostředí voláním statické funkce *setup()* třídy *Environment* a výchozí časová zóna. Závěrem je definována konstanta TMP_DIR, která odkazuje na umístění dočasného adresáře *temp* a tento adresář je následně vytvořen, pokud již neexistuje.

Jelikož testovaná aplikace pracuje s databází a tu, respektive metody a funkce, které ji využívají, je také důležité testovat, je před spouštěním těchto testů nutné vytvořit spojení s databází. Samozřejmostí je použití testovací databáze, nikoli produkční, aby nedošlo ke změně důležitých dat. Pro ještě větší bezpečnost je důležité vytvořit v databázi nového uživatele, který má přístup pouze k testovací databázi a nepoužívat přihlašovací údaje z produkční části aplikace. Pro usnadnění práce s databází a připojením k ní byla vytvořena trait *ConnectionT*. Trait je druh třídy

v PHP, která se nedědí při definici, ale použitím klausule USE ve třídě, kde má být využita. Používá se zejména pro použití metod ve více třídách, kde by klasická dědičnost byla zbytečně těžkopádná.

Jak ukazuje Zdrojový kód 4 – Trait ConnectionT, nejprve jsou definovány přihlašovací údaje k databázi a následně jsou zde dvě metody. První metoda *getConnection()*, která vrací instanci třídy *Connection,* nejprve vytvoří spojení s databází a následně se pokusí vytvořit strukturu databáze a nahrát do ní testovací data, skripty pro tyto úkony načítá z vytvořených SQL souborů. Na závěr vrací vytvořenou instanci třídy *Connection* s připojením k připravené databázi. Druhá metoda *getContext()*, vracející instanci třídy *Context*, umožňuje práci z databází na vyšší úrovni. V této metodě se na začátku vytvoří umístění pro dočasné soubory, kam si Nette Framework načítá data pro rychlejší práci z databází. V dalším kroku se vytvoří připojení pomocí předchozí metody *getConnection()* a následně inicializují dvě proměnné *structure* a *conventions*. Tyto dvě proměnné a proměnné *Connection* a *structure* se na závěr použijí k vytvoření instance třídy *Context*, který metoda v tomto kroku zároveň i vrací.

```php
trait ConnectionT
{
    private string $dsn = 'mysql:host=127.0.0.1;dbname=eshop_test';
    private string $user = 'test_user';
    private string $password = 'testpass';


    /**
     * @return Connection
     */
    public function getConnection(): Connection
    {
        $database = new Connection($this->dsn, $this->user,
            $this->password);

        try {
            Helpers::loadFromFile($database, __DIR__ . '/../../database.sql');

            Helpers::loadFromFile($database, __DIR__ . '/testing_data.sql');
        } catch (\Exception $ex) {
            echo $ex->getMessage();
        }

        return $database;
    }


    /**
     * @return Context
     */
    public function getContext(): Context
    {
        $storage = new FileStorage(TMP_DIR);
        $connection = $this->getConnection();
        $structure = new Structure($connection, $storage);
        $conventions = new DiscoveredConventions($structure);

        return new Context($connection, $structure, $conventions, storage);
    }
}
```

Zdrojový kód 4 – Trait ConnectionT
Zdroj: [autor]

I když je možné v Nette Tester psát jednotlivé testy jako obyčejné spustitelné PHP soubory, je zvolena cesta formou testovacích tříd, kdy testovací třída testuje jednu

testovanou třídu a umožňuje i další dále zmíněné výhody jako například dědičnost, použití vytvořené trait třídy a některé předpřipravené metody.

Základní vytvořenou testovací třídou je abstraktní třída *BaseTestCase*, která dědí ze třídy *TestCase.* Z této třídy dědí metody, které jsou zde využity, konkrétně *setUp()* a *tearDown()*. První je spouštěna před spuštěním testu a druhá po jeho provedení. Lze zde tedy nastavit testovací prostředí před samotným testem i po něm. V této třídě je využita trait *ConnectionT*, kdy právě v metodě *setUp()* dojde k připojení k databázi a získání instance třídy *Context*, přes kterou lze k databázi přistupovat, a poté k zahájení transakce, která je ukončena v metodě *tearDown()*. Transakce je zde využita pro vrácení změn v databázi po každém testu, aby byla konstantní a při vytváření nových testů bylo transparentní v jakém stavu databáze při spuštění testu bude.

V této třídě jsou definovány konstantní proměnné pro testování tříd a metod z CRUD operacemi, kdy je pro každou operaci definováno ID, pod kterým se v databázi nachází záznam a jedno ID, které je mimo rozsah a v databázi se pod ním žádný záznam nevyskytuje, aby bylo možné otestovat chování metod při pokusu o získání, úpravu nebo smazání neexistujícího záznamu a správné vyhození výjimky. V této abstraktní třídě je také ještě konstruktor, ve kterém se vytváří kontejner a *PresenterFactory*, třída, díky které je možné simulovat volání presenterů a zpracovávat jejich odpovědi, to umožňuje vytvářet integrační testování, ale i testování správného vykreslování šablon, jelikož je možné získat DOM objekt a dotazovat se na jednotlivé prvky, které mají být na vykreslené stránce obsaženy, případě dle různých atributů měněny a ověřit tím, že se aplikace chová dle očekávání. Zdrojový kód takovéto testovací třídy zaměřené na presenter s popisem jednotlivých částí a postupu vytváření bude následovat po testovacích třídách zaměřené na modelovou část aplikace.

```php
abstract class BaseTestCase extends TestCase
{
    use ConnectionT;

    const ID_GET = 1;
    const ID_UPDATE = 2;
    const ID_DELETE = 3;
    const ID_OUT_OF_RANGE = 99;

    /** @var Context */
    protected Context $db;

    /** @var Container */
    protected Container $container;

    /** @var PresenterFactory */
    protected PresenterFactory $presenterFactory;

    public function __construct()
    {
        $this->container = Bootstrap::bootForTests()->createContainer();
        $presenterFactory = $this->container->getByType('Nette\Application\IPresenterFactory');
        if ($presenterFactory instanceof PresenterFactory) {
            $this->presenterFactory = $presenterFactory;
        }
    }

    protected function setUp()
    {
        $this->db = $this->getContext();
        $this->db->beginTransaction();
    }

    protected function tearDown()
    {
        $this->db->rollBack();
    }
}
```

Zdrojový kód 5 - Třída BaseTestCase
Zdroj: [autor]

Poté co je vytvořena abstraktní třída *BaseTestCase,* která slouží jako základ pro další testovací třídy, je možné začít s vytvářením testovacích tříd pro jednotlivé části aplikace. Aplikace skládá ze 3 hlavních modulů (model, view, presenter) a dalších pomocných komponent. Tyto moduly je důležité testovat pro ověření správného fungování aplikace. Jelikož se od sebe jednotlivé třídy v těchto modulech funkčností zásadně neliší a z důvodu rozsáhlosti je zde není vhodné všechny uvádět, jsou uvedeny příklady vybraných tříd a postup, jak k nim vytvořit automatizované testy, které lze stejným způsobem implementovat na další třídy. Jako první byly zvoleny

k testování třídy modelu, které se starají o práci s databází a obsahují tak základní CRUD operace. To jsou vytváření, vypsání, upravování a smazání záznamu. Testovací třída, vytvořená pro tuto třídu, může obsahovat jednotlivé metody, které odrážejí metody modelové třídy. Tím lze dosáhnou přehlednosti a organizovanosti testů.

Jako ukázka je zde vybrána testovací třída odrážející základní modelovou třídu *Products.* V této třídě je deklarována proměnná *products* zmíněné modelové třídy *Products*. Ta je inicializována následně v metodě *setUp()* hned poté, co se provedou operace v rodičovské třídě. Dále již následují testovací metody pro jednotlivé metody z testované třídy. U těchto testovacích metod je důležité a nutné dodržet u názvu prefix „*test*“, aby framework správně rozeznal metody, které má spouštět jako testy, a dosáhlo se kýžených výsledků. Na konci souboru s testovací třídou je ještě nutné uvést příkaz *(new ProductsTest())->run().* Ten zajistí samotné vytvoření instance této třídy a spuštění testů.

První test, nebo také testovací metoda, je zde *testGetProducts(),* která testuje metodu *getProducts(),* třídy *Products.* Zde se testuje návratová hodnota metody *getProducts()*. Nejprve se otestuje, zda je proměnná, do které byla uložena návratová hodnota, typu *array* a jedná se tedy o pole prvků, což má být metodou ve korektním případě vráceno. Dále následuje otestování jednotlivých prvků v tomto poli, pokud nějaké obsahuje. Jestliže pole obsahuje nějaké prvky, musí být tyto prvky instancemi třídy *Product*. Pokud by tedy nebylo vráceno pole prvků a současně by se zde nacházel prvek jiného typu nežli *Product*, dojde k selhání tohoto testu.

Další testovací metoda je metoda *testCreateProduct().* Tato metoda testuje bez prefixu stejnojmennou metodu ze třídy *Products,* sloužící pro vytváření nových produktů. V této metodě je vytvořena instance třídy *Product*, která obsahuje testovací data – zde například název a cenu – a ta je předána metodě *createProduct()* třídy *Products*, kdy je očekávanou návratovou hodnotou booleanovská hodnota *true,* která se proto i testuje. V delším kroku je z databáze získán poslední záznam, je z něj

vytvořen instance třídy *Products* a metodou *Assert::same()* jsou tyto hodnoty porovnány s původními testovacími hodnotami, které měli být do databáze uloženy.

Třetí metoda, která testuje metodu na úpravu produktu, nese název *testUpdateProduct()* a obsahuje nejvíce testů jak na počet tak i typově. Stejně jako v předchozí metodě je zde vytvořena instance třídy *Product*, tentokrát je zde ale jako ID vložena hodnota z konstantní proměnné ID_UPDATE. Voláním metody *updateProduct()* je proveden pokus o změnu produktu v databázi a zároveň je otestována návratová hodnota této metody, která při správném fungování má být *true*. Tím je prověřena první část, následně je z databáze získán záznam s použitým ID a je ověřeno, zda byly hodnoty změněny porovnáním s původními. Nakonec je ještě proveden jeden test, který ověřuje správné chování v případě závadového vstupu, a to konkrétně při pokusu upravit záznam s neexistujícím ID. K tomuto je využita konstantní proměnná ID_OUT_OF_RANGE, kdy po pokusu o změnu je očekáváno vyhození výjimky, v tomto případě výjimky *FileNotFoundException*.

Poslední metoda *testDeleteProduct()* testuje odstranění záznamu z databáze. Metoda *deleteProduct()* přijímá jako parametr ID produktu a je zde tedy použito ID z konstantní proměnné ID_DELETE. Při správném fungování vrací tato testovaná metoda hodnotu *true* a na její vrácení je metoda testována. Samozřejmě je zde také ověřeno, že skutečně došlo k odstranění tohoto záznamu z databáze. K tomu je znovu využito testování na vyhození výjimky, při kterém je znovu proveden stejný příkaz na odstranění záznamu s ID_DELETE. Pokud byl záznam odstraněn, dojde k vyhození výjimky a je ověřeno, že metoda opravdu funguje, jak má. Aby se předešlo tomu, že k vyhození výjimky nedojde z nějakého jiného důvodu, než že by se záznam v databázi stále nacházel a vývojář by musel znovu provádět a vytvářet testy, aby odhalil skutečnou příčinu, je zde navíc doplněn test, kde je příkaz stejný jako před tím, ale ID je nahrazeno hodnotou ID_OUT_OF_RANGE, kdy v tomto případě musí dojít k vyhození výjimky v každém případě, pokud metoda funguje správně. Za zmínku ještě stojí, že v mnoha případech se mazání záznamů v databázi neprovádí pomocí SQL příkazu DELETE, ale pro zachování konzistence databáze a aplikace se pouze změní hodnota atributu, který značí, zda byla položka vymazána.

To by mohlo být testováno tak, že se získá z databáze předmětný záznam pomocí ID_DELETE a provede se otestování hodnoty příznaku, který značí vymazání.

```php
class ProductsTest extends BaseTestCase
{
    /** @var Products */
    private Products $products;


    protected function setUp()
    {
        parent::setUp();
        $this->products = new Products($this->db);
    }

    public function testGetProducts()
    {
        $products = $this->products->getProducts();
        Assert::type('array', $products);

        foreach ($products as $product) {
            Assert::type(Product::class, $product);
        }
    }

    public function testCreateProduct()
    {
        $product = new Product(0, "Nový produkt", 9900);
        Assert::true($products->createProduct($product));

        $lastRow = $this->db->table('products')->order('id DESC')->limit(1)
            ->fetch();

        if ($lastRow) {
            $lastProduct = new Product((int)$lastRow->id,
                (string)$lastRow->title, (float)$lastRow->price);

            Assert::same($product->getTitle(), $lastProduct->getTitle());
            Assert::same($product->getPrice(), $lastProduct->getPrice());
        }
    }

    public function testUpdateProduct()
    {
        $product = new Product(self::ID_UPDATE, "Upravený produkt", 5900);
        Assert::true($products->updateProduct($product));

        $row = $this->db->table('products')->get($product->getId());
        Assert::same($product->getTitle(), (string)$row->title);
        Assert::same($product->getPrice(), (float)$row->price);

        Assert::exception(function () use ($products) {
            $badProduct = new Product(self::ID_OUT_OF_RANGE, 'Produkt', 50);
            $products->updateProduct($badProduct);
        }, FileNotFoundException::class);
    }

    public function testDeleteProduct()
    {
        Assert::true($products->deleteProduct(self::ID_DELETE));
        Assert::exception(function () use ($products) {
            $products->deleteProduct(ID_DELETE);
            $products->deleteProduct(ID_OUT_OF_RANGE);
        }, FileNotFoundException::class);
    }
}

(new ProductsTest())->run();
```

Zdrojový kód 6 – TestCase třída ProductsTest
Zdroj: [autor]

Další podstatnou částí aplikace a zároveň poměrně konkrétní a jedinečnou je autentizace uživatelů. I proto je zde uvedena testovací třída pro tuto oblast. Aplikace využívá Nette Security, která má předpřipravený autentizátor. Ten lze pak dle potřeby upravit, aby splňoval požadavky aplikace. V testovací třídě je pouze jedna metoda. V této metodě se testuje ověření identity uživatele, dle záznamů v databázi a vytvoření jeho reprezentace pomocí *Nette\Security\User*. Nejprve se vytvoří instance této třídy *User* a poté se zavolá metoda *login()*, které se předá přihlašovací jméno (v tomto případě email) a heslo. Provede se ověření přihlašovacího jména v databázi a porovnání hash otisku hesla. Poté se otestuje metoda *isLoggedIn()* na proměnné *user*, zda je návratová hodnota *true* a uživatel je tedy přihlášen. Dále je otestováno správné vypsání atributů, a to konkrétně jména uživatele, čímž dojde k dalšímu ověření, že byl uživatel přihlášen a jeho údaje byly správně přiřazeny. Před dalšími testy je uživatel odhlášen metodou *logout().* Aby bylo testování kompletní, jsou ještě provedeny dva testy, na správné vyhození výjimky v případě, že je zadáno neexistující přihlašovací jméno nebo se neshoduje zadané heslo s uloženým. V reakci na toto se očekává výjimka *AuthenticationException*. Následující zdrojové kódy ukazují TestCase třídu *AuthenticatorTest* a také samotnou třídu *Authenticator* pro lepší porozumění. Autorizace uživatelských oprávnění není testována, jelikož se o tuto problematiku stará samotný Nette Framework.

```php
class AuthenticationTest extends BaseTestCase
{

    /** Test of authentication method */
    public function testAuthentication()
    {
        $user = $this->container->getByType('Nette\Security\User');
        $user->login('admin@eshop.cz', 'rightPass');

        Assert::true($user->isLoggedIn());
        Assert::same('Petr Novák', $user->getIdentity()->username);
        $user->logout();

        Assert::exception(function () use ($user) {
            $user->login('admin@eshop.cz', 'wrongPass');
        }, AuthenticationException::class);

        Assert::exception(function () use ($user) {
            $user->login('bad@email.com', 'password');
        }, AuthenticationException::class);
    }

}

(new AuthenticationTest())->run();
```

Zdrojový kód 7 – TestCase třída AuthenticationTest
Zdroj: [autor]

```php
class Authenticator implements NS\IAuthenticator
{
    /** @var Context */
    private Context $db;

    /** @var Users */
    private Users $user;

    public function __construct(Context $db, Users $user)
    {
        $this->db = $db;
        $this->user = $user;
    }

    function authenticate(array $credentials): IIdentity
    {
        list($email, $password) = $credentials;

        try {
            $user = $this->user->getUserByEmail($email);

            $passwords = new NS\Passwords(PASSWORD_BCRYPT, ['cost' => 12]);

            if (!$passwords->verify($password, $user->getPasswordHash())) {
                throw new \UnexpectedValueException();
            }

            return new NS\Identity($user->getId(), $user->getRoleId(),
                ['username' => $user->getFirstname() . ' ' . $user->getLastname()]);

        } catch (\Exception $ex) {
            throw new NS\AuthenticationException('Uživatel neexistuje,
            nebo bylo zadáno špatné heslo.');
        }
    }
}
```

Zdrojový kód 8 – Třída Authenticator
Zdroj: [autor]

Poslední využitý typ testu je integrační test. Ten je tady využit pro testování presenterů a jejich integrity. V kombinaci s tím jsou také testovány šablony na správné vykreslování, jelikož je při spuštění požadavku na presenter vrácena odpověď, ze které je možné získat vytvořený DOM. Jako příklad je zde uvedeno testování presenteru *ProductsPresenter*, kdy lze stejným způsobem testovat ostatní presentery a jejich metody. Na začátku testovací třídy *ProductsPresenterTest* je deklarována proměnná *productsPresenter* třídy *IPresenter* a následně je v metodě *startUp()* inicializována pomocí *presenterFactory->createPresenter('Products')*. Dále jsou zde dvě metody na otestování výchozí stránky v presenteru *Products* a to *testDefault()* a *testDefaultLogged()*. Jak již z názvu vyplývá, rozdíl v těchto metodách

je v tom, že v jedné je uživatel přihlášen a v druhé nikoli. Hlavní rozdíly se zde testují v rámci vykreslování šablony. V první metodě se testuje vykreslení nepřihlášeného uživatele. Nejprve se připraví požadavek, ve kterém se nastaví url cesta presenteru a HTTP metoda, zde tedy metoda GET. Poté se z tohoto požadavku za použití vytvořené proměnné *productsPresenter* získá odpověď, kde tímto dojde zároveň k otestování integrity metody presenteru, a z této odpovědi je získán DOM. Poté je možné provádět dílčí testy na jednotlivé části domu, zda jsou obsaženy. To mohou být například třídy, id, elementy a další. V tomto případě je nejprve, kdy se jedná o výchozí stránku s produkty, je testováno, zda je vykreslen element s třídou *productList* a je tak vidět výpis produktů a jelikož je zde kladen důraz na přihlášení uživatele je také otestováno, jestli jsou vykresleny elementy s třídou *not-logged* a naopak tu nejsou obsaženy třídy *logged*, protože uživatel se při tomto testu tváří jako nepřihlášený. Naopak ve druhé metodě je uživatel za použití validních přihlašovacích údajů přihlášen a až poté je vytvořen požadavek a získána odpověď. Tím, že došlo nejprve k přihlášení se metoda v presenteru provede tak, jako by byl klasicky uživatel přihlášen v aplikaci a dle jeho oprávnění dojde k vykreslení příslušného obsahu. Po uložení odpovědi do proměnné se získá objekt DOM stejně jako v předchozí metodě a provedou se testy vykreslených elementů s opačnými požadavky kdy elementy s třídou *not-logged* nemají být obsaženy a ty se třídou *logged* naopak vidět být mají.

Pro získání samotného objektu DOM je v testovací třídě vytvořena privátní metoda *getDom()*, která vrací objekt typu *DomQuery.* V této metodě jsou provedeny další dva testy. První test kontroluje zda je poskytnutá odpověď pomocí parametru typu *TextResponse* a druhý test, zda návratová hodnota metody *getSource(),* poskytnuté odpovědi typu *TextResponse*, je instancí třídy *Template*. Tato návratová hodnota je poté přetypována na řetězec a ten uložen do proměnné. Ta je následně zpracována statickou metodou *fromHtml()* třídy *DomQuery* a vrácena jako výsledek metody *getDom()*.

```php
class ProductsPresenterTest extends BaseTestCase
{
    /** @var IPresenter */
    private IPresenter $productsPresenter;

    protected function setUp()
    {
        parent::setUp();
        $this->productsPresenter =
            $this->presenterFactory->createPresenter('Products');

        $this->productsPresenter->autoCanonicalize = false;
    }

    /** Test of default render method - not logged in */
    public function testDefault()
    {
        $request = new Request('Products', 'GET');
        $response = $this->productsPresenter->run($request);

        $dom = $this->getDom($response);

        Assert::true($dom->has('.productList'));
        Assert::true($dom->has('.not-logged'));
        Assert::false($dom->has('.logged'));
    }
        $user = $this->container->getByType('Nette\Security\User');
        $user->login('admin@eshop.cz', 'rightPass');
        $request = new Request('Products', 'GET');
        $response = $this->productsPresenter->run($request);

        $dom = $this->getDom($response);

        Assert::true($dom->has('.logged'));
        Assert::false($dom->has('.not-logged'));
    }

    /** Returns instance of DomQuery from IResponce
     * @param IResponce $response
     * @return DomQuery
     */
    private function getDom(IResponse $response): DomQuery
    {
        Assert::type(TextResponse::class, $response);
        Assert::type(Template::class, $response->getSource());

        $html = (string)$response->getSource();
        return @DomQuery::fromHtml($html);
    }
}
(new ProductsPresenterTest())->run();
```

Zdrojový kód 9 – Test Products presenteru
Zdroj: [autor]

8.3.2 Selenium IDE

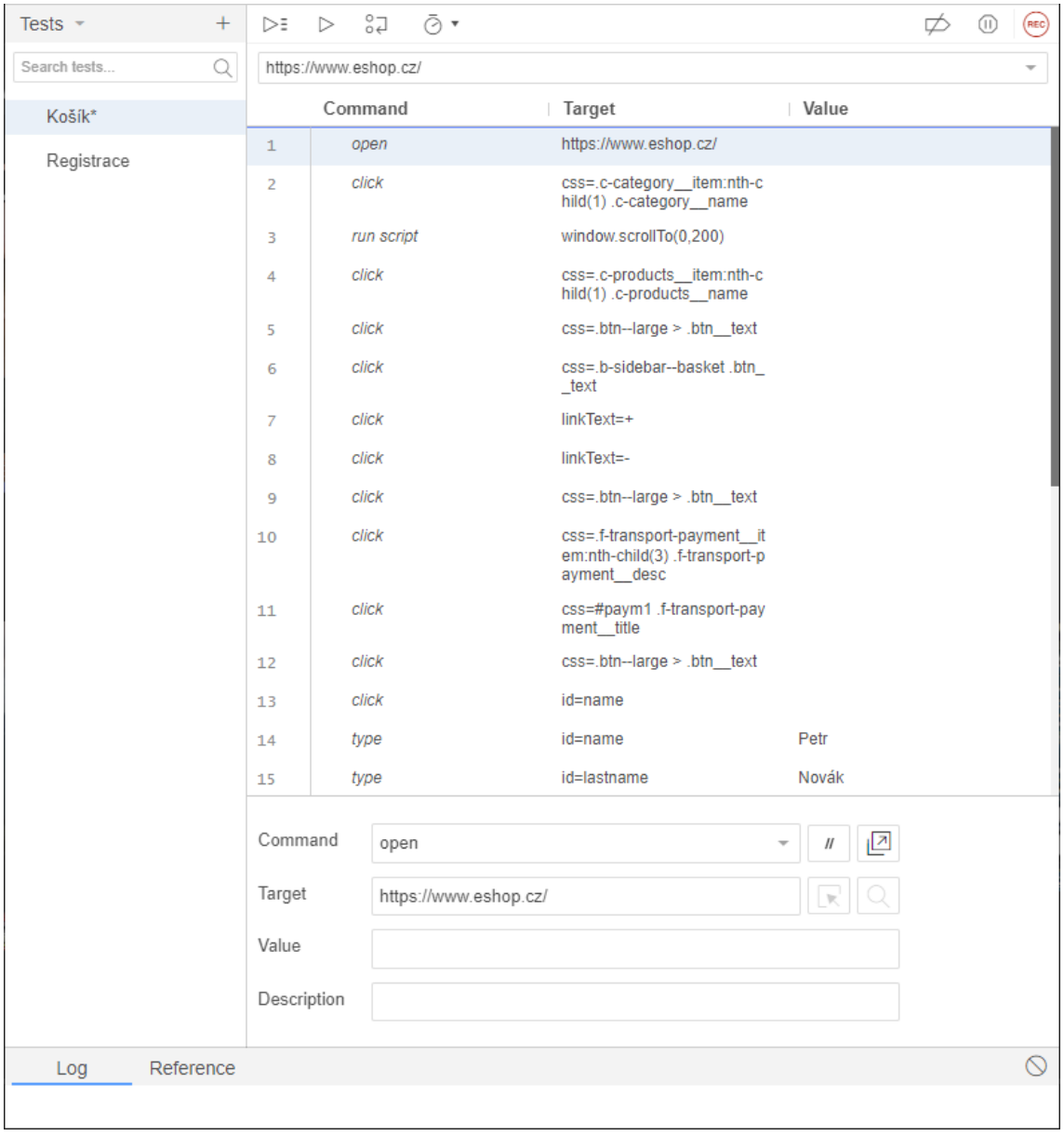
Další oblastí testování webových aplikací je testování uživatelského rozhraní a interakcí s aplikací, tedy funkční testování, kde je využívána metodika black box testování, jelikož není známé, jak se aplikace chová uvnitř. K tomuto byl vybrán nástroj Selenium IDE, spouštěný pomocí internetového prohlížeče Google Chrome, ve kterém se dále provádí základní testy uživatelského chování.

Nejběžnější interakce, které se pomocí Selenium provádí, jsou klikání na tlačítka a odkazy – testování funkcionalit těchto elementů – a vyplňování formulářů. Zároveň je dobré se nejprve zaměřit na procesy, které uživatelé dělají nejčastěji nebo jsou klíčové pro dosažení požadovaného chování uživatelů – např. vložení zboží do košíku, provedení objednávky a s tím související interakce (výběr platby a dopravy, vyplnění fakturačních údajů) a odeslání objednávky. Následující postupy vytváření testů lze dále uplatnit i na další interakce.

Prvním testovaným chováním je nákup zboží. To začíná vstoupením na úvodní stránku e-shopu, poté uživatel přejde na příslušnou kategorii zboží, které ho zajímá, otevře si detail vybraného produktu a ten si poté přidá do košíku. Dále uživatel přejde k obsahu svého košíku, kde se zobrazuje vybraný produkt a pokračuje dále procesem dokončení objednávky. Nejprve je vyplněna doprava a způsob platby a v následujícím kroku uživatel vyplňuje fakturační údaje pro dokončení objednávky. Po odsouhlasení obchodních podmínek je následně vytvořena objednávka. Tento celý proces byl nahrán pomocí nástroje Selenium IDE, kdy Obrázek 1 ukazuje, jak vypadají jednotlivé kroky zachycené tímto nástrojem.

Tento test byl pro znázornění a případnou budoucí práci s tímto testem převeden do zdrojového kódu v jazyce Java. Ten zobrazuje Zdrojový kód 10, kde je vidět třída KosikTest, která má své atributy *WebDriver, Map<String, Object> a JavascriptExecutor*. Dále obsahuje dvě metody, první *setUp()* se vykonává před samotným testem a druhá *tearDown()* po provedení testu. V první metodě se nastaví WebDriver, který je v tomto případě instancí třídy *ChromeDriver*,

JavascriptExecutor a instance *HashMap* do proměnné typu *Map*. Ve druhé zmíněné metodě dojde pouze k ukončení WebDriveru. Poslední metoda je samotný test se všemi příkazy, tak jak je tomu i v grafické podobě testu. Je zde vidět, že nejprve dochází k vyhledání prvku pomocí tříd nebo id HTML elementů a následné interakci s nimi ve formě provedení kliknutí nebo doplnění textu. Na závěr, po provedení všech požadovaných operací dojde k uzavření WebDriveru.



Obrázek 1 – Selenium IDE – nákupní košík
Zdroj: [autor]

```
public class KosikTest {
  private WebDriver driver;
  private Map<String, Object> vars;
  JavascriptExecutor js;
  @Before
  public void setUp() {
    driver = new ChromeDriver();
    js = (JavascriptExecutor) driver;
    vars = new HashMap<String, Object>();
  }
  @After
  public void tearDown() {
    driver.quit();
  }
  @Test
  public void kosik() {
    driver.get("https://www.eshop.cz/");
    driver.findElement(By.cssSelector(".c-category__item:nth-child(1) .c-category__name")).click();
    js.executeScript("window.scrollTo(0,200)");
    driver.findElement(By.cssSelector(".c-products__item:nth-child(1) .c-products__name")).click();
    driver.findElement(By.cssSelector(".btn--large > .btn__text")).click();
    driver.findElement(By.cssSelector(".b-sidebar--basket .btn__text")).click();
    driver.findElement(By.linkText("+")).click();
    driver.findElement(By.linkText("-")).click();
    driver.findElement(By.cssSelector(".btn--large > .btn__text")).click();
    driver.findElement(By.cssSelector(".f-transport-payment__item:nth-child(3) .f-transport-payment__desc")).click();
    driver.findElement(By.cssSelector("#payml .f-transport-payment__title")).click();
    driver.findElement(By.cssSelector(".btn--large > .btn__text")).click();
    driver.findElement(By.id("name")).click();
    driver.findElement(By.id("name")).sendKeys("Petr");
    driver.findElement(By.id("lastname")).sendKeys("Novák");
    driver.findElement(By.id("phone")).sendKeys("777555444");
    driver.findElement(By.id("email")).sendKeys("testt@gmail.com");
    driver.findElement(By.id("street")).sendKeys("Dlouhá 944");
    driver.findElement(By.id("city")).sendKeys("Hradec Králové");
    driver.findElement(By.id("zip")).sendKeys("50002");
    driver.findElement(By.cssSelector("#submit-form-button > .btn__text")).click();
    assertThat(driver.switchTo().alert().getText(), is("Pro pokračování je nutné odsouhlasit obchodní podmínky."));
    driver.findElement(By.cssSelector(".b-steps__item:nth-child(1) .b-steps__name")).click();
    driver.findElement(By.cssSelector(".b-basket-table__product")).click();
    driver.findElement(By.linkText("-")).click();
    driver.close();
  }
}
```

Zdrojový kód 10 – Selenium KosikTest
Zdroj: [autor]

Druhým testem je přihlášení uživatele a zobrazení jeho účtu. To zachycuje Zdrojový kód 11. Je zde uveden pouze zdrojový kód, jelikož podoba v grafické verzi Selenium IDE lze z tohoto snadno odvodit a je podobná ukázce výše. Třída tohoto testu obsahuje stejné metody jako u testu nákupu. Je zde vyvoláno přihlašovací modální okno, do kterého jsou vyplněny přihlašovací údaje. Poté je pro ověření úspěšného přihlášení přejito na stránku uživatelského profilu.

Takto lze vytvářet další testy pro různé interakce a chování uživatele, které by se jinak musely testovat pokaždé ručně, což by zabralo spoustu času a dalších prostředků.

```
public class PrihlaseniTest {
  private WebDriver driver;
  private Map<String, Object> vars;
  JavascriptExecutor js;
  @Before
  public void setUp() {
    driver = new ChromeDriver();
    js = (JavascriptExecutor) driver;
    vars = new HashMap<String, Object>();
  }
  @After
  public void tearDown() {
    driver.quit();
  }
  @Test
  public void prihlaseni() {
    driver.get("https://www.eshop.cz/");
    driver.findElement(By.cssSelector(".b-profile__name:nth-child(2)")).click();
    driver.findElement(By.id("login-name")).click();
    driver.findElement(By.id("login-name")).sendKeys("petrnovak");
    driver.findElement(By.id("login-password")).sendKeys("tajneHeslo123");
    driver.findElement(By.cssSelector(".btn--icon-left > .btn__text")).click();
    driver.findElement(By.cssSelector(".b-profile__name:nth-child(3)")).click();
    driver.findElement(By.linkText("Můj profil")).click();
    driver.close();
  }
}
```

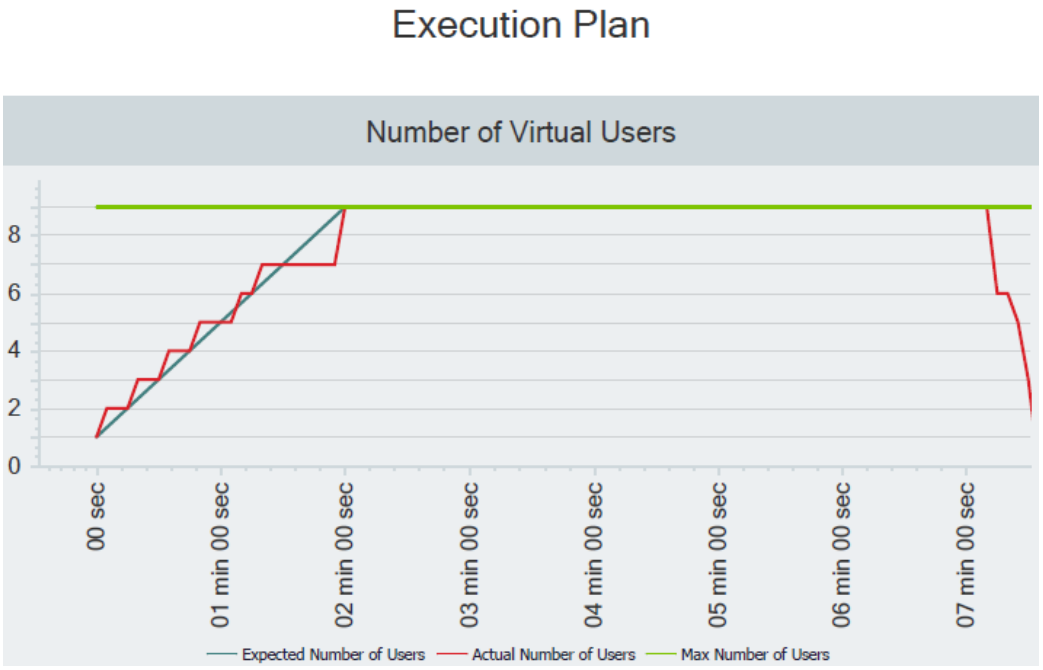
Zdrojový kód 11 – Selenium prihlaseni
Zdroj: [autor]

8.3.3 LoadView

Nástroj LoadView umožňuje provádět více druhů testování včetně podobného jaké dokáže Selenium IDE. Byl ale zvolen pro testování výkonu aplikace a zátěže serveru. K tomuto byl zvolen nabízený modul Stress Testing, který nabízí vytvoření a konfiguraci přístupů na zvolenou URL adresu a simulaci několika současných přístupů, následně jsou k tomuto vytvořeny grafy s výsledky.

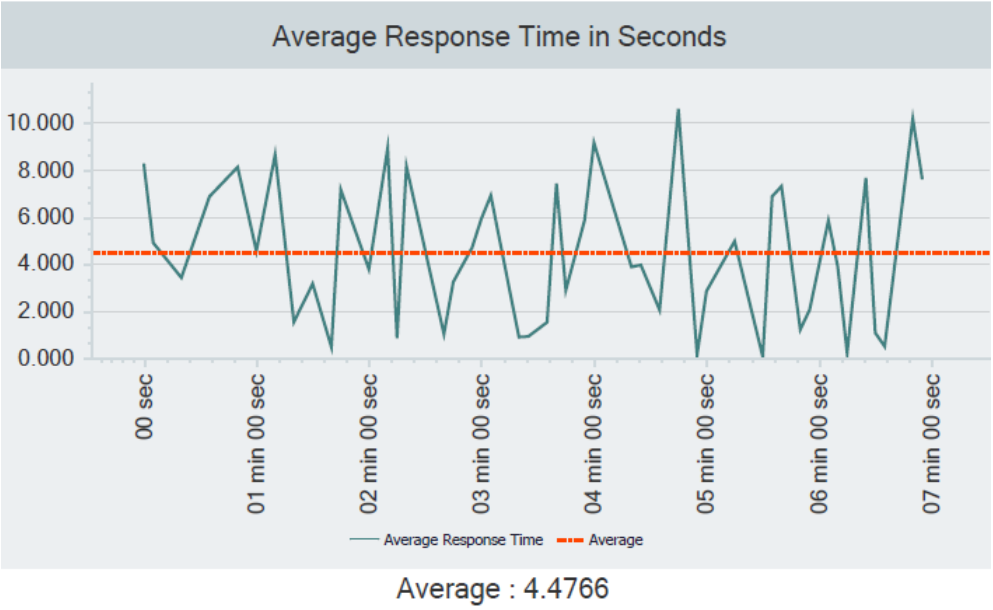
Pro první test byla zvolena úvodní stránka aplikace, na kterou jsou směřovány přístupy. V rámci zkušební verze zdarma je k dispozici maximálně 10 připojených uživatelů současně a maximálně 2 cloudové servery, ze kterých se simuluje připojení těchto uživatelů. Proto byl test nastaven tak, že se nejprve připojí jeden uživatel a následně se po první minutě připojí další 4 uživatelé a v další minutě další 4. Poté je těchto celkem 9 uživatelů drženo aktivně na stránkách po dobu 5 minut. Na obrázku Obrázek 2 je graf průběhu testování, který znázorňuje počet připojených

uživatelů v závislosti na čase. Je zde tedy vidět, jak se uživatelé postupně v prvních dvou minutách připojují a zůstávají připojeni až do konce testu.



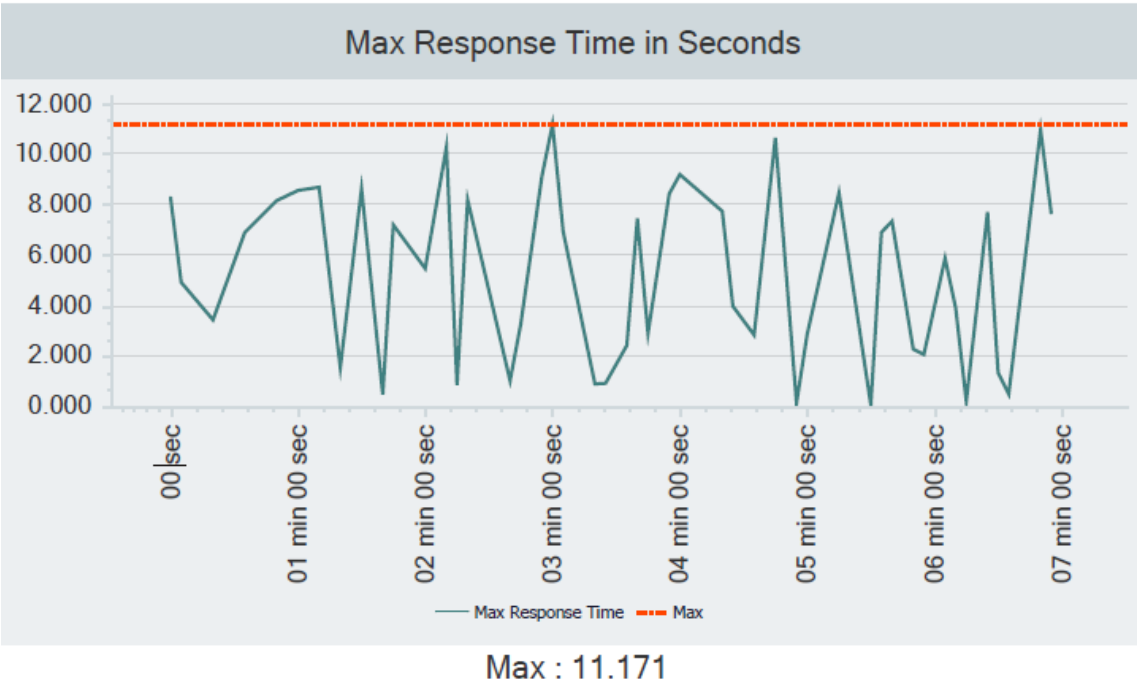
Obrázek 2 – LoadView – plán testu
Zdroj: [autor]

Další graf, který zachycuje Obrázek 3, je v závislosti na čase zobrazena průměrná a reálná doba odezvy. Jak je vidět reálná doba odezvy poměrně osciluje kolem průměru, který je vypočten na 4,4766 sekund.



Obrázek 3 – LoadView – průměrná odezva
Zdroj: [autor]

Obrázek 4 naopak ukazuje maximální dobu odezvy a zároveň znovu reálnou odezvu v závislosti na čase. Uvedená maximální doba odezvy je zde 11,171 sekundy.



Obrázek 4 – LoadView – Maximální doba odezvy
Zdroj: [autor]

Další grafy, které jsou součástí výsledku, ukazují počet vytvořených sessions (sezení) v čase a celkový kumulativní počet sessions v závislosti na čase, kdy se v tomto případě celkový počet dostal na hodnotu 70. Také je zde graf s počtem chyb v čase, který je zde prázdný a poslední graf ukazuje procenta zatížení procesorů jednotlivých serverů, na kterých byly spouštěni virtuální uživatelé.

Je tedy zřejmé, že webová aplikace na tom není výkonnostně úplně nejlépe a některé uživatele může doba odezvy větší než pár sekund odradit od prohlížení webu a v tomto případě i od potenciálního nákupu nabízeného zboží. Je tedy důležité nepodceňovat výkonnostní stránku webových aplikací, a i na toto testování se v průběhu vývoje zaměřit.

9 Shrnutí výsledků

V praktické části této práce byla provedena implementace nástrojů pro různé typy testování. Prvním z nich byl nástroj Nette Tester, který byl použit pro jednotkové a integrační testování. Druhým nástrojem bylo Selenium IDE, což je nástroj pro automatizované testování webových aplikací. A konečně byl implementován nástroj na testování výkonnosti LoadView.

Cílem práce bylo implementovat automatizované testování k již fungujícímu e-shopu a zlepšit tak jeho kvalitu. K tomuto bylo využito několika nástrojů, pro jednotlivé oblasti. Výsledky práce ukázaly, že implementace těchto nástrojů byla úspěšná a jejich použití vedlo k výraznému zlepšení testování e-shopu.

Implementace nástroje Nette Tester umožnila jednoduché a efektivní testování jednotlivých komponent e-shopu, jako jsou například presentery a modely. Díky tomu bylo možné rychle odhalovat a opravovat chyby a zlepšovat tak celkovou kvalitu e-shopu. Použití tohoto nástroje bylo časově nejnáročnější, neboť je třeba napsat jednotlivé testy ručně, což obzvláště u takto rozsáhlé aplikace trvá i několik dní. Následně ale dojde k velké úspoře času, které dosud bylo vynaloženo na testování, neboť spouštění testů je prováděno automaticky v řádu několika sekund, maximálně minut (v závislosti na výkonu hardwaru) a pokud jsou další testy vytvářeny souběžně s novými funkcemi, je časová náročnost velmi malá. Díky tomuto nástroji bylo odhaleno několik drobnějších chyb, které vznikly zastaralostí některých částí kódu a také velmi závažná chyba, kdy byl nefunkční modul v administraci e-shopu, na který se dosud kvůli nízkému přímému používání nepřišlo, ale z dlouhodobého hlediska mohl být dopad na funkčnost objednávek kritický.

Tento nástroj umožnil také částečně rozpoznání zastaralých testů. Nerozpozná přímo zastaralá testy, ale umožňuje zobrazit upozornění na testy, které selhaly nebo jsou napsány v zastaralém kódu – například ve starší verzi Nette frameworku se zastaralými funkcemi nebo metodami. Kromě toho lze použít další nástroje nebo

pluginy spolu s Nette Testerem, která mají schopnost identifikovat zastaralý kód v testech, například statický analyzátor kódu PHPStan nebo Psalm, který lze s Nette Testerem integrovat.

Selenium IDE bylo použito pro testování webového rozhraní e-shopu. Použití tohoto nástroje není nijak časově náročné, neboť umožňuje nahrávat běžnou práci s aplikací, při manuálním testování nových funkcionalit, kdy razantně urychlí budoucí testování, jelikož umožní automaticky spouštěn již uložené testy a otestovat tak velkou část aplikace, která by při ručním testování mohla být opomenuta. Jeho implementace umožnila automatizovat testování různých scénářů, jako jsou například přihlašování uživatelů, přidávání produktů do košíku a dokončování objednávek. Díky tomu bylo možné rychle odhalovat a opravovat chyby a zlepšovat tak uživatelskou zkušenost. Při výběru interakcí, které testovat je dobré se zaměřit na scénáře, které uživatelé často nebo opakovaně provádějí a také na ty, které jsou klíčové pro fungování aplikace a které umožňují dosažení požadovaných cílů, které má aplikace poskytovat (v tomto případě uskutečnění nákupu od nalezení produktu, vložení do košíku, po vyplnění formulářů a vytvoření objednávky).

Nástroj LoadView byl použit pro testování výkonnosti a stability e-shopu. Jeho implementace umožnila simulovat návštěvnost e-shopu v reálném čase a zjistit tak, jak rychle e-shop odpovídá na požadavky uživatelů. Je tak možné odhalit a zaměřit se na konkrétní částí webové aplikace, které mají problém s výkonem a doba odezvy může odradit zákazníky a zaměřit se na optimalizaci procesů, které při načítání probíhají. Díky tomu bylo možné zlepšit rychlost e-shopu a tím zlepšit uživatelskou zkušenost, kdy bylo odhaleno, že se příliš dlouho načítá seznam produktů při použití většího množství filtrů najednou, kdy byla na základě tohoto provedena optimalizace této funkcionality a došlo ke zlepšení výkonu.

Celkově lze tedy říci, že implementace těchto tří nástrojů umožnila automatizovat testování e-shopu a zlepšit tak jeho kvalitu a výkonnost, což se může odrážet i na počtu realizovaných prodejů.

10 Závěry a doporučení

V teoretické části práce byl čtenář seznámen s principy testování softwaru a následně byla uvedena problematika automatizovaného testování s důrazem na strategie a metodologii automatizovaného testování.

Teoretická část práce uvádí, že testování softwaru je kritickou fází životního cyklu vývoje softwaru. Odpovídá závěrečné kontrole specifikace, návrhu a kódování. Hlavním cílem návrhu testovacích případů je odvodit sadu testů, které mohou odhalit chyby softwaru. Důkladně otestovaný software udržuje kvalitu. Kvalitní produkt je cílem všech vývojářů a společností. Testování softwaru také poskytuje objektivní a nezávislý pohled na software a umožňuje podnikům posoudit a pochopit rizika spojená s implementací softwaru. Testovací techniky zahrnují mimo jiné proces spouštění programu nebo aplikace za účelem nalezení softwarových chyb. Rozsah testování softwaru často zahrnuje ověřování kódu a provádění tohoto kódu v různých prostředích a podmínkách, jakož i ověřování aspektů kódu, tj. zda dělá to, co má dělat nebo by měl dělat. V současné kultuře vývoje softwaru může být organizace zabývající se testováním oddělena od vývojového týmu, kde členové testovacího týmu mají různé role.

Cílem této práce bylo implementovat automatizované testování do funkčního e-shopu a usnadnit tak práci vývojářům a ušetřit jejich čas. Toho bylo dosaženo v praktické části implementací uvedených nástrojů a byla tak pokryta značná část testovacích oblastí. Jedná se o oblasti jednotkových testů, integračních testů, funkčního testování, testování uživatelského rozhraní a výkonnosti a stability aplikace, kdy byly tyto oblasti specifikovány v zadání vývojáři této webové aplikace. V této práci nebylo ovšem možné obsáhnout implementaci všech dostupných nástrojů pokrývajících další možné oblasti testování webové aplikace. Tyto oblasti, které by bylo možné pokrýt automatizovanými testy, jsou například testy zabezpečení softwaru a ověřování, že ochrana dat a soukromí uživatelů je zajištěna, nebo testy přijatelnosti, tedy, zda je software jednoduše použitelný pro uživatele. Problematika těchto oblastí by mohla být zkoumána v dalších odborných pracích.

Další oblastí, která by mohla být prozkoumána je integrace nástrojů nebo pluginů, které se zabývají analýzou kódu a hledáním zastaralých testů, s nástrojem Nette Tester.

Také je třeba uvést, že tato práce má za cíl zejména nastínit postup při automatizaci testování a uvádí specifické příklady, ze kterých lze dále čerpat při vytváření dalších dílčích testů v dané aplikaci a čtenářům může sloužit jako návod při vytváření vlastních automatizovaných testů do jejich aplikací. Proto práce neobsahuje veškeré testy, které bylo třeba vytvořit pro plnohodnotné automatizování zde užité webové aplikace. Zároveň existuje řada dalších nástrojů, které dokáží danou problematiku řešit a lze je zvolit místo zde použitých. Výběr konkrétních nástrojů vždy záleží na dané aplikaci a požadavcích, které mají splňovat. Je třeba brát v úvahu jazyk, ve kterém je aplikace vyvíjena, zkušenosti vývojářů či testerů s jednotlivými nástroji a možnosti těchto nástrojů. Dalším faktorem může být i licence a zpoplatnění nástrojů.

11 Seznam použité literatury

AKTAŞ, A. Z., YAĞDERELİ, E., & SERDAROĞLU, D. (2021). An introduction to software testing methodologies. V *Gazi University Journal of Science Part A: Engineering and Innovation* (stránky 1-15).

ALPER, M. (1994). Professional Software Testing: Practice in Optimizing the Quality of Commercial Software.

ARUMUGAM, A. K. (2019). Software Testing Techniques & New Trends. V *International Journal of Engineering Research & Technology.*

BATH, G., & MCKAY, J. (2010). Software Testing Experience - Test Analyst and Technical Test Analyst. Heidelberg.

BÉRARD, B., BIDOIT, M., FINKEL, A., LAROUSSINIE, F., PETIT, A., PETRUCCI, L., . . . MCKENZIE, P. (2001). Systems and Software Verification: Model-Checking Techniques and Tools. Heidelberg: Springer.

BLACK, R. (2009). Managing the Testing Process. Indianapolis.

BOOCH, G., REMBAUGH, J., & JACOBSON, I. (2005). The Unified Modeling Language User Guide, 2nd end. Boston.

BROWN, W., MALVEAU, R., MCCORMICK, H., & MOWBRAY, T. (1998). Anti-Patterns: Refactoring Software, Architectures and Projects in Crisis. New York.

CARTER, A. (21. 1 2018). *PHPUnit: What, Why, How?* (Sheffield Based Web Developer) Získáno 12. 2 2023, z https://andy-carter.com/blog/phpunit-what-why-how

CLEFF, T. (2010). Basiswissen Testen von Software. W3L GmbH.

FAGAN, M. (1976). Design and code inspections to reduce errors in program development. IBM.

FAHAD, M., & NADEEM, A. (2008). A Survey of UML Based Regression Testing. V *Intelligent Information Processing IV. - The International Federation for Information Procesing* (stránky 200-210). Boston: Springer.

FOWLER, M. (1999). Refactoring: Improving the Design of Existing Code. Boston.

GAMMA, E., HELM, R., JOHNSON, R., & VLISSIDES, J. (1995). Design patterns. Elements of Refactoring Software, Architectures and Projects in Crisis. Munich.

GEORGIEVA, K. (2010). Conducting FMEA over the software development process. Software Eng. Notes.

GRUDL, D. (7. 4 2014). *Proč používám Nette Tester*. (phpFashion) Získáno 15. 2 2023, z https://phpfashion.com/proc-pouzivam-nette-tester

GRUDL, D. (2023). *Začínáme s Nette Tester*. (Nette Tester) Získáno 10. 2 2023, z https://tester.nette.org/cs/guide

HENNELL, M., WOODWARD, M., & HEDLEY, D. (1976). On program analysis.

CHAUHAN, R., & SINGH, I. (2014). Latest Research and Development on Software Testing Techniques and Tools. V *International Journal of Current Engineering and Technology.*

IEEE Standard Glossary of Software Engineering Terminology (HyperCard Stack). (1990). New York: IEEE, The Institute of Electrical and Electronics Engineers, Inc.

IEEE Standard Glossary of Software Engineering Terminology. (1990). New York: IEEE, The Institute of Electrical and Electronics Engineers, Inc.

JAN, S., NGUYEN, C. D., & BRIAND, L. C. (2016). Automated and effective testing of web services for XML injection attacks. V *Proceedings of the 25th International Symposium on Software Testing and Analysis* (stránky 12-23).

KAUR, M., & KUMARI, R. (2011). Comparative study of automated testing tools: Testcomplete and quicktest pro. International Journal of Computer Applications.

KELLERWESSEL, H. (2002). Programming Guidlines in practice. Bonn.

KERIEVSKY, J. (2004). Refactoring to Patterns. London: Pearson Higher Education.

KOENIG, A. (1995). Patterns and antipatterns. JOOP.

KOŁTUN, A., & PAŃCZYK, B. (2020). Comparative analysis of web application performance testing tools. V *Journal of Computer Sciences Institute* (stránky 351-357).

KOREL, B., & LASKI, J. (1988). Dynamic program slicing.

KRISHNA, V. V., & GOPINATH, G. (2021). Test Automation of Web Application Login Page by Using Selenium Ide in a Web Browser. V *Management* (stránky 713-732).

LEE, G. (7. 2 2023). *LoadView Hands-On Review Tutorial: Load Testing From The Cloud*. (Software Testing Help) Získáno 11. 3 2023, z https://www.softwaretestinghelp.com/loadview-tutorial/

LEVENTHAL, L., & BARNES, J. (2007). Usability Engineering. V *Process,Products and Examples.* Upper Saddle River: Prentice Hall.

LIGGESMEYER, P. (2009). Testing, Analyzing and Verifying Software. Berlin.

MACHARLA, P. (2017). Working with Appium. V *Android Continuous Integration: Build-Deploy-Test Automation for Android Mobile Apps* (stránky 95-115).

MAJCHRZAK, T. (2010). Improving the technical aspects of software testing in enterprises.

MAJCHRZAK, T., & KUCHEN, H. (2009). Automated test case generation based on coverage analysis. V *TASE '09: Proceedings of the 2009 3rd IEEE International Symposium on Theoretical Aspects of Software Engineering* (stránky 259-266). Washington, DC: IEEE Computer Society.

MARIANI, L., HAO, D., SUBRAMANYAN, R., & ZHU, H. (2017). The central role of test automation in software quality assurance. V *Software Quality Journal* (stránky 797-802).

MEYER, B. (2008). Seven principles of software testing.

MYERS, G., & SANDLER, C. (2004). The Art of Software Testing. Chichester.

NASSI, I., & SHNEIDERMAN, B. (1973). Flowchart techniques for structured programming. SIGPLAN Notices.

NEUFELDER, A. (1992). Ensuring Software Reliability. New York: Marcel Dekker.

OLAN, M. (19. 2 2003). Unit testing: test early, test often. Journal of Computing Sciences in Colleges. Získáno 3. 2 2023

OLIINYK, B., & OLEKSIUK, V. (2019). Automation in software testing, can we automate anything we want. V *Proceedings of the 2nd Student Workshop on Computer Science & Software Engineering* (stránky 224-234). Ukraine.

OPDYKE, W., & JOHNSON, R. (1990). Refactoring: an aid in designing application frameworks and evolving object-oriented systems. V *Proceedings of the Symposium on Object Oriented Programming Emphasizing Practical Applications.*

Oracle. (nedatováno). *How to write Doc comments for the Javadoc tool*. Získáno 20. 3 2023, z https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html

PERRY, W. (2003). Software Testing. Bonn.

PEZZE, M., & YOUNG, M. (2007). Software Testing and Analysis. V *Process, Principles and Techniques.* New York.

PODGURSKI, A., & YANG, C. (1993). Partition testing, stratified sampling and cluster analysis. SIGSOFT Softw. Eng. Notes.

POL, M., KOOMEN, T., & SPILLNER, A. (nedatováno). A Practical Guide for Successful Software Testing with TPI and Tmap. V *Management and Optimalization of the Test Process.* 2002: Heidelberg.

RAMYA, P., SINDHURA, V., & SAGAR, P. V. (2017). Testing using selenium web driver. V *Second International Conference on Electrical, Computer and Communication Technologies (ICECCT)* (stránky 1-7). IEEE.

RICHARDSON, D., & CLARKE, L. (1985). Partition analysis: a method combining testing and verification. IEEE Trans. Softw. Eng.

RODRIGUEZ, C. (2022). *Testim Review: Coded or Codeless Test Automation*. (Software Testing Company - Leading QA Services) Získáno 15. 2 2023, z https://abstracta.us/blog/test-automation/testim-review-ai-automated-functional-testing-tool/

ROITZSCH, E. (2005). Analytical software quality assurance in theory and practice: The way to high-quality software through static testing, dynamic testing and formal proof.

RUBIN, J. C. (2008). Handbook of Usability Testing. V *How to Plan, Design and Conduct Effective Tests.* Hoboken.

SEARS, A., & JACKO, J. (2008). Fundamentals, Evolving Technologies and Emerging Applications. V *The Human-Computer Interaction Handbook.* Boca Raton: CRC Press.

SHARMA, S. (2012). Study and Analysis of Automation Testing Techniques. V *Journal of Global Research in Computer Science.*

SNEED, H., & WINTER, M. (2002). Testing of Object-Oriented Software. Munich.

SNEHA, K., & MALLE, G. (2017). Research on software testing techniques and software automation testing tools. V *International Conference on Energy, Communication, Data Analytics and Soft Computing (ICECDS).*

Software Quality: Testing, analyzing and Verifying Software. (2009). Berlin.

SPILLNER, A., ROSNER, T., WINTER, M., & LINZ, T. (2008). Practical knowledge of software testing - Test Management. Heidelberg.

SRIVASTAVA, N., KUMAR, U., & SINGH, P. (2021). Software and performance testing tools. V *Journal of Informatics Electrical and Electronics Engineering (JIEEE)* (stránky 1-12).

SUPRIYONO, S. (2020). Software testing with the approach of blackbox testing on the academic information system. V *International Journal of Information System and Technology* (stránky 227-233).

SUTTER, H., & ALEXANDRESCU, A. (2004). 101 Rules, Guidelines and Best Practices. V *C++ Coding Standards.* Boston.

Testim.Io Tutorial: Speed-Up Authoring And Executino Of Automated Tests. (25. 1 2023). (Software Testing Help) Získáno 11. 2 2023, z https://www.softwaretestinghelp.com/testim-io-tool-tutorial/

THALLER, G. (2002). Software-Test, 2nd edn. Hannover.

UMAR, M. (2019). Comprehensive study of software testing: Categories, levels, techniques and types. V *International Journal of Advance Research, Ideas and Innovations in Technology* (stránky 32-40).

UPPAL, N., & CHOPRA, V. (2012). Design and implementation in selenium ide with web driver. V *International Journal of Computer Application* (stránky 8-11).

VILA, E., NOVAKOVA, G., & TODOROVA, D. (2017). Automation testing framework for web applications with selenium webdriver: Opportunities and threats. V *Proceedings of the International Conference on Advances in Image Processing* (stránky 144-150).

WALLMÜLLER, E. (2001). Software Quality management in practice. Munich.

WATKINS, J. (2001). Testing IT: An Off-the-Shelf Software Testing Process. New York: Cambridge University Press.

WEISER, M. (1981). Program slicing. V *Proceedings of the 5th International Conference on Software Engineering, ISCE* (stránky 439-449). Piscataway: IEE Press.

What Is Selenium IDE? + 7 Important Selenium IDE Tips. (13. 12 2021). (BlazeMeter Continuous Testing | BlazeMeter by Perforce) Získáno 10. 3 2023, z https://www.blazemeter.com/blog/selenium-ide-tips

WOODWARD, M., & HENNELL, M. (2006). On the relationship between two control-flow coverage criteria: all JJ-paths and MCDC. Inf. Softw. Technol.

XU, B., QIAN, J., ZHANG, X., WU, Z., & CHEN, L. (2005). A brief survey of program slicing. SIGSOFT.

ZANDSTRA, M. (2016). Testing with PHPUnit. V *PHP Objects, Patterns, and Practice* (stránky 435-464). Berkeley: Apress. doi:10.1007/978-1-4842-1996-6_18

ZELLER, A. (2006). A Guide to Systematic Debugging. V *Why Programs Fail.* San Francisco.

UNIVERZITA HRADEC KRÁLOVÉ
Fakulta informatiky a managementu
Akademický rok: 2020/2021

Studijní program: Aplikovaná informatika
Forma studia: Prezenční
Obor/kombinace: Aplikovaná informatika (ai2-p)

Podklad pro zadání DIPLOMOVÉ práce studenta

Jméno a příjmení:	**Bc. Ondřej Habr**
Osobní číslo:	**I2000037**
Adresa:	Raichlové 855, Náchod, 54701 Náchod 1, Česká republika
Téma práce:	Automatizované testování eshopu
Téma práce anglicky:	Automated testing of e-shop
Jazyk práce:	Čeština
Vedoucí práce:	Mgr. Daniela Ponce, Ph.D. Katedra informačních technologií

Zásady pro vypracování:

Cílem práce je analyzovat, navrhnout a implementovat automatizované testování na vybraném e-shopu.
Osnova:

1. Úvod
2. Cíl práce
3. Metodika zpracování
4. Automatizované testování aplikací
5. Případová studie: e-shop
6. Analýza, návrh a implementace testování
7. Shrnutí
8. Závěr a doporučení

Seznam doporučené literatury:

Podpis studenta: Datum:

Podpis vedoucího práce: Datum: