VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

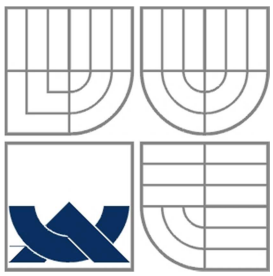
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

NÁVRH A IMPLEMENTÁCIA PRIEBEŽNEJ INTEGRÁCIE V SPOLOČNOSTI LOGIO

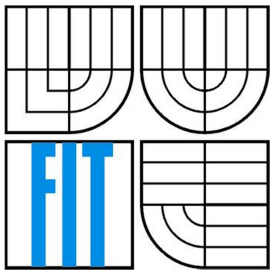
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. MICHAL MURÁŇ
AUTHOR

BRNO 2015



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

NÁVRH A IMPLEMENTACE PRŮBĚŽNÉ INTEGRACE VE SPOLEČNOSTI LOGIO

DESIGN AND IMPLEMENTATION OF CONTINUES INTEGRATION IN LOGIO COMPANY

DIPLOMOVÁ PRÁCE
MASTER‘S THESIS

AUTOR PRÁCE
AUTHOR

Bc. MICHAL MURÁŇ

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. PAVEL OČENÁŠEK, Ph.D.

BRNO 2015

Abstrakt

Tato diplomová práce popisuje možnost využití průběžné integrace a testování ve skutečné společnosti Logio. Výsledkem práce je implementace průběžné integrace v nástroji Jenkins a specifického testování pro firmu Logio. Teoretická část se zabývá základními principy, technikami a nástroji definujících průběžnou integraci. Následně jsou popsané již existující nástroje určené pro programovací jazyk PHP. Práce také rozebírá různé druhy a alternativy testování aplikací. Vysvětluje též problémy ve vývoji produktu ve společnosti Logio, které jsou důsledkem potřeby zavedení technik průběžné integrace. Praktická část práce popisuje implementaci testovacího rámce PwTester a jeho využití v průběžné integraci. Na závěr práce jsou zhodnoceny dosažené výsledky a nastíněné možnosti rozšíření.

Abstract

This master‘s thesis describes the possibility of using a continues integration and testing in real company Logio. The result of thesis is implementation of continuous integration tools Jenkins and specific testing for the company Logio. The theoretical part deals with basic principles, techniques and tools defining continuous integration. Subsequently are describing existing tools designed for PHP programming language. The thesis also analyzes various types and possibilities of application testing. Also explains the problems in product development in company Logio, which are consequence of the need for introducing continues integration techniques. The practical part of thesis describes the implementation of a test framework PwTester and its use in continuous integration. In conclusion, the results are evaluated and outlined the possibility of extension.

Klíčová slova

Průběžná integrace, PHP, server, Logio, testování, Selenium, jednotkové testování, sestavení, integrační testy, PHPUnit, Nette Tester, pahýl, úloha

Keywords

Continues integration, PHP, server, Logio, testing, Selenium, unit testing, build, integration tests, PHPUnit, Nette Tester, stub, job

Citace

Muráň Michal: Návrh a implementácia priebežnej integrácie v spoločnosti Logio, diplomová práce, Brno, FIT VUT v Brně, 2015

Návrh a implementácia priebežnej integrácie v spoločnosti Logio

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Pavla Očenáška, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Michal Muráň
18. 5. 2015

Poděkování

Chtěl bych poděkovat svému vedoucímu panu Ing. Pavlu Očenáškovi, Ph.D., za odbornou pomoc během tvorby práce. Za poskytnutí materiálů a informací bych také rád poděkoval společnosti Logio, pomocí které mohla tato práce vzniknout.

Obsah

Zoznam obrázkov

1 Úvod

Súčasná doba kladie vysoké nároky a očakávania na stále rozširujúci sa trend vývoja nových produktov v informačnom odvetví. To má za následok snahu znižovať náklady na tvorbu softvéru a tým umožniť rozsiahlejší vývoj. Pri tvorbe väčších projektov vznikajú opakujúce sa jednoduché a zložité úkony, ktoré je vždy nutné vykonať. Realizovanie týchto úloh uberá z priamych a nepriamych nákladov, ktoré je pri implementácii softvéru potreba vždy držať na čo najnižšej úrovni alebo tieto prostriedky využiť lepšie na dôležitejšie aspekty vývoja. Postupom času sa zistilo, že je vhodné zaviesť overené metódy, ktorými tieto úlohy môžeme automatizovať.

Preto sa od počiatku vývoja prvých softvérových produktov začalo pracovať na postupoch, návodoch a metodikách, ktoré by celý proces sprehľadnili a zrýchlili. Na základe týchto poznatkov vzniklo úplne nové odvetvie softvérové inžinierstvo, ktoré má pomáhať s hľadaním osvedčených postupov pri implementácii softvéru. Počiatkom 90. rokov prišiel nástup agilnej metodológie s názvom Extrémne programovanie [1] pomocou, ktorej sa snažíme využiť osvedčené techniky a na ich základe dosiahnuť maximálnu efektivitu. Vo väčších softvérových firmách, kde je cieľom rýchlosť a spoľahlivosť pri vývoji, začali tieto techniky využívať a úlohy automatizovať. A ako už bolo spomínané trend sa stále posúva ďalej. Aj menšie spoločnosti zaznamenali nutnosť zautomatizovať opakujúce sa úlohy a tým zefektívniť prácu a zvýšiť svoju konkurencieschopnosť.

Táto práca popisuje využitie princípov tzv. **Priebežnej integrácie** (angl. Continuous integration). Vznikla v spolupráci so spoločnosťou Logio. Kde priebežnú integráciu chápeme ako súhrn rôznych vývojárskych nástrojov a metód k urýchleniu vývoja softvéru, a spolupráce tímov. Priebežná integrácia je súčasťou metodiky Extrémneho programovania. Obsahom práce je taktiež špecifické testovanie určené pre produkt spoločnosti Logio s názvom Planning Wizard. Pod špecifickým testovaním sa chápe testovanie prepočtu systému Planning Wizard, testovanie funkčnosti modulov a testovanie grafického rozhrania.

V druhej kapitole práce sa popisuje obecný princíp priebežnej integrácie a príčiny jej zavádzania. Do tohto popisu patria aj jednotlivé najznámejšie už existujúce riešenia podporujúce priebežnú integráciu, ktoré sú spísané v tretej kapitole.

Dôležitou časťou práce je samostatné testovanie. Preto sú v štvrtej kapitole prvotne opísané rôzne spôsoby testovania a testovacie prístupy. Následne práca v piatej kapitole definuje princíp, výhody a nevýhody automatizácie testov, spolu s dostupnými nástrojmi umožňujúcimi ich automatizáciu. V rámci práce pôjde tiež o otestovanie výsledkov a prepočtu systému Planning Wizard.

V nasledujúcej kapitole šesť sú jednotlivé dôvody a problémy týkajúce sa nutnosti zavedenia nových techník pri vývoji softvéru v spoločnosti Logio. Na základe analýzy problémov a potrieb spolu s poznatkami z predošlých kapitol vznikol návrh nachádzajúci sa v siedmej kapitole.

Ôsma kapitola zachycuje samotnú implementáciu návrhu. Kde prvá časť obsahuje tvorbu testovacieho rámca PwTester určeného pre manuálne spúšťanie jednotkových a integračných testov. Druhá časť popisuje použitie nástroja Jenkins a jeho štruktúru pre podporu priebežnej integrácie.

V predposlednej kapitole sú zhodnotené výsledky práce. Nadväzujúce možné rozšírenia systému sú definované v desiatej kapitole.

Práca vychádza zo semestrálneho projektu, ktorý predstavil priebežnú integráciu spolu s vysvetlením a rozdelením alternatív testovania softvéru. Zároveň rozobral dostupné nástroje pre

testovanie, priebežnú integráciu a zhodnotil programovú situáciu v spoločnosti Logio, na základe ktorej vznikol návrh implementácie. Zo semestrálneho projektu boli prevzaté kapitoly 2 až 7.

2 Priebežná integrácia

S pojmom priebežná integrácia sme sa mohli stretnúť po prvý krát v práci Object-Oriented Analysis and Design with Application [2] od Grady Booch z roku 1994. Grady Booch vo svojej práci predstavuje ako pri vývoji využiť mikroprocesy, kde jednotlivé interné zverejnenia kódov predstavujú priebežnú integráciu systému a skladajú sa z opakujúcich úkonov. Do tejto platformy zahrňuje Grady Booch aj testovanie.

Priebežná integrácia sa chápe ako súčasť konceptu Extrémneho programovania. Za tvorcu extrémneho programovania sa berie Kent Beck a skupina ľudí okolo neho, ktorí do tejto metodológie zahrnuli aj priebežnú integráciu.

V roku 2000 vydal Martin Fowler článok o priebežnej integrácii, v ktorom spísal používané princípy a predstavil ich v rámci metodiky. Vo svojom článku Martin Fowler popisuje priebežnú integráciu ako: *„..praktiku softvérového vývoja, kde členovia tímu integrujú svoju prácu často, každá osoba aspoň jeden krát denne - čo vedie k viac integráciám denne. Každá integrácia je overená automatizovaným zostavením (vrátane testovania) pre detekcie integračných chýb tak rýchlo ako to je len možné. Veľa tímov zistilo, že tento prístup vedie k výraznému zníženiu problémov s integráciou a umožňuje tímu rýchlo vyvíjať súdržnejší softvér.“* [3].

2.1 Definícia priebežnej integrácie

Slovo integrácia ako už je zrejmé zo samotného názvu je kľúčom danej metodiky. Pod pojmom integrácie sa chápe zahrňovanie zmien do jednotného celku. Zjednotenie by sa malo realizovať vždy po každej zmene zdrojového kódu. Toto definuje aj prvé slovo z názvu a to priebežná - integrácia prebieha opakovane. Základne zloženie integrácie zobrazuje Obrázok 2.1.

V určitých metodológiách vývoja softvéru sa fáza integrácie a testovania nachádza až po skončení fáze implementácie, čo v niektorých prípadoch nie je vhodné. Príkladom môže byť tvorba komplexnejšieho softvéru, na ktorom pracuje väčšia skupina ľudí. Tí môžu pracovať na spoločných alebo samostatných moduloch, ktoré sa spájajú do jedného celku, či už funkčného alebo stále v etape vývoja. Spolu s týmto súvisia aj zmeny zadania projektu zo strany zákazníkov alebo aj od programátorov, ktoré je potreba zahrnúť do prebiehajúcej implementácie. V integrácií sa z toho dôvodu nachádza aj testovanie a kontrola kvality kódu, tým pádom sa nám tieto etapy rozložia do celého priebehu vývoja.

V rámci implementácie sa teda vykonávajú úkony, ktoré do nej spadajú, to znamená integrácia kódu a jeho otestovanie, poprípade vytvorenie spustiteľnej verzie a mnoho ďalších. Všetky tieto úkony sa v prípade nevyužitia priebežnej integrácie vykonajú len raz alebo párkrát manuálne a podľa všetkého, dôjde skôr k väčším problémom ako k úžitku.

Obrázok 2.1: Definícia a zloženie integrácie [2]

2.2 Ciele a hodnoty priebežnej integrácie

Jedným z cieľov priebežnej integrácie je primárne ušetriť či už hmotné alebo nehmotné zdroje. Na vykonanie úkonov, ktoré zahrňuje priebežná integrácia je nutné disponovať jedným alebo viacerými pracovníkmi primárne vyčlenenými na tento druh práce. Z praxe sa vypozorovalo, že ak je po človeku vyžadované vykonávanie rutinných činností, je väčšia pravdepodobnosť výskytu chýb a celková motivácia človeka klesá. Tieto aspekty pomáha vyriešiť zavedenie priebežnej integrácie, kde dôjde k automatizovaniu celého procesu, zrýchleniu vývoja a odstráneniu faktoru ľudskej chybovosti.

Ďalším aspektom, kde priebežná integrácia prevyšuje človeka je kontrola danej integrácie a jej následné testovanie. Ako je známe človek nie je neomylný a vždy sa môže stať, že niečo prehliadne alebo pozabudne.

Z programátorského pohľadu je výhoda v získavaní rýchlej spätnej väzby a to či už v rámci každého zostavenia alebo v priebehu celého vývoja. Na najvyššej úrovni teda hodnota priebežnej integrácie pozostáva z týchto bodov [4]:

1. **Znižovanie rizika** – integráciou niekoľko krát za deň znižujeme riziká na projekte. A to vďaka tomu, že integrácia spúšťa testovanie a kontroluje kód vždy, čím sa dosahuje väčšia pravdepodobnosť odhalenia chýb. Všetko sa vykoná ešte skôr ako sa dostaneme do fáze koncového testovania. V priebehu času je rovnako možné sledovať zdravie softvérového produktu. Priebežná integrácia v skutočnosti predstavuje záchrannú sieť pred zavádzaním chýb do kódu. Môže ísť o nasledujúce riziká, ako je nedostatok súdržného a spustiteľného kódu, neskoré objavenie chýb, nekvalitný softvér a nedostatočná prehľadnosť kódu.
2. **Zredukovanie opakujúcich sa procesov** – nám šetrí čas, peniaze a námahu. Opakujúce sa procesy sú naprieč všetkými aktivitami pri práci na projekte. Pomocou automatizácie vieme zabezpečiť, že procesy budú vždy vykonané rovnako a môžeme si nastaviť ich poradie. Spustenie procesov nastáva buď na základe užívateľského nastavenia, čo môže byť vždy po nahratí novej zmeny na centrálne úložisko alebo cyklicky v daný čas. Ďalšou zásluhou automatizácie je voľnosť pre vývojárov, ušetrenie ich času, ktorý môžu využiť na iné aspekty vývoja.

3. **Vytvorenie spustiteľného softvéru** – priebežná integrácia je taktiež schopná vyprodukovať softvér, ktorý sa dá priamo ponúknuť zákazníkovi. Dá sa povedať, že ide o jednu z jej najväčších výhod. Výhoda spočíva v tom, že vždy na základe nejakej zmeny v kóde alebo návrhu máme po ich implementácií na konci pripravený softvér pre zákazníka. Nie je potreba zložitého pripravovania produktu, a v prípade chýb sú vývojári ihneď informovaní a majú možnosť zasiahnuť. Tento prínos je prevažne na strane zákazníkov, ktorí ocenia rýchlosť zmien.
4. **Zlepšenie čitateľnosti kódu** – ide vlastne o statickú analýzu kódu a kontrolu kvality. Integračný server poskytuje túto eventualitu a pomocou nástrojov môže dlhodobo sledovať trendy pri vývoji, čím pomáha pri rozhodovaní a zavádzaní nových vylepšení. Ďalej sú to jednotlivé informácie o metrikách kódu a ďalších vlastnostiach. Výsledkom priebežnej integrácie je teda poskytnúť alternatívu efektívne sa rozhodnúť na základe informácií z výsledkov a upozorniť na vyskytujúce sa trendy v kóde.
5. **Zvýšenie dôvery v softvérový produkt zo strany vývojárov** – využitie priebežnej integrácie podnecuje vo vývojároch pocit dôvery v prácu, ktorú vykonávajú. Dôvodom je, že po každej svojej zmene sa kód vždy nanovo integruje a teda vykoná sa testovanie, ktoré nás utvrdí o správnosti danej zmeny, skontroluje splnenie noriem alebo máme ihneď k dispozícií informáciu o chybe.

2.3 Priebežná integrácia v softvérovej spoločnosti

Základné výhody plynúce z priebežnej integrácie už sú popísané v kapitole 2.2. Nasledujúca kapitola bude rozdelená do dvoch častí.

Prvá obsahuje dôvody zavedenia priebežnej integrácie do spoločnosti a potom body, ktoré predstavujú základ pre implementáciu priebežnej integrácie.

Druhá podkapitola obsahuje body, prečo niektoré spoločnosti predchádzajú implementácií priebežnej integrácie.

2.3.1 Dôvody použitia priebežnej integrácie

V nasledujúcich pár bodoch sú spísane dôvody na základe, ktorých by sa malo rozhodnúť, či je vhodné implementovať priebežnú integráciu v spoločnosti zaoberajúcej sa vývojom softvéru. Medzi základné dôvody implementácie patria [3]:

- vysoká chybovosť v zdrojových kódoch,
- zostavenie aplikácie je zložitý proces,
- zostavenie aplikácie je každodenná záležitosť,
- nájdenie chýb je časovo a kapacitne náročné,
- chaos vo verzovaní jednotlivých zostaveniach aplikácie,
- neporiadok v úložisku zdrojových kódov,
- nedostatočný technický dohľad nad softvérovými projektmi.

Je možné, že vo firme sa už využívajú princípy alebo techniky priebežnej integrácie a ani sa o tom nevie alebo sa vo firme disponuje predpokladmi na jej využitie. V takejto situácií sa dostávame

k otázke, či je lepšie zostať pri doterajšom riešení alebo prevziať techniky priebežnej integrácie. Samotné rozhodnutie závisí na vedení spoločnosti a k jej prístupu k výhodám z toho plynúcich. Následné body by mali pomôcť pri rozhodovaní, či zaviesť priebežnú integráciu [3]:

- Všetci vývojári si vykonávajú zostavenia pred odovzdaním kódu do spoločného repozitára na svojich lokálnych pracovných strojoch. Dôvodom je kontrola kódu pred jej zverejnením.
- Zverejnenie kódov do spoločného repozitára sa uskutočňuje aspoň raz za deň.
- K zostaveniu kódu dochádza niekoľkokrát za deň na rozdielnych výpočetných strojoch.
- Testy musia byť vykonané na 100% pri každom zostavení.
- Produkt môže byť funkčne testovateľný.
- Najväčšiu prioritu majú opravy chybných zostavení.
- Niektorý programátori na základe štatistík a metrík z vykonaných zostavení hľadajú príležitosti zlepšenia.

2.3.2 Prečo priebežnú integráciu nezavádzať

Priebežná integrácia ponúka veľa výhod avšak existujú dôvody, kedy sa predchádza jej implementácií. Medzi dané dôvody môže patriť [4]:

- **Zvýšené nároky na udržiavanie systému priebežnej integrácie** – tomuto tvrdeniu sa dá oponovať, pretože nutnosť integrovať, testovať a ostatné kroky je nutné vždy. A je pravdepodobne lepšie spravovať robustný systém, ako riadiť manuálne procesy. V rámci multiplatformných projektoch je priebežná integrácia najviac potrebná.
- **Príliš veľa zmien** – niektorý užívatelia majú pocit, že je potreba zmeniť veľa procesov, aby bolo možné použiť priebežnú integráciu na staršie projekty.
- **Príliš veľa zlyhaných zostavení** – často k tomu dochádza, keď si vývojári nevyskúšajú preložiť svoj kód pred jeho uložením do spoločného centrálneho úložiska. Tu je nevyhnutná rýchla reakcia, a to z dôvodu frekvencie zmien v kóde.
- **Navýšenie nákladov na hardvér/softvér** – pre efektívne využitie priebežnej integrácie je vhodné zabezpečiť stroj primárne určený na túto prácu, avšak nie je to nevyhnutnosť. Náklady použité na tieto účely sú malé v porovnaní s nákladmi vynaloženými na hľadanie neskorších problémov vo fáze vývoja.
- **Programátori by mali tieto aktivity vykonávať** – v istých situáciách má manažment pocit, že priebežná integrácia je len duplikácia činností, ktoré by mali vykonávať vývojári tak ako tak. Pravdou je, že by sa mali tieto činnosti vykonávať, ale je lepšie a spoľahlivejšie, keď sa vykonávajú v oddelenom a čistom prostredí. Využitie automatizovaných procesov zvyšuje efektívnosť a frekvenciu týchto činností.

2.4 Techniky a funkcie priebežnej integrácie

Priebežná integrácia je akýsi súhrn techník a metód, ktoré je možné pri vývoji softvéru uplatniť. Ich základy spísali vo svojich dielach Fowler [3] a Duvall [4]. Podkapitoly nižšie pojednávajú a definujú kľúčové praktiky, ktoré tvoria účinnú priebežnú integráciu.

2.4.1 Centrálne úložisko zdrojových kódov

Každý softvérový projekt obsahuje veľké množstvo súborov, na ktorých pracuje rôzny počet ľudí. Najlepším riešením je použiť centrálne úložisko, ktoré nám dovoľuje spravovať súbory na jednom mieste, o ktorom všetci vedia a majú k nemu prístup. To nám zaistí konzistenciu všetkých súborov a taktiež zálohovanie jednotlivých verzií. Úložisko môže byť buď distribuované alebo centrálne. Momentálne existuje veľké množstvo či už voľne dostupných alebo komerčných riešení. Medzi najznámejšie voľne šíriteľné nástroje sa dá považovať centrálny Subversion[1] alebo distribuovaný Git[2].

Zo zavedením jednotného úložiska súvisia aj určité pravidlá a štruktúra, ktoré je potrebné dodržiavať. Pravidlom je, aby repozitár obsahoval všetky potrebné zdroje a dodržiavali sa jednotné konvencie pridávania nových súborov. Ako tvrdí Fowler: *„Základným pravidlom je, že keď vstúpim do vývoja projektu s novo nainštalovaným počítačom, mal by som po získaní všetkých súborov z repozitára, byť schopný vykonať kompiláciu aplikácie“* [3]. Zároveň by sa v úložisku nemali nachádzať súbory, ktoré vznikajú pri kompilácií. Ich výskyt by podľa Fowlera znamenal, že nie sme schopný skompilovať projekt bez týchto súborov, a preto sú v úložisku. Je potrebné, aby programátori ukladali svoje zmeny častejšie. Má to za následok lepšiu správu vývoja projektu, lebo vývojári musia svoju prácu deliť na čo najmenšie celky.

2.4.2 Automatizácia zostavenia

Pri prevode zdrojových kódov do funkčného produktu ide často o zložitý proces. Tento proces zahŕňa kompiláciu, presúvanie súborov, nahrávanie schémy databáze a mnoho ďalších úkonov. Úlohou priebežnej integrácie je tieto procesy automatizovať a zaistiť správnosť integrácie zmien. Zostavenie sa vykonáva na integračnom serveri. Ide v skutočnosti o zostavujúce skripty obsahujúce súbor krokov, ktoré sa majú dať do pohybu. Medzi známe nástroje pre automatizované zostavenie patrí *Ant*[3] pre kompiláciu projektov v jazyku Java, ďalej známy *Make*[4] využívaný v systémoch Linux a Unix a mnoho ďalších. Ak nemôžeme využiť nejaký z už existujúcich skriptov nie je problém vytvoriť si vlastný.

Spustenie jednotlivých zostavení by malo byť na základe zmien uložených do už spomínaného centrálneho repozitára. Dôležitým prvkom je, aby sa pred uložením zmien do repozitára overila ich funkčnosť na lokálnom stroji. Ak by sa tento postup nedodržal integračný server by slúžil len na overenie kompilovateľnosti a nie pre kontrolu integrácie zmien [3].

2.4.3 Testovanie zostavenia

To, že je produkt po zostavení schopný behu, nemusí ešte znamenať, že vykonáva to čo má. Preto je vhodné zaradiť ako ďalší krok po zostavení testovanie. Najlepším spôsobom ako čo najrýchlejšie a efektívne odchytiť chyby v rámci integrácie sú automatizované testy.

---

[1] Viac informácií na URL: https://subversion.apache.org/

[2] Viac informácií na URL: http://git-scm.com/

[3] Viac informácií na URL: http://ant.apache.org/

[4] Viac informácií na URL: http://www.gnu.org/software/make/

V prípade automatizovaného testovania kódu potrebujeme skupinu takých testov, ktoré dokážu skontrolovať veľkú časť kódu na základne chyby. Dôležitým prvkom je, aby bolo možné tieto testy spustiť pomocou jednoduchého príkazu a boli schopné sebakontroly. Ak testy zlyhajú, malo by dôjsť k zastaveniu integrácie a odoslaniu správy o zlyhaní.

V poslednej dobe sa na základe metodiky Testami riadený vývoj[5] rozšírili voľne dostupné nástroje rodiny *xUnit*. Ide o veľmi vhodné nástroje pre tento druh testovania. Pod *xUnit* sa schovávajú nástroje podporujúce vytváranie jednoduchých automatizovaných testov, ktoré si môžu sami napísať programátori a nepotrebujú k tomu väčšie skúsenosti [3].

2.4.4 Ukladanie kódu do spoločného repozitára každý deň

V priebežnej integrácie ide predovšetkým o komunikáciu medzi vývojármi. Jedným s nástrojov, ktorý to umožňujú je už spomínaný spoločný repozitár z kapitoly 2.4.1. Pomocou repozitára si vývojári dávajú vedieť o jednotlivých zmenách v kóde medzi sebou.

Primárnym účelom integrácie je, aby všetci programátori ukladali svoje zmeny do jednotného hlavného vlákna. Postup práce by mal byť nasledujúci:

1. Ak má vývojár vytvorený kód, ktorý chce nahrať do repozitára mal by si najprv stiahnuť najaktuálnejšie zdrojové kódy.
2. Potom by mal vyriešiť prípadné konflikty v súboroch.
3. Na svojom lokálnom stroji otestovať dané zmeny, myslí sa tým vyskúšať zostaviť daný kód.
4. Ak všetky body doposiaľ prebehli v poriadku, programátor už môže svoje zmeny v kóde zahrnúť do spoločného úložiska.
5. Na záver je už na práci integračného serveru zabezpečiť zostavenie a následné otestovanie.

Celý tento postup by mal zabezpečiť správnosť vývoja. Čím častejšie sa bude vykonávať tento cyklus, tým skôr sa príde na dané konflikty medzi kódmi viacerých vývojárov. A kľúčom k rýchlemu odstráneniu chýb je ich rýchle nájdenie [3].

2.4.5 Po každej zmene kódu dôjde k zostaveniu aplikácie na integračnom stroji

Každá zmena uložená do centrálneho repozitára musí vyvolať zostavenie. A na základe pozitívneho výsledku z danej integrácie by sa mala daná zmena považovať za hotovú.

Za každú zmenu kódu by mal byť zodpovedný ten kto ju vytvoril. A je na ňom uistiť sa, že jeho zmeny sú v poriadku. Nestačí len otestovať kód na lokálnom stroji. Existujú dva hlavné spôsoby ako zaistiť správnosť zmien v kóde, a to buď manuálnym zostavením alebo pomocou integračného serveru.

Veľa firiem vykonáva pravidelné zostavenia podľa časového plánu, najčastejšie každý večer. Takýto prístup nie je dostatočný pre priebežnú integráciu. Zmyslom priebežnej integrácie je odhaliť problém čo najskôr. Večerné zostavenia znamenajú, že chyby zostávajú neodhalené celý deň, dokým ich niekto objaví. Čím dlhšie sú chyby v kóde, tým dlhšie trvá ich nájdenie a odstránenie [3].

[5] Viac informácií na URL: http://agiledata.org/essays/tdd.html

2.4.6 Okamžité opravenie chybných zostavení

Kľúčovou časťou priebežnej integrácie je, že ak nastane chyba v zostavení, je potreba chybu ihneď odstrániť. Zmyslom je, aby sa vyvíjalo po celú dobu na stabilnej verzii kódu. Chyba počas zostavenia nie je zlá vec, ale ak dochádza k tomuto stavu často je pravdepodobné, že programátori nie sú dostatočne obozretný pri vývoji.

Ako tvrdí Kent Beck: *„Nikto nemá prioritnejšiu úlohu ako je opravenie zostavenia.“*. To však neznamená, že by teraz mal celý vývojársky tím pracovať na opravení zostavenia, zvyčajne na to stačí pár ľudí [1].

2.4.7 Udržiavať zostavenie rýchle

Priebežná integrácia by mala poskytnúť čo najrýchlejšiu spätnú väzbu. Zostavenie by teda nemalo trvať príliš dlho.

V prípade Extrémneho programovania sa desať minút považuje za perfektný čas potrebný na zostavenie. Sústredenie sa na čo najväčšie zníženie času trvania zostavenia teda stojí za tu námahu, pretože každá zredukovaná minúta je minúta, ušetrená pri každej novej zmene kódu a teda ďalšom cykle integrácie.

Avšak existujú aj projekty, pri ktorých nie je možné sa dostať na také číslo ako je desať minút. Obvyklou prekážkou je testovanie, ktoré zahŕňa externé služby ako sú napríklad databáze.

Najlepším krokom v takejto situácii je zostavenie rozdeliť na časti. V skutočnosti budeme mať niekoľko zostavení vykonávajúcich sa sekvenčne. Najčastejším prípadom je rozdelenie na dve časti, kde prvá primárna časť sa spúšťa po uložení zmien do repozitára. Toto primárne zostavenie je najdôležitejšie, a teda aj najrýchlejšie. Za cenu rýchlosti je potreba vzdať sa istých úkonov, pravdepodobne zložitých testov. Trik je v tom nájsť rovnováhu medzi rýchlosťou a potrebou objavenia chýb, tak aby primárne zostavenie bolo dostatočne stabilné pre ďalšiu prácu. Ako náhle prejde primárne zostavenie v poriadku, programátori môžu následne pracovať s dôverou na kóde.

Po úspešnom primárnom zostavení sa môže dostať na rad sekundárne zostavenie vykonávajúce pokročilejšie testovanie alebo ďalšie požadované úkony. Ak dôjde počas sekundárneho zostavenia k odhaleniu chýb nejde o veľký problém. Je to vďaka tomu, že dôležité testy funkčnosti sú zahrnuté v primárnom zostavení.

U zložitých výpočtoch je možné vykonávať zostavenia na viacerých serveroch a taktiež vykonávať úlohy paralelne [3], [4].

2.4.8 Testovať v prostredí čo najpodobnejšiemu produkčnému

Snahou testovania je odhaliť, za kontrolovaných podmienok, hocijaký problém, ktorý môže nastať v produkčnom prostredí. Významnou časťou je prostredie, na ktorom sa bude systém spúšťať. Ak testujeme v rôznom prostredí, vedie každý rozdielny výsledok k riziku, že to čo sa deje v rámci testu nemusí nastať v produkčnom prostredí.

V dôsledku toho by sa malo testovacie prostredie nastaviť tak, aby čo najpresnejšie ako sa len dá napodobovalo produkčné prostredie. Tým sa myslí napríklad použitie rovnakého databázového softvéru, rovnaký operačný systém a iné. Naďalej sprístupniť všetky príslušné knižnice, ktoré sú v produkčnom systéme, a to aj ak ich produkt nevyužíva.

V dnešnej dobe sa často využíva virtualizácia. Pomocou virtualizácie je veľmi jednoduché vytvoriť potrebné testovacie prostredie, ktoré bude obsahovať všetky potrebné prvky. Je teda pomerne jednoduché nainštalovať posledné zostavenie produktu a realizovať testy. Okrem toho virtualizácia umožňuje povoliť viac testov na jednom počítači alebo simulovať viac počítačov v sieti [3].

2.4.9 Čo najľahšie sprístupniť každému posledný spustiteľný kód

Jednou z najťažších častí vývoja softvéru je uistiť sa, že budeme vytvárať ten správny softvér. Popravde je veľmi ťažké špecifikovať vopred a správne, čo chceme. Ľudia prišli na to, že je oveľa ľahšie vidieť niečo čo nie je správne a povedať, čo je potreba zmeniť. Agilný vývoj výslovne očakáva a využíva takýto spôsob ľudského zmýšľania.

Ak chceme, aby takýto princíp fungoval, mali by sme všetkým tým, ktorý sú zahrnutý do vývoja projektu sprístupniť posledné spustiteľné súbory a musia byť schopný spustiť demonštráciu, testovanie alebo len preto, aby videli, aké zmeny vo vývoji nastali. Veľmi užitočné je uchovávať niekoľko takýchto posledných spustiteľných verzií [3].

2.4.10 Každý vidí čo sa deje

Priebežná integrácia je celá o komunikácií, takže je potreba zabezpečiť, aby každý mohol ľahko vidieť stav systému a zmeny, ktoré boli vykonané.

Jednou z najdôležitejších informácii je stav alebo priebeh posledného zostavenia. Je dôležité informovať programátorov o jeho úspešnosti alebo neúspešnosti. Na základe danej informácie vedia, či je možné si stiahnuť novú verziu súborov z hlavného repozitára.

V rámci integračného serveru sú dostupné webové stránky, ktoré obsahujú viac informácií než u manuálneho zostavenia. Vývojári môžu byť informovaný taktiež pomocou emailov, SMS, aplikáciách bežiacich na ich lokálnych strojoch alebo pomocou rôznych rozšírení dostupných pre ich vývojové prostredia. Poskytuje informácie nielen o tom kto spustil zostavenie, ale aj aké zmeny vykonal. Taktiež disponuje informáciami o predošlých zmenách, takže členovia tímu majú dobrý prehľad o nedávnej činnosti na projekte. Ďalšou výhodou webových stránok je, že ak na projekte pracujú ľudia, ktorý sa nenachádzajú na rovnakom mieste stále môžu získať informácie o stave vývoja [3].

2.4.11 Automatizované nasadenie

Vo vymedzených prípadoch priebežnej integrácie potrebujeme multiplatformné prostredia, jedno pre beh primárneho zostavenia, jedno alebo viac pre sekundárne zostavenia. V týchto situáciách musíme prenášať spustiteľné verzie z jedného prostredia na druhé a to automaticky. Pre takéto akcie potrebujeme mať skripty, ktoré nám umožnia umiestniť produkt do daného prostredia čo najľahšie.

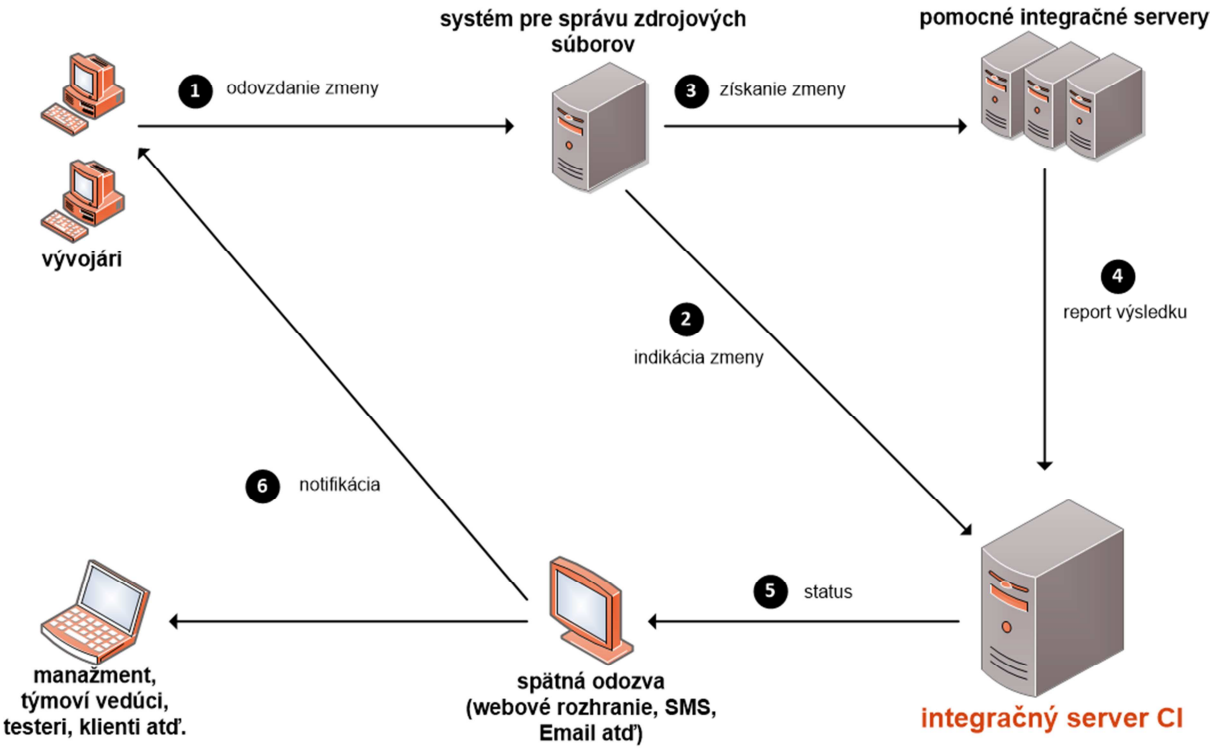
Prirodzeným dôsledkom je, že keď už disponujeme takýmito skriptami umožňujúcimi jednoduché nasadenie produktu na jednotlivé prostredia, prečo to nevyužiť na nasadenie do produkčného prostredia. Pravidlom je, že k nasadzovaniu by nemalo dochádzať každý deň, ale automatické nasadenie pomáha urýchliť vývoj a redukuje množstvo chýb. Taktiež je to lacná alternatíva, lebo využívame rovnaké funkcie ako pri nasadzovaní do testovacích prostredí.

Zaujímavým riešením je nasadiť skúšobne zostavenie len pre podmnožinu užívateľov. Tím má tak možnosť vidieť ako sa zostavenie správa ešte pred rozhodnutím, či aplikáciu nasadiť do prevádzky pre všetkých používateľov. Umožní to otestovať nové funkcie a užívateľské rozhranie [3], [4].

Automatickým nasadením sa zaoberá samostatná metodika **Priebežné nasadenie** (angl. Continuous Delivery) [5] ide o prirodzené rozšírenie priebežnej integrácie. Je to prístup, pri ktorom tím zabezpečí, že po každej zmene je možné produkt ihneď uvoľniť do sveta a nie je problém tak učiniť s akoukoľvek verziou. Priebežné nasadenie si berie za cieľ, aby sa uvoľňovanie verzií stalo jednoduchým, takže ho budeme schopný vykonávať často a získame rýchlu spätnú odozvu o tom na čom používateľovi záleží.

2.4.12 Cyklus priebežnej integrácie

Pre využitie priebežnej integrácie ako už bolo spomínané nie je potreba disponovať integračným serverom. Ale na základe výhod plynúcich z integračného serveru nie je dôvod prečo ho nepoužiť. Na nasledujúcom obrázku je znázornený základný cyklus priebežnej integrácie s využitím integračného serveru spolu s popisom jednotlivých krokov [6].



Obrázok 2.2: Typický cyklus priebežnej integrácie [6]

Popis cyklu priebežnej integrácie:

1) Vývojári uložia zmenené súbory na centrálny repozitár (Subversion, Git, ...).
2) Centrálny repozitár pravidelne informuje integračný server o zmenách v zdrojových kódoch.
3) Na základe indikácie zmeny sa z centrálneho repozitára získajú nové súbory a je spustené nové zostavenie. Na zložitejšie zostavení je možné využiť pomocné servery.

4) Výsledky zo zostavenia sú predané integračnému serveru na spracovanie (výsledky testov, analýza kódu, atď.).
5) Integračný server na základe výstupu zo zostavenia nastaví stav projektu ako stabilný, nestabilný alebo neúspešný.
6) V závere sú informácie o zostavení sprístupnené a zaslané vývojárom a participujúcim osobám.

3 Systémy pre priebežnú integráciu

Priebežnú integráciu môžeme implementovať jednoducho pomocou skriptov, ktoré budú dodržiavať, zabezpečovať techniky a princípy z kapitoly 2.4. Tento prístup sa odporúča len v prípade špeciálnych požiadaviek, ktoré nedokážu byť uspokojené niektorým z už dostupných systémov pre priebežnú integráciu. Avšak v dnešnej dobe je takýchto prípadov veľmi málo. Súčasne systémy pre priebežnú integráciu v sebe zahrňujú aj funkčnosti navyše spolu s podporou rozširujúcich nástrojov. Čo sa týka alternatív vlastnej úpravy a nastavení je ponuka viac ako dostačujúca.

Nižšie v kapitole sú rozobrané najčastejšie používané systémy pre priebežnú integrácie so zameraním na programovací jazyk PHP. Zameranie sa na programovací jazyk PHP je z dôvodu najviac vyhovujúcemu pre potreby spoločnosti Logio.

3.1 Jenkins

Jenkins je voľne dostupný nástroj pod licenciou MIT napísaný v jazyku Java. Momentálne je k dispozícii verzia 1.596.2 s dlhotrvajúcou podporou. Ide o samostatnú vývojovú vetvu od známeho nástroja Hudson, od spoločnosti Oracle. Jenkins je jedným z najrozšírenejších nástrojov pre priebežnú integráciu, ktorý podporuje monitorovanie a vykonávania opakujúcich sa úloh, ako je zostavovanie softvérových projektov. Popri týchto funkciách sa Jenkins aktuálne zameriava aj na nasledujúce dve úlohy. Jednou z nich je testovanie alebo zostavenie softvérových projektov nepretržite a druhou je sledovanie vykonávania externe bežiacich úloh.

Ide o veľmi obľúbený nástroj. Zásluhu na tom má široká komunita dobrovoľníkov starajúcich sa o vývoj. Jenkins disponuje širokou podporou jazykov a nástrojov, ktoré je možné rozšíriť o viac než tisíc dostupných zásuvných modulov. Ak užívateľovi nevyhovuje žiadne z dostupných rozšírení nie je problém vytvoriť si svoje vlastné. Jenkins používa veľa známych firiem napríklad Dell, LinkedIn, NASA, Sony a mnoho ďalších.

Jednou z nevýhod alebo skôr jedinou najčastejšie spomínanou je grafické rozhranie samotnej aplikácie. Veľa užívateľov naráža na vzhľad, ktorý považujú za zastaraný a navyše je občas problém sa zorientovať v jednotlivých položkách. V prílohe A sa nachádzajú obrázky rozhrania. Za výhodu sa považuje dobre sformovaná dokumentácia a široká komunita fór, pomocou ktorých sa problémy dajú vyriešiť veľmi rýchlo [7].

3.2 TeamCity

TeamCity je komerčný softvér s uzatvoreným zdrojovým kódom. Je dostupný zadarmo pre malé alebo neziskové projekty, ktoré splnia licenčné podmienky. Rovnako ako Jenkins je napísaný v jazyku Java a za jeho vývojom stojí spoločnosť JetBrains. Vďaka aktívnemu vývoju a silnej komunite je momentálne k dispozícii verzia s číslom 9.

Užívateľské rozhranie pôsobí moderným dojmom a jednotlivé položky sú zoradené logicky oproti spomínanému Jenkinsu, obrázok sa nachádza v prílohe A. Rovnako ako ostatné nástroje disponuje rôznymi grafickými zobrazeniami výsledkov a štatistík. Za výhodu považujem veľmi pekné

zobrazenie vývojových vetví a podrobné informácie o zmenách. Zaujímavou funkčnosťou je združovanie užívateľov do skupín, ktorým je možné nastaviť oprávnenie, ale aj rôzne spôsoby reportovania.

TeamCity vie vykonávať zostavenia a testy ešte pred samotným vložením zmien do repozitára. Taktiež disponuje možnosťou zálohovania celého systému, kde sa automaticky zhromaždia všetky informácie o projektoch a ich nastaveniach. Súvisiacou funkčnosťou je aj alternatíva uložiť nastavenia projektu priamo do repozitára.

Čo sa týka eventuality rozšírenia pomocou zásuvných modulov nie je k dispozícií také množstvo, ako u spomínaného Jenkinsu, ale s číslom presahujúcim sto sú obsiahnuté všetky skupiny nástrojov. K vývoju vlastného modulu sú k dispozícii prehľadné návody a dokumentácia.

Veľkou prednosťou TeamCity je jeho integrácia do vývojového prostredia. Podporovaných je veľa nástrojov. Vzniká tak príležitosť spravovania TeamCity vzdialenie cez vývojové prostredie a získať tak prehľad o dianí a výsledkoch zostavení [9].

3.3 PhpUnderControl

Ide o doplnok k softvéru CruiseControl, ktorý v sebe zahŕňa niektoré z najlepších vývojových nástrojov pre projekty v programovacom jazyku PHP. CruiseControl je jedným z najstarších programov zameraných na podporu priebežnej integrácie. PhpUnderControl beží na platforme Java a je distribuovaný pod licenciou BSD.

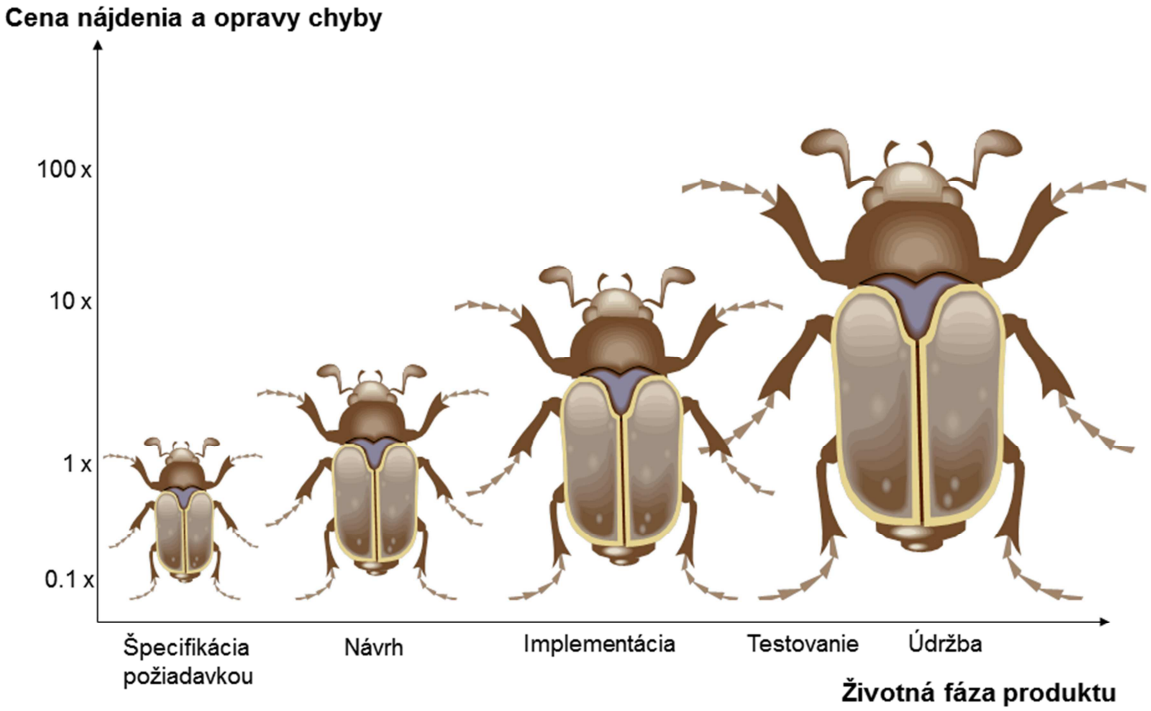
Výhoda softvéru je v jeho prehľadnosti a jednoduchosti používania. PhpUnderControl je skôr vhodný pre malé projekty. Disponuje základnými nástrojmi pre správu PHP projektov ako je testovanie, dokumentácia a zber štatistík. Čo sa týka ďalšieho vývoja a komunity okolo PhpUnderControl má situácia stagnačný charakter [11]. Obrázok systému sa nachádza v prílohe A.

4 Testovanie softvéru

Ako už bolo spomínané testovanie softvéru je nedielnou súčasťou priebežnej integrácie. A o čo vlastne ide pri testovaní softvéru? Definícia testovania sa priebehom času mení, ale tu je pár základných definícií:

- Testovanie je proces spúšťania programu so zámerom nájdenia chýb [12].
- Testovanie softvéru je empirický technický výskum kvality testovaného produktu alebo služby, vykonávane za účelom poskytnutia týchto informácií zainteresovaným osobám [13].

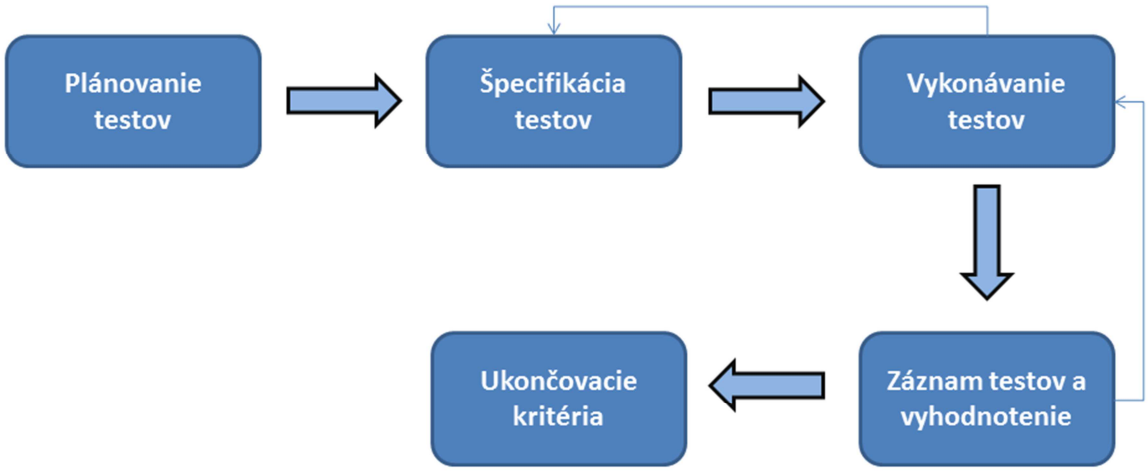
Hlavnými funkciami testov je overiť, či daný softvér odpovedá špecifickým potrebám a identifikovať defekty. Zároveň pomôcť pri znižovaní nákladov vynaložených na neskoršie opravy chýb a pomôcť so zvýšením kvality softvéru.



Obrázok 4.1: Náklady spojené s opravou chýb počas životných fáz produktu [14] [spracovanie vlastné]

Musíme si však uvedomiť, že pomocou testovania sa dá objaviť veľké množstvo chýb, napriek tomu sa nemusia nájsť všetky. Ako tvrdil Dijskra: „*Testovanie môže ukázať prítomnosť chýb, ale nemôže preukázať ich absenciu.*“ [15].

Testovanie softvéru sa považuje za kritickú fázu počas vývoja. Výsledné hodnotenie softvéru je priamo úmerné kvalite, a preto sa odporúča dôkladne venovať testovaniu.



Obrázok 4.2: Definícia základného iteratívneho testovacieho procesu [16] [spracovanie vlastné]

Následne sú v podkapitolách priblížené základné pohľady na typy, metódy a úrovne testovania. Jednotlivé definície vychádzajú z nasledujúcich dostupných materiálov [17], [18], [19], [20].

4.1 Spôsob testovania

Táto podkapitola popisuje spôsoby testovania, ktoré môžu byť použité, počas životného cyklu informačného systému. Rozdelenie je definované na základe toho, či sú testy vykonávané človekom alebo softvérom.

4.1.1 Manuálne testovanie

Zahrňuje testovanie softvéru manuálne a to bez použitia automatizovaných nástrojov alebo skriptov. V tejto situácii, tester berie na seba rolu koncového užívateľa a testuje softvér na neočakávané chovanie alebo chyby. Existujú rôzne stupne pre manuálne testovanie, ako sú jednotkové testy, integračné testy, systémové a akceptačné testy. Tester využíva testovacie plány, testovacie prípady alebo testovacie scenáre pre zabezpečenie úplnosti testovania. Manuálne testovanie sa preferuje v situáciách, kedy je potreba využiť ľudskú inteligenciu, nie sú dostupné prostriedky pre automatizáciu alebo nastavenie prostredia pre test je časovo náročné [19].

4.1.2 Automatizované testovanie

U automatizovaného testovania tester napíše testovacie skripty alebo využíva softvér pre otestovanie produktu. V skutočnosti tento proces zahŕňa automatizáciu manuálnych procesov. Automatizované testovanie sa využíva na znovu spúšťanie veľkého počtu testov alebo testov s veľkým množstvom generovaných dát. Taktiež sa používa na testovanie aplikácie od záťaže, výkonu a stresového hľadiska. Zvyšuje pokrytie testami, presnosť, šetrí čas a peniaze v porovnaní s manuálnym testovaním [19]. Viac informácií v kapitole 5.

4.2 Testovacie prístupy

Existujú rôzne prístupy použiteľné pre testovanie softvéru. Táto podkapitola obsahuje popis najzákladnejších z nich.

4.2.1 Čierna skrinka

Ide o techniku testovania bez akýchkoľvek znalostí o vnútornom fungovaní aplikácie. Zameriavame sa na vstupy a výstupy aplikácie. Tester si nevšíma architektúru systému a nemá prístup k zdrojovému kódu. Typicky pri vykonávaní testovania pomocou čiernej skrinky bude tester komunikovať primárne s užívateľským rozhraním, ktoré poskytuje vstupy a neskôr vyhodnocovať výstupy bez toho, aby vedel ako a kde sa vstupy spracovávajú. Tester sa zameriava na testovanie funkčnosti aplikácie vzhľadom k špecifikácii. Čierna skrinka oddeľuje pohľad užívateľa od pohľadu programátora. Výhodou je jednoduchosť, rýchlosť a nezávislosť na zmene hardvéru či operačného systému. Je efektívne pre veľké segmenty kódu. Medzi nevýhody však patrí nižšia kvalita kódu, spôsobená obmedzeným pokrytím, pretože len vybraný počet testovacích scenárov je skutočne vykonaný. Taktiež vytváranie testovacích prípadov je veľmi zložité [17], [18], [19].

4.2.2 Biela skrinka

Biela skrinka detailne skúma vnútornú logiku a štruktúru kódu. Je potreba nahliadnuť do vnútra aplikácie a zistiť, ktorá časť kódu sa chová nesprávne. Tým sa nám ale stráca pohľad užívateľa, avšak na základe znalosti kódu môžeme lepšie odhadnúť, kde hľadať chybu a kde nie. Medzi výhody patrí optimalizácia kódu. Pri znalosti kódu je jednoduchšie zistiť aký typ dát pomôže pri efektívnom testovaní aplikácie, a pri písaní testovacích scenárov sa dosahuje maximálneho pokrytia. Nevýhody sú zvýšenie nákladov, potreba špecializovaných nástrojov pre testovanie a nemožnosť overenia špecifikácie, lebo sa zameriavame len na vnútornú logiku a nie logiku špecifikácie [17], [18], [19].

4.2.3 Šedá skrinka

Je kompromis medzi čiernou a bielou skrinkou zaštiťujúca výhody z oboch riešení. Tester študuje požiadavky špecifikácie a komunikuje s vývojárom pre pochopenie vnútornej štruktúry systému. S danými znalosťami je tester schopný pripraviť lepšie testovacie dáta a scenáre pri tvorbe testovacieho plánu [17], [19].

4.2.4 Statické testovanie

Pri statickom testovaní nie je potreba softvér spúšťať. Je možné ho vykonávať ešte pred vytvorením prvého prototypu. Ako statické testovanie sa berie napríklad kontrola syntaxe, prechádzanie alebo inšpekcia zdrojového kódu, ktoré je možné vykonať či už manuálne alebo pomocou nástrojov ako je vývojové prostredie, napríklad pri kompilácii kódu [17], [18].

4.2.5 Dynamické testovanie

Vyžaduje spustenie aplikácie. Ide o testovanie dynamického chovania kódu. Prebieha hlavne na základe poskytovania rôznych vstupov a vyhodnocovania výstupov. Používa sa k zisteniu chýb a zabezpečeniu kvality kódu [17], [18].

4.2.6 Opätovné testovanie

Ak test zlyhá, a zistíme, že príčinou neúspechu je softvérová chyba, tak chyba je nahlásená a očakáva sa nová verzia softvéru, ktorá chybu opraví. Pri takejto situácii je potreba uskutočniť test znovu a potvrdiť, že chyba bola opravená. Tento princíp sa nazýva opätovné testovanie.

U opätovného testovania je dôležité zabezpečiť, aby sa test vykonal rovnakým spôsobom ako po prvýkrát, to znamená s použitím rovnakých vstupných dát a v rovnakom testovacom prostredí. Ak sa test vykonal správne, tak vieme, že časť kde bola chyba je opravená. Avšak oprava mohla zaviesť do softvéru novú alebo odkryť ďalšiu chybu. Spôsob ako odhaliť tieto nečakané vedľajšie účinky opráv, je regresné testovanie [20].

4.2.7 Regresné testovanie

Regresné testovanie rovnako ako opätovné zahŕňa v sebe vykonávanie testov, ktoré už boli vykonané predtým. Rozdiel je v tom, že pri regresnom testovaní, naposledy spustené testovacie prípady pravdepodobne skončili úspešne.

Cieľom regresného testovania je overiť, že zmeny v softvéri alebo v prostredí nespôsobili negatívne vedľajšie účinky, a teda systém stále spĺňa dané potreby. Regresné testovanie je vhodné mať na všetkých úrovniach vývoja. Pri vytvorení novej verzie systému by mali byť vyvolané všetky regresné testy, a to z nich robí ideálneho kandidáta pre automatizáciu. Regresné testy sa odporúča spúšťať nielen na základe zmeny softvéru, ale aj pri zmenách v prostredí, ako môže byť napríklad prechod na nový systém pre správu databáz [20].

4.2.8 Funkčné testovanie

Funkcie systému alebo jeho častí sú zvyčajne popísané v špecifikácií požiadaviek, vo funkčných nárokoch alebo v prípadoch užitia. Funkčné testovanie je založené na testovaní funkcií, popísaných v týchto dokumentoch. Podľa normy ISO9126[6] sa funkčné testovanie vykonáva so zameraním na vhodnosť, interoperabilitu, bezpečnosť, presnosť a dodržiavanie predpisov. Testovanie funkčnosti môže byť vykonané z dvoch hľadísk, a to na základe požiadaviek alebo obchodných procesoch.

Testovanie na základe požiadaviek využíva špecifikáciu funkčných nárokov na systém ako základ pre vytváranie testov. Jednotlivé potreby na testovanie by sa mali uprednostňovať na základe kritéria rizika. Tým sa zabezpečí, že najdôležitejšie a najkritickejšie testy sú zahrnuté v testovaní.

Testovanie na základe obchodných procesoch využíva znalosti podnikových činností. Podnikové procesy popisujú scenáre podieľajúce sa na každodennom využívaní systému. Prípady použitia (angl. use cases) pochádzajú z objektovo orientovaného vývoja, ale v dnešnej dobe sú

[6] Viac informácií na URL: http://www.sqa.net/iso9126.html

populárne v mnohých modeloch životného cyklu vývoja. Prípady užitia sú veľmi vhodným základom pre testovacie prípady z podnikového hľadiska [20].

4.2.9 Nefunkčné testovanie

V tomto prípade ide o testovanie kvalitatívnych charakteristík alebo nefunkčných vlastností systému. Zaujíma nás ako dobre alebo, ako rýchlo sa niečo vykonalo. Testujeme niečo čo potrebujeme zmerať, napríklad čas odozvy. Nefunkčné testovanie a rovnako aj funkčne sa vykonávajú na všetkých úrovniach testovania [20].

4.3 Modely životného cyklu vývoja

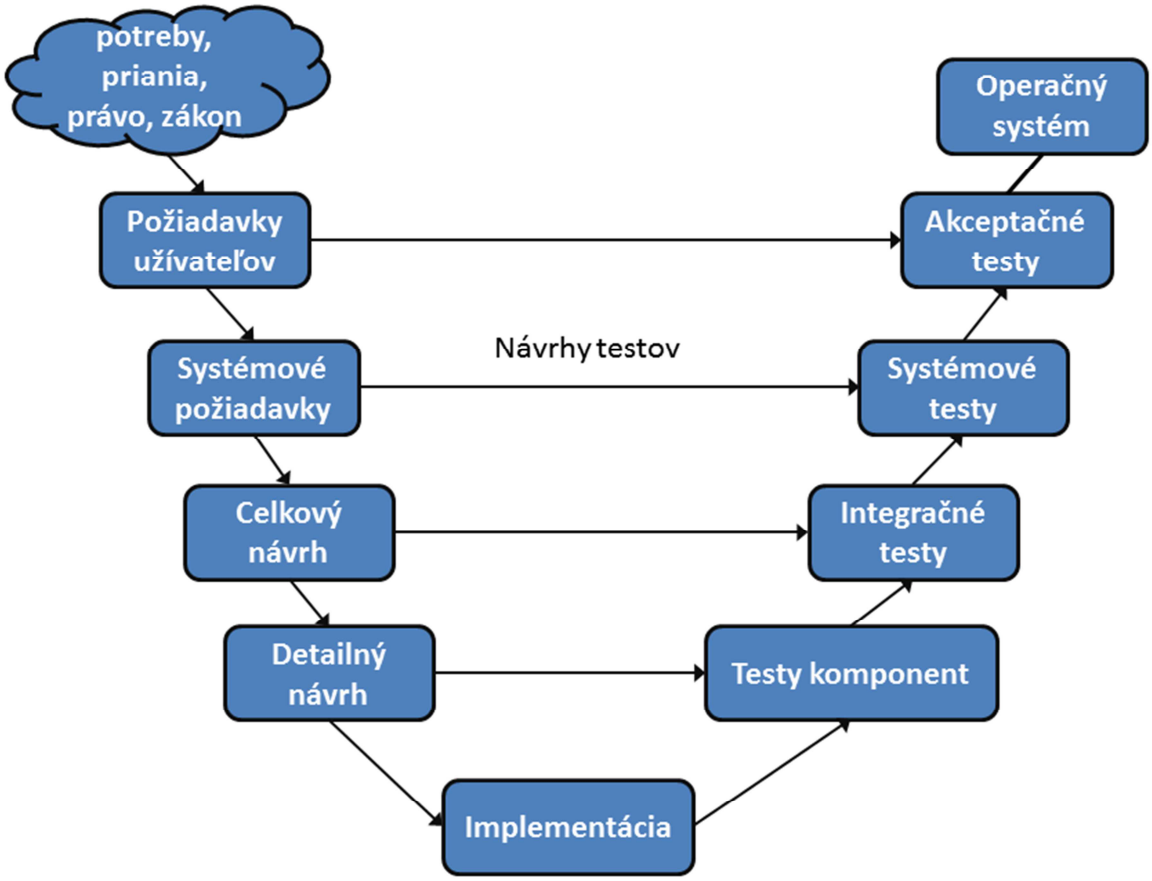
Testovanie nie je samostatnou činnosťou, má svoje miesto v rámci modelu životného cyklu softvéru. Cyklus vývoja softvéru teda do značnej miery určuje jednotlivé fáze testovania. V dnešnej dobe existuje mnoho životných cyklov, ktoré boli vyvinuté za účelom dosiahnutia rôznych požadovaných cieľov. Medzi jedny z najčastejších používaných modelov vývoja softvéru patria V-model a Inkrementálny model. Táto práca bude pojednávať o V-modeli a s ním spojenými úrovňami testovania [20].

4.3.1 V-model

V-model bol vyvinutý zo zámerom vyriešiť niektoré problémy, ktorým čelí tradičné využitie vodopádového prístupu. Problémom bolo neskoré objavenie chýb v dôsledku zapojenia testovania do cyklu vývoja až na jeho konci. V-model, preto definuje prístup, pri ktorom sa testovanie musí začať čo najskôr v priebehu životného cyklu. Ide o jeden zo základných princípov štruktúrovaného testovania. Výstupy práce programátorov a obchodných analytikov počas vývoja produktu sú základom pre testovanie v jednej alebo viacerých úrovniach. Zahájením testovania návrhu čo najskôr, je možné často objaviť chyby priamo už v základných testovacích dokumentoch (angl. test basis documents)[7]. V-model znázorňuje, ako môžu byť testovacie činnosti integrované do každej fáze životného cyklu.

Existujú rôzne varianty V-modelu, ale za bežný typ sa považuje model používajúci štyri úrovne testovania, z ktorých každá má svoje vlastné ciele. V praxi môže teda V-model obsahovať viac alebo menej rôznych úrovní vývoja a testovania, záleží to na projekte a softvérovom produkte.

[7] Sú definované ako zdroj informácií alebo dokumentov, ktoré sú potrebné k písaniu testovacích prípadov a tiež pre analýzu testov.



Obrázok 4.3: Základný V-model [20]

4.4 Úrovne testovania

Táto kapitola sa zaoberá podrobnejším popisom jednotlivých úrovní testovania vo V-modeli. Kľúčové vlastnosti každej úrovne sú popísané a definované, aby bolo možné jasnejšie rozlíšiť jednotlivé úrovne.

4.4.1 Testy komponent

Tiež známe pod názvom jednotkové testovanie, testovanie modulov a programové testovanie. Princípom je hľadanie chýb a overenie funkčnosti softvéru v častiach, ako napríklad v moduloch, objektoch, triedach a mnoho ďalších, ktoré je možné samostatne otestovať.

Testovanie komponent môže byť vykonávané v izolácii od zvyšku systému v závislosti na kontexte životného cyklu vývoja a systému. Pre simuláciu rozhrania medzi softvérovými komponentmi a nahradenie chýbajúcich časti softvéru jednoduchým spôsobom sa najčastejšie používa koreň (angl. stub) a ovládač (angl. driver). Koreň je volaný softvérovými komponentmi, aby bol otestovaný a ovládač volá komponenty, ktoré majú byť otestované.

Testovanie komponent zahŕňa funkčné, ale aj špecifické prípady nefunkčného testovania, ako je využitie zdrojov v systéme, výkonu a testovanie robustnosti, ale aj štrukturálne testovanie. Testovacie

prípady sú odvodené z výstupných dokumentov vývoja ako sú návrh softvéru alebo dátový model [20].

4.4.2 Integračné testy

Integračné testy testujú rozhranie medzi komponentmi, interakciu rôznych časti systému, ako je operačný systém, súborový systém a hardvér alebo rozhranie medzi systémami. Integračné testovanie by malo byť oddelené od ostatných integračných aktivít. Môže existovať viac ako jedna úroveň testovania integrácie a môže byť vykonávaná na testovacích objektoch rôznej veľkosti.

U rozsiahlych integrácií je ťažké izolovať zlyhanie na konkrétnom rozhraní. Na základe toho boli vyvinuté prístupy k testovaniu integrácie. Jedným z nich je „big-bang“ testovanie, ktoré ráta s tým, že pred začiatkom testovania sú už všetky komponenty vytvorené. Nevýhoda tkvie v časovej náročnosti a je zložité sledovať príčiny porúch, pri takto neskorej integrácií. Big-bang testovanie je optimistické plánovanie a očakávanie, že sa neobjavia žiadne chyby.

Ďalším prístupom je inkrementálne testovanie, kde všetky časti systému sú integrované jeden po druhom a testovanie sa vykonáva po každom kroku. Pri inkrementálnom testovaní výhoda spočíva v objavení chýb už na začiatku, kedy je relatívne ľahké zistiť príčinu. Nevýhodou je opätovne časová náročnosť, potrebná na vytvorenie koreňov a ovládačov [20].

4.4.3 Systémové testy

Systémové testy sa zaoberajú správaním celého systému, ako je stanovené v rámci vývoja projektu alebo produktu. To zahŕňa testy založené na rizikách a špecifických požiadavkách, obchodných procesoch, prípadoch užitia alebo na základe iných definovaných popisoch a systémových zdrojoch. Systémové testy sú najčastejšie jedny s posledných testov prebiehajúcich počas vývoja produktu. Slúžia na overenie, že systém vyhovuje špecifikácií s úmyslom objaviť čo najviac chýb.

Systémové testy by mali skúmať, ako funkčné tak aj nefunkčné nároky systému. Testovanie vyžaduje kontrolované prostredie pre vykonávanie testov, s ohľadom na kontrolu verzií softvéru a testovacích dát. Testovacie prostredie by malo odpovedať čo najviac finálnemu produkčnému prostrediu [20].

4.4.4 Akceptačné testy

Ak systémové testy prebehnú v poriadku a podarí sa opraviť všetky alebo väčšinu chýb je možné systém doručiť zákazníkovi na akceptačné testovanie. Akceptačné testovanie by nám malo odpovedať na nasledujúce otázky: „Môže byť systém vypustený do produkčného prostredia?“, „Splnil vývoj svoje záväzky?“, „Sú vyriešené všetky riziká?“.

Akceptačné testovanie je väčšinou na zodpovednosti užívateľov alebo zákazníkov. Pre vykonanie testov je potreba produkčného prostredia. Zmyslom akceptačného testovania je budovať dôveru v systém. Zameriava sa na validačné testy, kde sa snažíme zistiť, či je systém vhodný pre daný účel. Hľadanie chýb by nemalo byť primárnym cieľom akceptačného testovania [20].

5 Automatizované testovanie softvéru

Testovanie softvéru je drahý a náročný proces vyžadujúci až 50% z nákladov vynaložených na vývoj softvéru, spolu s rozsiahlym využitím ľudských zdrojov [21]. Dôsledkom toho by malo byť jedným z cieľov testovania aj jej automatizácia, teda zníženie nákladov.

Základom pribežnej integrácie je automatizácia všetkých procesov, a tým aj testovania. Bez automatizovaných testov je ťažké pre vývojárov alebo iných zainteresovaných pracovníkov mať dôveru v softvérové zmeny. Snahou priebežnej integrácie je automatizovať všetky štyri úrovne testovania vo V-modeli.

Kapitola ďalej popisuje proces automatizácie, spolu s jej výhodami a nevýhodami. V závere kapitoly sú spísané dostupné nástroje určené pre automatizáciu testov.

5.1 Automatizácia testov

Pred samotným zavedením automatizácie by mala byť vykonaná analýza na základe, ktorej by sa mali zhodnotiť prínosy plynúce z jej využitia.

Prvým dôležitým faktorom je etapa, v ktorej sa projekt aktuálne nachádza. U systémov, ktoré sú už v prevádzke, je aplikácia automatizácie zložitá, ale nie neuskutočniteľná. Najvhodnejšia fáza pre vytváranie automatizovaných testov je už počas návrhu a vývoja softvéru. Architektúra aplikácie musí umožňovať automatizáciu testov, a preto je potreba s týmto rátať už od počiatku projektu.

Obecne sa pomocou automatizácie snažíme pokryť testami čo najväčšiu časť aplikácie. S aplikáciou automatizácie sa prevažne počíta u rozsiahlejších projektoch s dlhodobejšou životnosťou (roky, viac verzií), kde vidina optimalizácie je najreálnejšia. Pri vytváraní testov je potreba zohľadňovať aj častú aktualizáciu testovaného kódu.

Nasledujúce body popisujú proces automatizáciu testov [22]:

1. Výber vhodného nástroja pre automatizáciu.
2. Návrh testov. Testy musia byť naplánované. Nástroje pre automatizáciu umožňujú špecifikovať testy pred ich implementáciou.
3. Kontrola navrhnutých testov. Stále by sme mali byť schopný vykonať tieto testy manuálne. Vyberieme test, ktorý má pre nás najvyššiu prioritou, pretože predstavuje najväčšiu pravdepodobnosť, že prejdem systémom. Ostatným testom priradíme priority. Potom skontrolujeme, či sa nami zvolené priority zhodujú s prioritami vývojárov, marketingového oddelenia a užívateľov.
4. Automatizujeme a spustíme test, predstavujúci šablónu, ktorá bude môcť byť kopírovaná a upravovaná na základe rôznych testovacích scenárov.
5. Spustíme ďalšie automatizované testy podľa priorít.
6. Zvážime použitie nástrojov na profilovanie pre zistenie aká časť kódu je skutočne pokrytá automatizovanými testami.
7. Zdokumentovanie testov a výsledkov.

5.2 Výhody a nevýhody automatizácie

Samotná automatizácia prináša mnoho výhod, ale taktiež disponuje pár nevýhodami. Pri posudzovaní týchto vlastností je treba brať ohľad aj na konkrétny projekt. Automatizácia je vykonávaná na základe dostupných nástrojov alebo skriptov vykonávajúcich zautomatizované manuálne procesy. Medzi základné výhody takýchto nástrojov a teda automatizácie patrí [23]:

- **Rýchlosť a časová úspora –** nástroje sú všeobecne rýchlejšie a pracujú nonstop. Ak je automatizácia aplikovaná na správny projekt v správnom čase, náklady klesajú. Taktiež umožňuje uvoľnenie ľudských zdrojov.
- **Lepšia presnosť –** nástroje sú schopné zmerať to, čo si človek nemusí všimnúť. A nezanášajú chyby do testovania, ako u ľudského zásahu.
- **Stabilná kvalita testovania –** každé testovanie je vykonávané rovnakým spôsobom.
- **Automatické logovanie a vytváranie správ –** získanie rýchlej spätnej väzby v požadovanom formáte.
- **Zjednodušenie regresného testovania –** rýchlejšie vykonávanie dovoľuje rozsiahlejšiu regresiu, čo podporuje rýchlejšie pridávanie funkcií, opravu chýb, atď.
- **Lepšie pokrytie testami –** rýchlejšie testovanie znamená viac priestoru pre ďalšie testy.
- **Vykonanie „nemožných“ testov –** existujú dané testy, ktoré tester nie je schopný vykonať, napríklad záťažové testy.
- **Spoľahlivejšie a kvalitnejšie odhady –** vďaka automatizácii sú k dispozícii údaje pre analýzu procesov a informácie o možných vylepšeniach.

S využitím automatizácie sú spojené aj nasledujúce nevýhody a náklady [22], [23]:

- **Viac času, peňazí a zdrojov –** automatizované testovanie môže byť drahšie ako manuálne. Je potreba zabezpečiť nástroje vhodné pre automatizáciu a vyškoliť užívateľov. Potom je nutné vytvoriť testy a starať sa o ich aktualizáciu.
- **Neexistencia testovacích procesoch –** pre použite nástrojov na automatizáciu je treba špecifikovať testovacie procesy.
- **Chýbajúce požiadavky a špecifikácie testov –** dôležité je navrhnúť prípady užitia, a mať jasné požiadavky na testovanie.
- **Nekompatibilita s ostatnými nástrojmi.**
- **Potreba vlastného nástroja** – v niektorých príkladoch nie je možné využiť žiadny z dostupných nástrojov.
- **Náklady na vyhodnotenie výsledkov**.

5.3 Nástroje pre automatizáciu testovania

V súčasnej dobe existuje veľké množstvo aplikácií a rámcov umožňujúcich automatizáciu testovania. Nástroje rozdelíme do dvoch skupín. Prvá skupina sa zaoberá rámcami z rodiny *xUnit*, podporujúcich tvorbu, správu a spúšťanie automatizovaných testov v programovacom jazyku PHP. Ďalšou skupinou sú nástroje automatizujúce webové prehliadače, čím môžeme simulovať chovanie užívateľa na stránkach. Selenium nástroje sú momentálne jedným z najlepších a najpoužívanejších riešení.

5.3.1 PHPUnit

PHPUnit je voľne dostupný aplikačný rámec pre tvorbu automatizovaného testovania napísaný v jazyku PHP šírený pod licenciou BSD, ktorý vytvoril Sebastian Bergmann. Pôvodne bol vytvorený ako port k *JUnit*. Hoci sú k dispozícii aj iné rámce podporujúce jazyk PHP, PHPUnit sa stal prakticky štandardom.

Jedným z cieľov PHPUnit je, aby bolo možné testy organizovať a vytvárať z nich testovacie súpravy (angl. test suites). Vďaka tomu sme schopný spúšťať ľubovoľný počet alebo kombináciu testov dohromady, napríklad testy pre celý projekt alebo len testy pre danú triedu.

Za najväčšiu výhodu PHPUnit sa považuje využitie falošných objektov (angl. mock objects). Falošné objekty umožňujú nahradiť závislosti medzi testovacím kódom a objektom, ktorý ma preddefinované správanie. Správnym použitím falošných objektov sa vieme uistiť, že testujeme len potrebný kód a nie iný.

Medzi ďalšie výhody sa považuje funkcia na zistenie pokrytia kódu danými testami. Pomáha to zabrániť falošnému pocitu bezpečia. PHPUnit dokáže generovať správu o pokrytí testami v rôznych formátoch, ako napríklad v HTML.

PHPUnit je schopné vygenerovať kostru triedy na základe testovacieho prípadu. Tento proces vie aj zvrátiť a vygenerovať kostru testu na základe triedy.

Vytvorené testy je možné spúšťať pomocou príkazového riadku alebo integrácie do jedného z mnohých podporovaných vývojových prostredí. Spúšťanie testov môže byť začlenené aj do systémov priebežnej integrácie, ako napríklad Jenkins.

PHPUnit podporuje množstvo rozšírení, ktoré umožnia písať aj špecializované testy. Patrí sem napríklad práca s databázou, testovanie webových aplikácií pomocou Selenia, profilovanie pomocou XHProf alebo vykonávanie testov paralelne [24].

5.3.2 Nette Tester

Ide o ďalší z dostupných nástrojov z rodiny *xUnit*, napísaný v jazyku PHP. Nette Tester vznikol ako testovací nástroj pre Nette aplikačný rámec. Za tvorbou stojí David Grudl a je voľne šíriteľný pod licenciou BSD.

Nette Tester je oproti svojej konkurencii intuitívnejší, disponuje minimalistickým programovým rozhraním. Každý vytvorený test je samostatný PHP skript, ktorý je možné ľahko odladiť.

Nette Tester disponuje podobnými funkčnosťami ako ostatné nástroje. Ide o kontrolu pokrytia kódu, podporu databáz a tvorbu testovacích prípadov. Testy je možné spúšťať cez príkazový riadok a momentálne je dostupné len rozšírenie pre vývojové prostredie NetBeans.

Oproti PHPUnit má v sebe už primárne zabudované paralelné vykonávanie testov. Každý test teda beží vo vlastnom vlákne, čím sa maximalizuje izolácia testov.

Zaujímavým riešením je beh Nette Testeru na pozadí, pričom sleduje zmeny vo zvolenom adresári. Na základe zmien je schopný testy znova spustiť. Takéto riešenie sa hodí pri tvorbe a ladení testov.

Medzi základne nevýhody patrí absencia predvoleného nástroja pre tvorbu falošných objektov. Pre ich implementáciu je potreba využiť niektorý z iných dostupných nástrojov. Rovnako chýba podpora Selenia a v niektorých prípadoch je k dispozícii nedostatočný výpis informácií z testov [25].

5.3.3 Selenium

Selenium je súprava nástrojov využívaná k automatizovanému testovaniu grafického rozhrania webových aplikácií pomocou metódy čiernej skrinky. Testy je možné spúšťať vo veľkom množstve webových prehliadačoch a tiež na veľa platformách. V tom spočíva aj ich najväčšia výhoda. Samotné testy je možné písať v rôznych dostupných programovacích jazykoch. Momentálne sú podporované C#, Java, Perl, PHP, Python a Ruby. Všetky projekty Selenium sú dostupné pod licenciou Apache 2.0. Selenium sa skladá z nasledujúcich projektov Selenium IDE, Selenium RC, Selenium WebDriver a Selenium Grid. V súčasnosti je k dispozícii Selenium 2, v ktorom Selenium RC už nie je naďalej vyvíjané, bolo nahradené WebDriverom, ale kvôli spätnej kompatibilite sa stále využíva.

Funkciou Selenia IDE je vytvárať testovacie skripty. Ide o nástroj integrovaný do prehliadača Firefox. Vytváranie testov je jednoduché a prebieha pomocou zaznamenávania interakcie užívateľa s prehliadačom. Činnosti sú ukladané formou príkazov do tabuľky. Každý príkaz musí vždy obsahovať tieto tri položky:

- command – príkaz, ktorý sa má realizovať,
- target – cieľový element príkazu,
- value – definuje výstupnú hodnotu príkazu.

Vytvorené testovacie scenáre je možné vyexportovať do jedného z podporovaných programovacích jazykov.

Selenium WebDriver prišiel ako náhrada za Selenium RC. V prípade Selenia RC šlo o testovací nástroj na báze vzdialene ovládaného serveru vykonávajúcom testy nad protokolom HTTP. Jednotlivé príkazy testu boli interpretované do JavaScriptu a vykonané v prehliadači. WebDriver disponuje jednoduchým a priateľským programovacím rozhraním. Cieľom nástroja je podpora dynamických webových stránok, v ktorých sa prvky môžu často meniť. Komunikáciu priamo s webovými prehliadačmi zabezpečuje pomocou ich natívnej podpory automatizácie na rozdiel od Selenia RC, ktorý prehliadaču predsúva interpretovaný JavaScriptový kód.

Selenium Grid je schopný spúšťať testy na rôznych strojoch voči rôznym prehliadačom paralelne. Tím sa mnohonásobne zrýchli vykonávanie testov spolu s eventualitou otestovať aplikáciu v rôznych produkčných prostrediach [26].

6 Analýza požiadaviek a problémov pri vývoji softvéru v spoločnosti Logio

Spoločnosť Logio vyvíja od roku 2004 softvérový produkt s názvom Planning Wizard. Účelom produktu je poskytnúť efektívne riadenie materiálových tokov a zásob. Počas desaťročného vývoja sa na projekte vystriedalo veľa programátorov čo malo z danej časti neblahý vplyv na softvér. V kapitole sú spísané problémy a požiadavky, ktoré vedú k zavedeniu metodiky priebežnej integrácie a potrebnému testovaniu.

6.1 Analýza problémov

Za dobu vývoja sa nahromadili isté problémy, s ktorými sa v súčasnosti stretávame. Spolu s tým aká je rozsiahlosť projektu a počtu ľudí, ktorý sa na ňom podieľajú, bolo len otázkou času, kedy bude potrebné sa zamerať na jednotlivé problémy.

Medzi prvé problémy by som zahrnul absenciu podrobnejšej dokumentácie. Firma disponuje vlastnou on-line encyklopédiou, ale bohužiaľ s postaršími informáciami. Aktuálne sa pracuje na obnove. Rovnako na tom je aj zdokumentovanie kódu. Počas desiatich rokoch vývoja sa na projekte vystriedalo veľa programátorov a dokumentovanie kódu sa úplne nedodržiavalo. Momentálne pre vybrané časti kódu nie sú k dispozícii ľudia, ktorý na nich pracovali a spolu s chýbajúcou dokumentáciou je potreba prechádzať celý kód čo ma za následok spomalenie vývoja. S tým súvisí aj problémom s rýchlim vyškolením nových zamestnancov. Firma pomaly naberá na počte pracovníkov a rýchle zaučenie je kľúčovým bodom.

Ďalším priamo súvisiacim problémom je množstvo už nepoužiteľného a nevyužívaného kódu. Určité moduly a algoritmy sa už nepožívajú a len zbytočne zneprehľadňujú kód.

Dôležitým aspektom je doposiaľ neprítomnosť testovania. Problémom pri vývoji je čas potrebný na objavenie chýb. Jednotlivé chyby sú zanášané do kódu a k ich objaveniu dôjde až po dlhšom čase. Rovnako je problém identifikovať kto chybu do kódu vložil.

Pri vývoji produktu sa pracuje s dvoma verziami kódov. Ide o hlavné jadro produktu obsahujúce základ, následne je špecifický kód pre každého zákazníka. V rámci špecifického kódu sa jednotlivé funkčnosti preťažujú zo základu. Chyba nastáva pri duplicite kódu, kedy sa nevyužíva alternatíva preťaženia alebo sa zbytočne vytvára už existujúca funkčnosť.

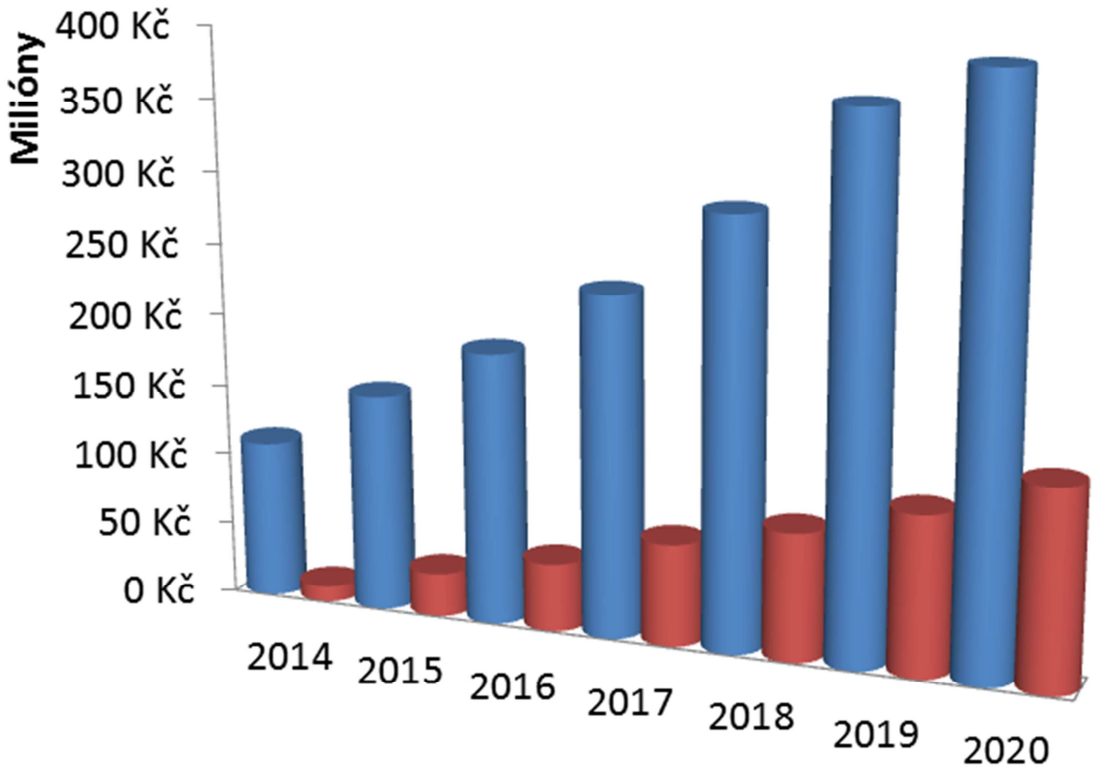
Zhrnutie základných problémov:

- chýbajúca rozsiahlejšia dokumentácia,
- neokomentovaný kód,
- absencia testovania,
- nemožnosť rýchleho zaučenia nových pracovníkov,
- duplicita, nevyužitie a zastaranosť kódu.

6.2 Analýza požiadaviek

Požiadavky primárne vychádzajú z problémov spísaných vyššie. Medzi najzákladnejšie potreby patrí zrýchlenie, skvalitnenie a zlacnenie vývoja. Ide o jedny z najdôležitejších nárokov u firiem zaoberajúcich sa vývojom softvéru.

Spoločnosť Logio očakáva v nasledujúcich rokoch výrazne zvýšenie zisku a príjmu. Predpokladá sa každoročné zvýšenie zisku o 30% a príjmu o 25%.



Graf 6.1: Predpokladaný zisk a príjem za obdobie 2014-2020 [vlastné]

Tým pádom je potreba implementovať mnohonásobne viac zákaziek ročne. To je možné zabezpečiť rýchlejším vývojom a implementáciou. Momentálne je k dispozícii len celkové oddelenie vývoja, kde sa nerozlišuje medzi implementáciou projektov a vývojom nových vecí. Do budúcnosti sa preto ráta s rozdelením oddelenia na samostatný vývojový a implementačný tím čo bude mať veľmi veľký vplyv na rýchlosť vývoja. Ďalšou alternatívou ako zrýchliť vývoj je disponovať automatizovaným a manuálnym testovaním, a rovnako zrozumiteľným a čitateľným kódom.

Pre zabezpečenie čo najkvalitnejšieho kódu je potreba kód testovať a refaktorovať. Produkt Planning Wizard momentálne nedisponuje dostatočným testovaním. Požaduje sa teda zabezpečiť základne testovanie komponent ako sú triedy, metódy, atď. Produkt tiež obsahuje zložité algoritmy, ako je výpočet objednávok či predpoveď predaja. Zámerom je tieto algoritmy zabezpečiť pred chybami, ktoré by pozmenili ich logiku. Testovanie by malo tiež pomôcť pri refaktorizácii kódu, ktorá je v súčasnej dobe vyžadujúca. Pre väčšiu prívetivosť voči zákazníkom je vhodné produkt podporovať aj na iných webových prehliadačov ako len pre Firefox, čím je potreba ďalšieho testovania.

Systém Planning Wizard stojí na každodennom prepočte na základe, ktorého zákazníci získavajú najaktuálnejšie údaje. Spravidla k prepočtu dochádza každý večer a je potrebné, aby nové dáta boli k dispozícii čo najskôr a prepočet prebehol bez chýb. U istých projektoch s veľkým množstvom dát sú prepočty časovo rozsiahle. Zámerom je teda sledovať ich časový vývoj, aby neprekročili kritickú hodnotu a zabezpečiť ich bezproblémový chod.

Na kvalitu a čitateľnosť kódu má vplyv štýl tvorby programátora. Každý používa svoje techniky, ktoré je potreba zjednotiť do uceleného formátu. Využívanie jednotlivých trendov je možné sledovať pomocou statickej analýzy, ktorá je jednou z požiadaviek.

Dôležitým aspektom je aj rýchla reakcia na časté potreby zákazníkov. V súčasnosti proces zahrnutia zmien stojí na ich implementácii, potom je zmena ihneď nasadená alebo až po nejakej dobe je väčšie množstvo zmien dodané zákazníkovi k dispozícii. Snahou je tento proces zefektívniť.

Akceptačné testovanie produktu dodnes prebieha nasadením produktu k zákazníkovi do jeho produkčného prostredia a sprístupnenie ho základným užívateľom. O sprístupnenie nových funkčností sa musí starať priradený implementačný programátor. S využitím priebežnej integrácie prichádza príležitosť využiť automatického nasadenia produktu priamo po vykonaní zostavenia a tým znížiť vyťaženie programátorov.

Zhrnutie základných požiadaviek:

- zrýchlenie a zefektívnenie vývoja,
- doplnenie základného testovania (jednotkové testy),
- refaktorizácia kódu,
- potreba testovať nové a stávajúce algoritmy,
- testovanie grafického rozhrania,
- kontrola prepočtu systému,
- rýchla reakcia na potreby zákazníka.

6.3 Predpoklady pre zavedenie priebežnej integrácie

Odpoveďou na problémy a požiadavky spoločnosti Logio je princíp priebežnej integrácie. Hodnoty a ciele plynúce z nej v kapitole 2.2 sa zhodujú s potrebami spoločnosti. Medzi dôvody použitia priebežnej integrácie v spoločnosti Logio patria:

- chybovosť v kóde,
- nájdenie chýb je časovo náročné,
- absencia testovania,
- absencia statickej analýzy,
- potreba refaktorizácie,
- zostavenie softvéru je zložité a vykonávané každý deň,
- nedostatočný prehľad o projektoch,
- automatizácia opakujúcich procesov,
- zníženie nákladov (čas, kapacita) na implementáciu projektu.

V prospech zavedenia princípu priebežnej integrácie v spoločnosti sú nasledujúce body:

- Programátori majú k dispozícii centrálne úložisko pre zdrojové kódy.
- K ukladaniu dát na úložisko dochádza aspoň raz za deň.
- Pred uložením kódu na repozitár dochádza k otestovaniu zostavenia.
- Najväčšiu prioritu majú opravy chýb.
- Finančné nároky na zavedenie priebežnej integráciu nehrajú rolu.

7 Návrh implementácie priebežnej integrácie a testovania

Na základe požiadaviek a možností spoločnosti Logio vznikol nasledujúci návrh pre implementáciu priebežnej integrácie spolu so špecifickým testovaním.

7.1 Návrh testovania

Na základe požiadaviek je vhodné testy rozdeliť na statické a dynamické, spolu s možnosťou spúšťať testy lokálne na vývojovom stroji programátora a automatizovane na stroji s priebežnou integráciou. U statického testovania pôjde o analýzu kódu pomocou dostupných nástrojov určených pre programovací jazyk PHP. Tieto programy dokážu analyzovať napríklad metriky kódu, skontrolovať kódovací štandard alebo duplicitu kódu.

Dynamické testy rozdelíme na:

- jednotkové testy – spúšťajú lokálne a na integračnom stroji,
- regresné testy – vykonávajú lokálne a na integračnom stroji,
- testovanie grafického rozhrania – na integračnom stroji,
- kontrola prepočtov systému Planning Wizard – na integračnom stroji.

Jednotkové testy budú predstavovať klasické testy základných jednotiek zdrojového kódu (triedy, metódy), ktoré je možné testovať samostatne. Tieto testy sú najlepšou formou pre automatizáciu a ich využitie v priebežnej integrácií sa považuje za základ. Na implementáciu testov sa využije dostupný aplikačný rámec schopný vytvárať a spravovať jednotkové testy. Regresné testovanie je spísané nižšie v nasledujúcej podkapitole 7.1.1.

Testovanie grafického rozhrania bude metódou čiernej skrinky. Funkciou bude simulácia chovania reálneho užívateľa a predísť tak chybovým oznámeniam počas reálnej prevádzky. Tiež bude využité pre otestovanie produktu Planning Wizard na rôznych webových prehliadačoch.

Prepočet produktu Planning Wizard je najkritickejšou časťou systému, ktorá zabezpečuje jeho správnu funkčnosť na každý deň. K prepočtu teda dochádza každý večer, preto je potrebné zabezpečiť jeho bezchybnosť. Požadované je zobraziť čas prepočtu, aby sme boli schopný porovnať ich časový vývoj. Taktiež je potreba zabezpečiť správnosť výstupu. Úmyslom je teda sledovať rôzne oznámenia, výnimky a chyby, ktoré by mohli počas prepočtu nastať. Testovanie prepočtov bude vykonávané na integračnom serveri. Na základe získaných údajov o čase a chybových výstupov bude daný projekt optimalizovaný a zefektívnený. Účelom je postupne takto prejsť všetky doteraz reálne bežiace projekty.

7.1.1 Regresné testy

Pri regresných testoch pôjde o testovanie zložitých algoritmov, ako je napríklad výpočet objednávok alebo vlastností dodávateľov. Zámerom je zabezpečiť, že zmeny v kóde nemajú vplyv na dané výpočty. Pre zaistenie správnej funkčnosti algoritmov budeme potrebovať dáta, ktoré do výpočtu vstupujú a tiež výstupné dáta. Tieto dáta si nazveme pahýľ dáta (angl. stub data). Proces testu si predstavíme napríklad na výpočte objednávok pre daný produkt a mal by byť nasledujúci:

1. Musíme disponovať prvotným dátovým pahýľom, v ktorom považujeme výsledky z algoritmu za správne.
2. Napríklad po refaktorizácii kódu výpočtu objednávok je potreba otestovať stálu funkčnosť algoritmu.
3. Preto na základe vstupných údajov z prvotného pahýľa spustíme výpočet objednávok pre daný produkt.
4. Výsledok výpočtu porovnáme s výsledkom z pahýľa. Ak sa výsledky zhodujú znamená to, že refaktorizácia prebehla v poriadku.

Porovnávanie výsledkov testov a hodnôt v dátových pahýľoch bude prebiehať na základe druhu algoritmu. Bude možné hodnoty porovnávať na základe rovnosti.

Štruktúra dátového pahýľa bude nasledujúca, pôjde o asociatívne pole ukladané do databáze:

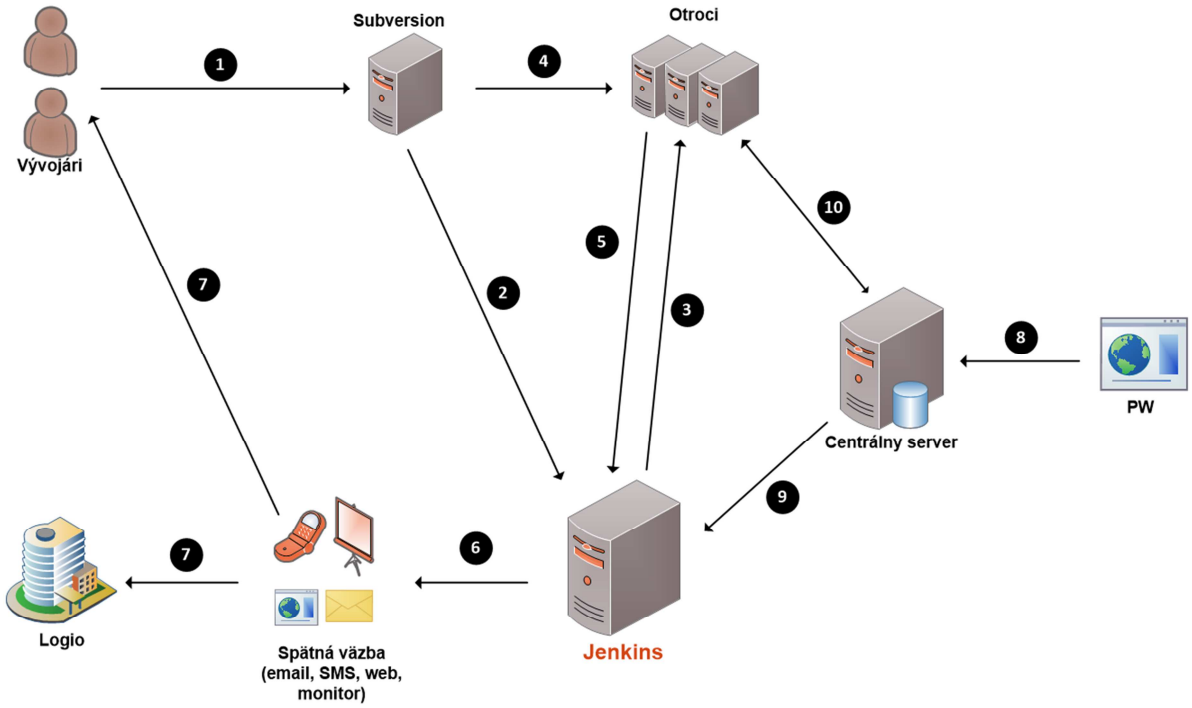
1. **Metadáta** – základné informácie o dátovom pahýli. Informácie budú uložené ako samostatné stĺpce v databáze:
   1.1. identifikačné číslo,
   1.2. názov projektu,
   1.3. názov súboru s testom,
   1.4. názov testovanej metódy,
   1.5. identifikácia testu,
   1.6. identifikácia užívateľa,
   1.7. dátum vytvorenia.

2. **Dáta** – pôjde o vstupné dáta potrebné pre vykonanie výpočtu algoritmu. Budú vo formáte *identifikácia inštancie triedy => dáta triedy* a u dát z databázy *názov tabuľky => skript tabuľky*. V databázy budú uložené ako dátový typ BLOB.

Jednotlivé pahýle bude možné vytvárať automaticky v grafickom rozhraní aplikácie a potrebné údaje sa získajú automaticky. Pahýle sa budú ukladať na centrálny server obsahujúci ich databázu. Staré pahýle sa nebudú mazať a za aktuálne správny sa bude vždy považovať posledný vytvorený.

7.2 Návrh systému priebežnej integrácie

Pre implementáciu priebežnej integrácie sa vo firme rozhodlo použiť dostupný softvér s názvom Jenkins. Nástroj Jenkins bol vybraný pre svoju voľnú dostupnosť a požadované funkcie, ktoré sa od integračného serveru očakávali, spolu s alternatívou rozsiahleho rozšírenia. Obrázok na nasledujúcej stránke zobrazuje základné zloženie navrhovanej architektúry priebežnej integrácie.



Obrázok 7.1: Architektúra priebežnej integrácie v spoločnosti Logio [vlastné]

Popis základných komponent architektúry:

- **Subversion –** je systém pre správu zdrojových súborov používaný v spoločnosti Logio.
- **Jenkins –** ide o aplikáciu pre implementáciu serveru priebežnej integrácie. Server Jenkins bude spravovať jednotlivé úlohy a zhromažďovať údaje o zostaveniach. Na tomto stroji nebudú prebiehať zostavenia, z dôvodu výkonu.
- **Otroci (angl. slaves) –** Jenkins disponuje schopnosťou distribuovaného zostavenia na čo slúžia práve otroci. Pomocou režimu správca/otrok (angl. master/slave) vie Jenkins spravovať veľké množstvo úloh. Môže ísť o virtuálne alebo skutočné stroje disponujúce rôznymi prostrediami pre zostavenie a testovanie. Pre diplomovú prácu je možné využiť server, ktorý bude rozdelený na menšie virtuálne stroje.
- **Centrálny server –** ide o server obsahujúci databázu dátových pahýľov.
- **PW –** ide o samotný produkt Planning Wizard. Priamo v aplikácii bude možné vytvárať dátové pahýle. Pôjde o jednoduché rozhranie, ktoré zapíše dáta na centrálny server.

Popis bodov architektúry:

1. Zmeny vykonané v kóde sú vývojármi uložené na Subversion server.
2. Jenkins sa v intervaloch pýta repozitára Subversion o zmenách v súboroch a na základe zistenia zmeny sa spustí príslušná úloha.
3. Jenkins spustí vykonávanie zostavení na otrokoch.
4. Otrok si aktualizuje zdrojové súbory zo Subversion.
5. Výsledky zo zostavení na otrokoch sú predané Jenkinsu.

6. Jenkins na základe výsledkov zo zostavení nastaví príslušný stav u úlohy (stabilná, nestabilná, neúspešná).
7. Programátori a zainteresované osoby z Logia sú informovaný o výsledkoch.
8. Pomocou rozhrania v Planning Wizard sa vytvorí dátový pahýľ a uloží na centrálny server.
9. Centrálny server môže eventuálne predať dáta Jenkinsu.
10. Otroci a centrálny server budú môcť vzájomne komunikovať pre potreby získania dát.

7.2.1 Štruktúra úloh na integračnom serveri Jenkins

V Jenkinse sa jednotlivé projekty nazývajú úlohy (angl. jobs). U rozsiahlych aplikácii ako je aj Planning Wizard sa jednotlivé zostavenia rozdeľujú do viacerých úloh, aby princíp zodpovedal priebežnej integrácií. Na základe tohto poznatku sa navrhlo nasledujúce rozdelenie úloh:

1. **PW-Commit –** táto úloha sa bude spúšťať pri každej zmene zdrojových súborov. Zostavenie by nemalo presiahnuť desať minút a v rámci neho sa budú vykonávať jednotkové testy spolu so statickou analýzou kódu.
2. **PW-Multiprojects –** úloha sa bude vykonávať na základe úspešného zostavenia úlohy PW-Commit. Pôjde o prepočet celého systému Planning Wizard s reálnymi dátami a integračnými testami, spolu s testovaním grafického rozhrania na vopred zadefinovaných projektoch.
3. **Čistý systém Planning Wizard –** pri vzniku samotného oddelenia vývoju bude vhodné disponovať takýmito úlohami obsahujúcimi čisté jadro systému so vzorovými dátami, na ktorých sa budú testovať zmeny vo vývoji.

8 Implementácia

Implementácia je rozdelená do dvoch väčších celkom a to **implementácia lokálneho (manuálneho) testovania** a **implementácia priebežnej integrácie**. Dôvodom implementácie lokálneho testovania je mať príležitosť si svoje zmeny v kóde otestovať priamo na svojom stroji a nemusieť čakať na ich automatizované vykonanie. Automatizované testy zas zabezpečia otestovanie naprieč všetkými projektmi pomocou všetkých typov testov.

Jadro testovacieho rámca (ďalej PwTester) je založené na balíku Nette Tester. Celý systém vychádza z konvencií Nette, jednotlivé triedy sú napísané ako služby, ktoré sú prepojené DI kontajnerom. Aplikačný rámec sa konfiguruje pomocou súboru *config.neon*. Súbor obsahuje dve hlavné sekcie:

- services – zoznam služieb a ich vzájomných závislostí,
- parameters – extra parametre.

Súborová štruktúra PwTesteru je nasledujúca:

- classes – obsahuje všetky triedy PwTesteru,
- config – obsahuje konfiguračné súbory,
- css – štýly pre grafické rozhranie,
- fonts – súbory s štýlmi písma pre grafické rozhranie,
- images – obrázky pre grafické rozhranie,
- jenkins – skripty a nastavenia pre spúšťanie priebežnej integrácie na Jenkinse,
- js – javascripty pre grafické rozhranie,
- mocks – obsahuje Factory triedy pre falošné objekty.

8.1 Implementácia lokálneho testovania

Lokálne testovanie umožňuje programátorom priamo si na svojom vývojovom stroji otestovať kód čo zabezpečí rýchlejší a bezchybnejší vývoj. V lokálnom testovaní sú implementované jednotkové a integračné testy. Pre účely diplomovej práce sú integračné testy implementované spôsobom vykonávania ako regresné testy pre kontrolu výpočtu objednávok.

Použité technológie:

- **Nette Tester** – je využitý pre implementáciu jednotkových a regresných testov.
- **Bootstrap**[8] – slúži pre implementáciu rozhrania pre spúšťanie testov.
- **dibi**[9] – pre prácu s databázou je použitá knižnica dibi.
- **Kdyby/Curl**[10] – slúži na jednoduché posielanie rôznych HTTP požiadavkou.

[8] Viac informácií na URL: http://getbootstrap.com/

[9] Viac informácií na URL: http://dibiphp.com/

[10] Viac informácií na URL: http://addons.nette.org/kdyby/curl

- **Mockery**[11] – ide o veľmi jednoduchú a flexibilnú knižnicu určenú pre PHP využívanú pri testovaní na vytváranie falošných objektov.
- **Finder**[12] – slúži na ľahšie prechádzanie adresárovej štruktúry na disku.
- **Google Charts**[13] – sú použité na prehľadnejšie zobrazenie dát prezentujúcich objednávky.

8.1.1 Implementácia jednotkových testov

Jednotkové testy slúžia na testovanie menších funkčných častí programu. Cieľom je, aby bolo ich písanie čo najjednoduchšie. Test typicky pozostáva z niekoľkých vstupov a výstupov, na ktorých postupne voláme funkciu a porovnávame, či sa spočítaný výsledok rovná očakávanému. Vstupom však nemusíme rozumieť len skutočné parametre funkcií, ale aj vnútorný stav triedy, prípadne iné externé závislosti, napríklad databáza, súborový systém, atď.

Aby bolo možné použiť Nette Tester je potreba zabezpečiť na testovacích strojoch minimálne verziu PHP 5.3.0. Nette Tester je do testovacieho rámca PwTester zakomponovaný ako balíček pomocou Composeru[14] spolu s ostatnými použitými nástrojmi viď Použité technológie.

Izolácia a falošné objekty

Jednotkové testy testujeme v izolácii tak, aby prípadná nefunkčnosť jedného modulu neovplyvnila ostatné. Ak má modul externé závislosti, musíme tieto závislosti nasimulovať, a to tak, aby pri každom spustení bolo prostredie testu v rovnakom, definovanom, počiatočnom stave. Inak by sa mohlo stávať, že testy občas prejdú a občas nie, v závislosti na prostredí. K tomuto cieľu je možné použiť techniku zvanú imitácia (mock). Vonkajšie prostredie teda môžeme simulovať pomocou falošných objektov, ktoré imitujú skutočné zdroje, ktoré sa správajú deterministicky podľa vopred určenými pravidlami. Testovací aplikačný rámec PwTester využíva na implementáciu falošných objektov knižnicu Mockery.

Jednotkové testy v Planning Wizard

Konvencia pomenovania a umiestnenia jednotkových testov je nasledujúca:

- testové súbory sú umiestnené v adresárovej zložke modulu, ktorý testujú,
- daná testovacia trieda je vlastne testovací prípad zaobaľujúci viac testov, pre rôzne metódy z daného modulu,
- konvencia pomenovania je *<názov-modulu>**.unit.phpt.***

Na nasledujúcej ukážke si vysvetlíme jednoduchý jednotkový test:

```
<?php

$container = require_once(__DIR__.'/tools/tester/bootstrap.php');
use PwTester;
use Tester\Assert;
```

[11] Viac informácií na URL: http://docs.mockery.io/en/latest/

[12] Viac informácií na URL: http://doc.nette.org/cs/2.3/finder

[13] Viac informácií na URL: https://developers.google.com/chart/

[14] Viac informácií na URL: https://getcomposer.org/

```php
/**
 * Ukážka jednotkového testu.
 * Test sčítania dvoch hodnôt.
 *
 * @testCase
 */
class mathUnitTest extends UnitTestCase
{
    private $ma;

    public function setupEnvironment()
    {
        $this->ma = new MathClass();
    }

    /**
     * @dataProvider dataProvider
     */
    public function testAdd($expected,$a,$b)
    {
        Assert::equal($expected, $this->ma->add($a,$b));
    }

    public function dataProvider()
    {
        return array(
            array(2,1,1),
            array(5,4,1),
            array(1,1,0),
        );
    }
}

$testCase = new addUnitTest($container);
$testCase->run();
```

Prvý riadok kódu inicializuje inštanciu kontajneru, cez ktorý sa inicializujú služby PwTesteru. Nasledujúce dva riadky s `use` deklarujú `namespace` pre jednoduchší prístup. Dve uvedené triedy by sa mali minimálne vždy použiť.

Komentár nad triedou je tiež užitočný a môže obsahovať rôzne metadáta. Prvý riadok by mal byť krátky popis celého testovacieho prípadu, ktorý je možné si zobraziť v grafickom rozhraní jednotkových testov. Dôležitá je samotná anotácia `@testcase`, ktorá umožní PwTesteru spúšťať testy paralelne.

Trieda predstavuje skupinu testov, ktoré spolu súvisia. Jednotlivé testy sú vždy verejné metódy a musia začínať slovom test ako v našom príklade `public function addTest` a typicky sa použije jedna testovacia metóda pre jednu verejnú metódu z testovaného modulu. Testovací aplikačný rámec ich na základe týchto údajov rozozná, automaticky spustí a vyhodnotí ich stavy. Samotný test predstavuje asercia `Assert::equal($expected, $this->ma->add($a,$b));`. Asercie sú tvrdenia o kóde, ktoré sa snažíme dokázať. Príkladom môže byť tvrdenie $2 == 1 + 1$. V PwTesteri sú použité asercie z balíku Nette Tester.

Keď chceme spúšťať test viac krát a s rôznymi hodnotami je nepraktické vypisovať niekoľko asercií za sebou. Vhodnejším spôsobom je využiť poskytovateľa dát (angl. data provider). V PwTesteri je k dispozícii metóda, ktorá poskytuje vstupné a výstupne dáta. Ako si môžeme všimnúť v ukážkovom príklade hodnota anotácie `@dataProvider` nad testovacou metódou obsahuje názov metódy generujúcej dáta. Metóda `dataProvider` obsahuje pole s troma zložkami a teda test sa spustí tri krát. Prvá zložka každého pod-poľa je výsledok a druhé dve zložky sú parametre v poradí, aké je uvedené v hlavičke testovacej metódy.

Metóda `setupEnviroment` sa volá pred každým spúšťaním testovacej metódy a slúži na nastavenie prostredia. U ukážky slúži na vytvorenie novej inštancie triedy `MathClass`, aby sme zaručili, že stav z minulého behu nemá vplyv na aktuálny test.

Na konci sa vždy vytvorí inštancia testu, ktorému sa predá kontajner, aby vedel zinicializovať potrebné služby a zavolá sa metóda `run`, ktorá spustí testy.

Najjednoduchšie sa testujú takzvané čisté metódy, kde:

- ich jediný vstup sú hodnoty predané ako parametre,
- ich výstup je pre rovnaké parametre vždy ten istý,
- nezávisia na stave triedy, v ktorej sú definované.

Ukážkou takejto metódy je aj príklad s ukážkovým jednotkovým testom. V prípade čistých metód nie sú potrebné falošné objekty ale s takýmito prípadmi sa v produkte Planning Wizard stretneme len občasne, preto si predstavíme spôsob vytvárania falošných objektov používaných v PwTesteri.

Ukážka jednoduchého falošného objektu reprezentujúci modul `CUtilsNdebugDb`:

```php
<?php
namespace PwTester;
use Mockery;

class UtilsNdebugDbMockFactory
{
    public function create()
    {
        $mock = Mockery::mock('CUtilsNdebugDb');

        $mock->shouldReceive('isEnabled')
             ->byDefault()
             ->andReturn(false);
        return $mock;
    }
}
```

Ako môžeme vidieť v ukážke trieda reprezentujúca falošný objekt by mala obsahovať príponu `MockFactory`. Trieda musí obsahovať metódu `create`, ktorá daný objekt vytvára. Pomocou `Mockery::mock()`, kde ako parameter je predaný názov triedy, z ktorej chceme vytvoriť falošný objekt. Následne musíme špecifikovať, ako sa má inštancia správať, ak na nej zavoláme nejakú metódu. To dosiahneme volaním metód `shouldReceive` a `andReturn`. V rámci ukážky ide o metódu `isEnabled`, ktorá vracia `boolean` hodnotu, a je zadefinované, že pri jej volaní sa vždy

vráti hodnota `false`. Ak zadáme viac ako jednu hodnotu, tieto sa postupne vracajú pri opakovanom volaní metódy uvedenej v `shouldReceive`. Pri vyčerpaní všetkých variant sa bude vracať stále posledná uvedená hodnota.

Falošné objekty v produkte Planning Wizard

Väčšina modulov v produkte Planning Wizard je navrhnutá podľa vzoru *Service Locator*[15], tzn. každý modul si sám získava závislosti pomocou metódy `$this->f->getModule()`. Preto sa často opakujú určité scenáre, typicky predávanie falošných objektov cez aplikačný rámec, pre ktoré je v PwTesteri špeciálna podpora.

V rámci PwTesteru sú implementované falošné objekty najčastejších modulov a ich bežných funkcií so základnými hodnotami. Falošné objekty pre jednotlivé moduly sa vymedzujú v triedach *<ModulName>MockFactory* v adresári *./mocks/*. PwTester automaticky zaregistruje všetky triedy splňujúce túto konvenciu ako služby s názvom *<modulName>MockFactory*. Továrne pre falošné objekty musia implementovať rozhranie `MockFactory`.

Pri vytvorení novej továrne pre falošný objekt je potreba túto továreň zaregistrovať v súbore *FrameworkMockFactory.php* v konštruktore. Týmto naučíme aplikačný rámec vracať štandardné falošné objekty daného modulu.

Na otestovanie modulu v Plannig Wizard je vždy potrebné získať jeho inštanciu. V PwTesteri máme dva základné spôsoby:

- vytvoríme inštanciu pomocou kľúčového slova `new` a názvu triedy implementujúcej modul,
- vytvoríme inštanciu pomocou metódy `getPwModule`.

Prvá možnosť je vhodná pri moduloch, ktoré sú nezávisle, nepotrebujú zložitý `init` a nepodriaďujú sa rámcu Planning Wizard. V momente, keď požadujeme pre modul zaistiť závislosti, použijeme metódu `getPwModule`.

K správnej inicializácii je dôležité najprv vytvoriť inštanciu triedy `CFramework` a uložiť do premennej `$this->f`. Nový falošný objekt aplikačného rámca získame cez metódu `createFrameworkMock`. Metóda `getPwModule` potom využíva túto inštanciu rámca na predávanie závislostí.

Metóda `getPwModule` má tri argumenty:

- prvý argument je názov modulu podľa konvencie v Planning Wizard,
- druhý argument určuje, či chceme na novom module volať `init`,
- tretí argument špecifikuje alternatívnu implementáciu triedy.

Metóda samovoľne vypočíta názov triedy z názvu modulu. Ak existujú špecifické preťaženia, vytvorí sa inštancia preťaženého modulu.

V hlavnom nastavení teda falošný objekt rámca získaný cez `createFrameworkMock` vracia základné falošné objekty požadovaných modulov. Ak potrebujeme špecifické správanie falošného objektu je k dispozícii trieda `FrameworkMockUtils` dostupná v `$this->fmu`, ktorá

[15] Viac informácií na URL: https://msdn.microsoft.com/en-us/library/ff648968.aspx

zjednodušuje najčastejšie operácie nad falošnými objektmi. Napríklad, ak chceme namiesto falošného objektu vrátiť skutočný modul, použijeme špecifikáciu:

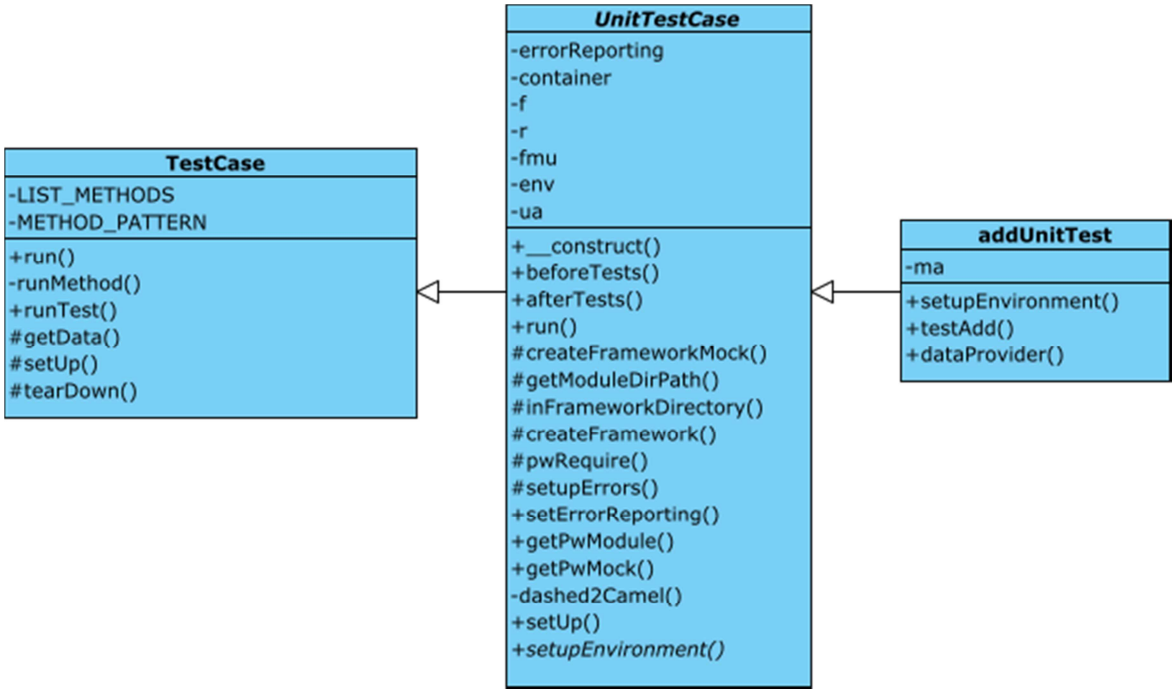
```
$this->fmu->shouldReturnPwModule($this->f, 'utils-datetime', true);
```

Ak chceme vrátiť falošný objekt modulu špeciálne upravený pre konkrétnu situáciu, použijeme volanie `shoulReturnMockModuleUsing`.

```
$this->fmu->shouldReturnMockModuleUsing($this->f,'utils-datetime',function($name, $_)
{
    $mock = Mockery::mock($name);
    $mock->shouldReceive(..)->andReturn(..);
    return $mock;
});
```

Prvý parameter v `callback` funkcii obsahuje názov požadovaného modulu, druhý volajúcu inštanciu. Na konci špecifikácie špeciálnych falošných objektov je treba vždy zavolať `$this->fmu->shouldReturnDefaultModuleMock($this->f);` aby sme naznačili, že všetky ostatné volania `getModule` majú vracať falošné objekty zo štandardných tovární.

Falošné objekty je možné tiež vytvoriť a predať do inštancie ručne pomocou metódy `getPwMock`, ktorá berie ako argument názov modulu. Tieto falošné objekty sa vytvárajú cez továrne implementujúce `MockFactory`. Taktiež je možné falošné objekty vytvoriť priamo použitím `Mock::mock`. Pri tvorbe jednotkových testov sa vychádza z nasledujúcej architektúry:

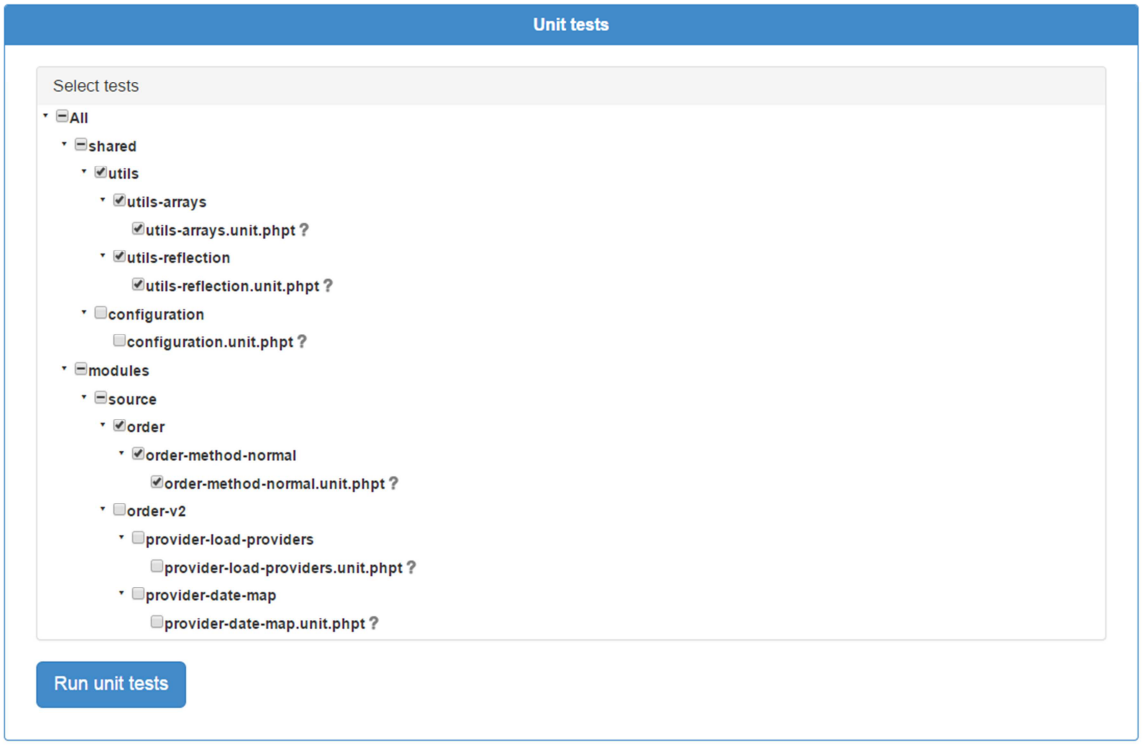


Obrázok 8.1: Diagram tried jednotkových testov [vlastné]

Pri vytváraní testovacej triedy v Nette Testeru musí byť táto trieda potomkom `Tester\TestCase`. Trieda `TestCase` zabezpečuje spúšťanie a nastavenie prostredia pre vykonanie testov. U PwTesteru sa medzi samotnou testovacou triedou a triedou `TestCase` nachádza ešte abstraktná trieda `PwTester\UnitTestCase`. Táto trieda obsahuje kľúčové nastavenia potrebné pre beh Nette Testeru pod aplikačným rámcom produktu Planning Wizard. Primárne ide o ošetrenie chýb a výjimok.

Grafické rozhranie jednotkových testov

Celé grafické rozhranie je založené na aplikačnom rámci Bootstrap vhodnom pre vytváranie moderného webu a webových aplikácií. Štruktúra rozhrania je definovaná v dvoch základných súboroch a to *TestRunner.php*, ktorý určuje logiku a *TestRunnerGUI.php* obsahujúci grafické rozhranie. Súbor *api.php* zabezpečuje rozhranie k logike a *test-runner.js* slúži na ovládanie grafického rozhrania.



Obrázok 8.2: Grafické rozhranie jednotkové testy - výber testov [vlastné]

Na obrázku vyššie je zobrazený prvok akordeón, ktorý v sebe skrýva stromovú štruktúru jednotkových testov s eventualitou vybrať si dané testy a spustiť ich. Pomocou unixového príkazu `find` je prehľadaná adresárová štruktúra produktu Planning Wizard, kde sú vyhľadané súbory s koncovkou *unit.phpt*. Ako východzie miesto sa považuje root zložka produktu, od ktorej sa odvodzuje zanorenie k daným súborom. Na stromovú štruktúru testov je použitý jquery zásuvný modul jquery-bonsai[16].

[16] Viac informácií na URL: http://aexmachina.info/jquery-bonsai

U jednotlivých súbor môžeme vidieť otázniky, ktoré sa odkazujú na dialógové okno obsahujúce informácie o danom teste. Ide o zobrazenie komentáru nachádzajúceho sa nad triedou indikujúcou testy. Ukážka sa nachádza v prílohe C.

Vyhodnotenie testov sa zobrazuje v samostatnom elemente ako môžeme vidieť na nasledujúcom obrázku.

Obrázok 8.3: Výsledok vyhodnotenia jednotkových testov [vlastné]

U jednotkových testov ide o základný formát výstupu Nette Testeru, doplnený o pár informatívnych údajov, ako je čas spustenia a druh spustených testov. Základný výstup vyhodnotenia testov je reprezentovaný symbolmi **.**, **s** a **F**. Kde **.** predstavuje test, ktorý prešiel, **s** symbolizuje preskočený test a **F** test, ktorý zlyhal. K dispozícií je aj krátky popis chyby. Na záver sú zhrnuté výsledky testov a doba trvania.

8.1.2 Implementácia integračných testov

Zmyslom integračného testovania je testovať moduly a ich vzájomné interakcie. Oproti jednotkovým testom môžu tieto testy bežať nad databázou, posielať požiadavky cez sieť, atď. Tieto predpoklady robia písanie integračných testov náročnejším, je totiž omnoho zložitejšie dosiahnuť deterministický počiatočný stav.

V rámci tejto diplomovej práce pôjde primárne o testy určené pre zložité algoritmy výpočtov ako sú napríklad výpočty objednávok. Čo sa týka testovacieho prístupu ide o regresné testy. Zámerom regresného testovania je, že pri opravách a zmenách kódu, ako je napríklad výpočet objednávok zabezpečiť, že sa do kódu nedostanú nové chyby alebo sa nevynoria už chyby vyriešené. Takéto regresné testy je veľmi vhodné automatizovať čo využijeme pri implementácií priebežnej integrácie.

Izolácia

Podobne ako pri jednotkových testoch sa na izoláciu používajú falošné objekty, u integračných testov sa používa pahýľ (angl. stub). V testovacom rámci PwTester pahýľ predstavuje dáta, ktoré sú potrebné k inicializovaniu počiatočného stavu pred testom. Typicky sa jedná o serializované stavy tried, dáta pre testovacie databáze, výsledok výpočtu a ďalšie pomocné informácie. Samozrejme sa pri integračných testoch zároveň využívajú falošné objekty, ktoré odtieňujú test od častí, ktoré nie sú zaujímavé a na daný test by nemali mať žiadny efekt.

Jednotlivé pahýle sa ukladajú na databázový server odkiaľ sú prístupné pre testy. O obsluhu ukladania sa na strane databázového serveru stará skript *uploadstub.php*, ktorý disponuje aj grafickým rozhraním, v prípade nahrávanie pahýľu ručne na server. Ich vytváranie je možné pomocou modulu *utils-stub-button.php*, ktorý integruje do grafického rozhrania tlačítko. Toto tlačítko zabezpečí vytvorenie a uloženie pahýľu na databázový server.

Záznam pahýľu v databázy je vyobrazený na nasledujúcom obrázku nižšie. Štruktúra pahýľu je:

- **id** – identifikátor pahýľu,
- **project** – názov projektu, pre ktorý bol vytvorený pahýľ,
- **testfile** – názov súboru s testom,
- **method** – metóda, ktorá sa testuje,
- **identifier** – identifikátor pre objekt, ktorý slúži ako vstup pre testovaný algoritmus,
- **author** – meno užívateľa, ktorý vytvoril pahýľ,
- **date** – čas a dátum vytvorenia pahýľa,
- **data** – dáta potrebné pre počiatočnú inicializáciu a zároveň obsahuje výstupy algoritmu, ktoré sa považujú za správne a slúžia ako základ pre porovnanie. Obsah závisí na teste. Tieto dáta sú v databáze komprimované pre ušetrenie miesta a sú vo formáte JSON.

id	project	testfile	method	identifier	author	date	data
2617	adler	order-method-normal.compute.phps	CAbstractOrderMethod::compute	1743	Michal	2015-04-07 15:24:24	0xECFD6BAF6349921D88FE958BFC1CB9E16E66FEEA6F…
2719	coopcz-cb	order-method-by-basestock.compute.phps	CAbstractOrderMethod::compute	565182	Michal	2015-05-08 21:13:13	0xEDBD6F73E3C891EEFB5DFA357B83F51F9877F678E75…
2519	orders-refactor	order-method-by-basestock.compute.phps	CAbstractOrderMethod::compute	28527	Michal	2015-04-07 14:33:00	0xECFD6DAF2E49721D86FE97F94C26325E33D3DF6C09…
2737	raven	order-method-normal.compute.phps	CAbstractOrderMethod::compute	2570	Michal	2015-05-11 09:25:53	0xECFDDD9224C7912508BFCA2EAE091733553355B3BE…

Obrázok 8.4: Záznam pahýľov v databázy [vlastné]

Pahýľ je programovo reprezentovaný triedou `Stub`, ktorá disponuje primárnymi operáciami pre prácu s informáciami nachádzajúcimi sa v databázy pod stĺpcom *data*. Ide o nastavenie, mazanie a získavanie dát, tieto operácie sú reprezentované metódami `setData`, `getData`, `unsetData`. Dôležité sú metódy na ukladanie a získavanie stavu tried `setClassState`, `getClassState`. Pre obsluhu databázových dát sú k dispozícii metódy `addTable` a `restoreDatabase`. Na prácu s dátami reprezentujúcimi výsledok algoritmu sú určené metódy `addResult` a `getResult`,

podobné sú operácie `addGraph` a `getGraphs` pre prácu s dátami na vyobrazenie grafu. Pahýľ je prispôsobený tak, aby mohol niesť viac ako jeden výsledok.

Trieda `StubManager` zaobaľuje triedu `Stub` a stará sa o načítanie a ukladanie pahýľov z databáze alebo zo súboru. Príkladom môže byť metóda starajúca sa o načítanie pahýľa z databáze:

```
public function loadStubFromDatabase($id)
{
   $result=$this->connection->query('SELECT [data] FROM [stubs] WHERE [id] = %i',$id);
   $data = $result->fetchSingle();
   $data = gzinflate($data);
   $data = json_decode($data, true);
   return $this->createStubFromArray($data);
}
```

Celá architektúra starajúca sa o prácu a reprezentáciu pahýľu je vyobrazená v prílohe B.

Integračné testy v Planning Wizard

Konvencia pomenovania a umiestnenia integračných testov je nasledujúca:

- testové súbory sú umiestnené v adresárovej zložke modulu, ktorý testujú, avšak ak je daný algoritmus špecificky preťažený je potreba test umiestniť do zložky k tomuto modulu,
- konvencia pomenovania je *<názov-modulu>*.***<názov-testovanej-metódy>.phps***.

Pre správne fungovanie integračných testov v Planning Wizard si musíme najprv uvedomiť tieto body:

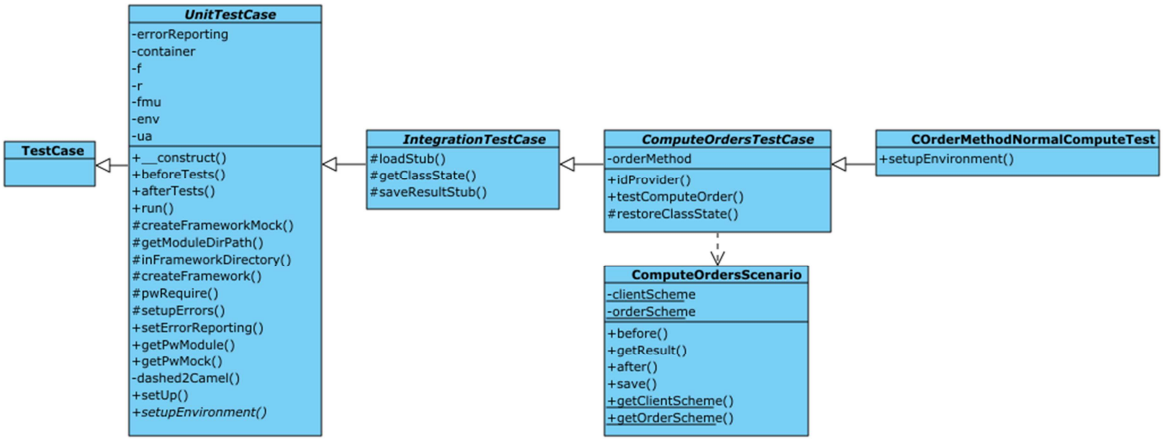
1. Pre každý projekt existujú špecifické preťaženia, ktoré môžu zmeniť správanie algoritmov.
2. Plannig Wizard potrebuje pre svoj beh nastavenie definujúce projekt v súbore s príponou *pwm*.

Ďalej si vysvetlíme integračný test určený pre otestovanie výpočtu objednávok v produkte Planning Wizard. Štruktúra testov je veľmi podobná jednotkovým testom, majú rovnaký základ. V Planning Wizard sa objednávky počítajú pomocou rôznych metód, ktoré závisia na rôznych vlastnostiach. Spôsob testovania je založený na porovnaní výsledku výpočtu objednávok v reálnom čase na aktuálnych dátach s výsledkom nachádzajúcom sa v pahýli, ktorý je k dispozícii na databázovom serveri.

Regresný test objednávkovej metódy *order-method-normal* sa nachádza v súbore *order-method-normal.compute.phps*. Trieda `COrderMethodNormalComputeTest` v tomto súbore obsahuje metódu na nastavenia prostredia `setupEnviroment`, ktorá je pre každý test špecifická. Čo v našej situácii znamená, že je pre každú metódu výpočtu objednávok rozdielna. Táto metóda sa primárne stará o vytvorenie potrebných falošných objektov, aby bolo možné vykonať test. Práca s falošnými objektmi v PwTesteri je vysvetlená na strane 42. Príklad metódy `setupEnviroment`:

```
public function setupEnvironment()
{
    $this->f = $this->createFrameworkMock();
    $this->fmu->shouldReturnPwModule($this->f, 'utils-datetime', true);
    $this->fmu->shouldReturnDefaultModuleMock($this->f);
}
```

V samotnej triede `COrderMethodNormalComputeTest` sa nenachádza testovacia metóda ale trieda dedí z triedy `ComputeOrdersTestCase`, ktorá predstavuje spoločnú nadtriedu pre testy objednávkových metód. Základná architektúra integračných testov je vyobrazená na nasledujúcom obrázku. Danú architektúru je potrebné dodržať pre správnu funkčnosť testov.



Obrázok 8.5: Diagram tried integračného testu [vlastné]

Nasledujúca ukážka metódy predstavuje základ celého integračného testu pre objednávkovú metódu *order-method-normal*:

```
/**
 * @dataProvider idProvider
 */
public function testComputeOrder($id)
{
    $ua = $this->ua;
    $_GET['loadstubmode'] = 'CAbstractOrderMethod::compute';
    $scenario = $this->container->getService('scriptwriter')-
>createComputeOrdersScenario($this->orderMethod);
    $stub = $this->loadStub($id);
    $stub->restorePwDate();
    $state = $this->getClassState($stub, $this->orderMethod);
    $this->restoreClassState($this->orderMethod, $state);

    $expectedResults = $stub->getResults();

    $this->orderMethod->setUserMode(true);
    $this->orderMethod->compute();

    $actual = $scenario->getResult();
    $resultStub = new Stub();
    $resultStub->addResult($actual);

    $hasGood = false;
    $exceptions = array();
    foreach($expectedResults as $expected)
    {
       try {
          Assert::count(count($expected), $actual);
```

```
            foreach($expected as $exp)
            {
                $act = current($actual);
                if(isset($exp['orders']))
                {
                   $act = $act['orders'];
                   $exp = $exp['orders'];
                   Assert::count(count($exp), $act);
                   foreach ($exp as $e)
                   {
                       $a = current($act);
                       $a = $ua->copyArray($a, ComputeOrdersScenario::getOrderScheme());
                       $ua->assertSimilar($e, $a);
                       next($act);
                   }
                }
                else
                {
                   Assert::null($exp);
                }
                next($actual);
            }
            $hasGood = true;
            break;
        }
        catch(\Exception $e)
        {
            $exceptions[] = $e;
        }
    }
    if(!$hasGood)
    {
        $resultStub->setData('exceptions', array_map(function ($e) {
        return array('type'=> get_class($e),'message'=>$e->getMessage(),'trace'=>$e-
>getTraceAsString());
        }, $exceptions));
        $resultStub = $scenario->after($resultStub);
        $this->saveResultStub($resultStub, $id);
        throw $exceptions[0];
    }
}
```

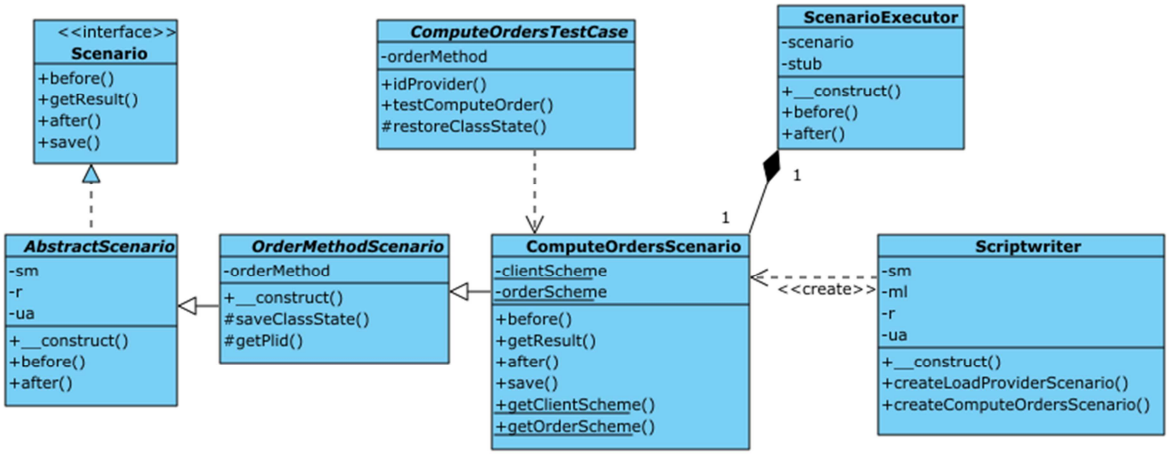
Metóda je spúšťaná na základe poskytovateľa dát, ako u jednotkových testov. Rozdiel je v tom, že vstupné dáta sa nachádzajú v súbore vo formáte *ini*[17], ktorý obsahuje identifikačné čísla pahýľov v databázy a tento súbor vzniká na základe vybraných testov cez grafické rozhranie. Ukážka dát vo formáte *ini*:

```
[2321]
id=2321
```

[17] Viac informácií na URL: https://msdn.microsoft.com/en-us/library/windows/desktop/ms717987%28v=vs.85%29.aspx

Vykonanie testu prebieha nasledujúcim spôsobom:

1. Pomocou triedy `ScriptWriter`, ktorá predstavuje návrhový vzor továrne si vytvoríme inštanciu príslušného scenára pre danú metódu výpočtu objednávok. Čo v rámci ukážky znamená scenár `ComputeOrderScenario`. Jeho úlohou je zabezpečiť vykonanie úloh pred a po teste, ako je získanie výsledku výpočtu objednávok či uchovanie stavu tried. Daný scenár je riadený triedou `ScenarioExecutor`. Nižšie je vyobrazený diagram tried reprezentujúci architektúru scenára.



Obrázok 8.6: Diagram tried reprezentujúci scenár pre test objednávok (bez závislosti na vedľajších triedach) [vlastné]

2. Dôjde k nahraniu pahýľa z databázového serveru.
3. Na základe údajov z pahýľa sa nastavia vstupné údaje potrebné pre spustenie výpočtu objednávok.
4. Následné sa spustí výpočet objednávok a získa sa aj výsledok objednávok z pahýľa, ktorý sa považuje za správny.
5. Pomocou asercie sa vykoná porovnanie výsledku z reálne výpočtu objednávok zo všetkými dostupnými výsledkami v pahýli.
6. V prípade, že sa dané výsledky nerovnajú, vytvorí sa súbor s názvom *<id pahýľa>.res*, obsahujúci výpočet objednávok, ktorý sa neskôr využije pri zobrazení výsledku testu.

Grafické rozhranie integračných testov

Grafické rozhranie je založené na aplikačnom rámci Bootstrap a obsahuje tri stránky. Základná stránka je rozdelená na dve časti. Prvá časť slúži pre výber testov a projektov. Obrázok na nasledujúcej strane ukazuje, že je možné vybrať viac ako jeden projekt. Ak je projekt zobrazený červenou farbou symbolizuje to, že nemáme k dispozícii zdrojové súbory pre daný projekt.

Obrázok 8.7: Grafické rozhranie pre spustenie integračných testov [vlastné]

Druhá časť obrazovky prezentuje výsledok testov. Ak je vybraných viac projektov tak pre každý sa vytvorí tabuľka aká je na obrázku nižšie. Dostupný je aj základný konzolový výstup a to cez odkaz s názvom Log. Ak nastane chyba v teste, sprístupní sa odkaz na ďalšiu stránku cez názov testu.

Obrázok 8.8: Tabuľka reprezentujúca výsledok testov [vlastné]

Programová reprezentácia stránky sa nachádza v súboroch *TestRunner.php*, ktorý obsahuje logiku a *TestRunnerGUI.php* disponujúci grafikou. Čo sa týka grafiky sú použité rovnaké elementy a techniky ako pri jednotkových testoch. Rozdiel je v logike. Je potreba pristupovať k databázovému serveru pre získanie informácií o existujúcich pahýľov pre dané projekty. Taktiež je nutné zistiť informácie o dostupných zdrojových súboroch pre rôzne projekty.

```
public function getSpecificProjects()
{
    $pwRoot = $this->env->getPWRoot();
    $specificProjects = explode("\n", shell_exec("cd $pwRoot ; ls -F ./specific/source
| grep / | cut -f1 -d'/' | grep -v _ "));
    array_pop($specificProjects);
    return $specificProjects;
}
```

Všetky tieto informácie sú použité pre spustenie testov. Ako už bolo spomínané v produkte Plannig Wizard sa nachádzajú špecifické preťaženia pre jednotlivé projekty, preto je potreba si tieto prípady ošetriť, aby dochádzalo k správnemu priebehu testov.

O spustenie testov sa stará metóda `runStubTest`, ktorú si bližšie popíšeme. Pre každý vybraný projekt sa spustí cyklus nad označenými testami. Aby sa zabezpečila jednoznačnosť jednotlivých behov testov, vytvorí sa v zložke *cache/tester* ďalšia zložka. Táto zložka je pomenovaná podľa univerzálneho identifikátoru (uuid), ktorý získame pomocou nasledujúceho kódu `$uuid = shell_exec('cat /proc/sys/kernel/random/uuid | tr -d \'\r\n\'');`. Do tejto zložky sa ukladajú výstupy testov, spolu so súbormi potrebnými pre beh testu (samotný testovací súbor, súbor s identifikátormi pahýľov určenými pre spustenie).

Potom sa vykoná cyklus, ktorý pre každý testovací súbor zistí, či neexistuje jeho špecifické preťaženie, či sú pre test a projekt prístupné pahýle a na záver skopíruje súbory s testom do zložky v *cache/tester/<uuid>*. Súbory sa skopírujú z dôvodu, aby jeden test mohol bežať paralelne viac krát s rôznymi dátami.

Pre prípadne potreby testov sa taktiež vytvorí súbor *default.pwm*, ktorý definuje základne nastavenie pre produkt Planning Wizard. Spustenie testov je cez nasledujúci príkaz:

```
$cmd = "PW_TEST_UUID=$uuid PW_ROOT=$pwRoot php $pwRoot/composer/vendor/bin/tester -j 8
-c $pwRoot/tools/tester/php.ini --log $pwRoot/cache/tester/$uuid/log
".$output.$testFilePaths;
$out = shell_exec($cmd);
```

Po vykonaní testov je výstup prekontrolovaný a sú vytvorené súbory s informáciami pre reprezentáciu chýb. Pre potreby priebežnej integrácie sú dostupné metódy, ktoré vytvoria výstup vo formáte TAP[18] a HTML.

Obrázok nižšie ukazuje tabuľku zobrazujúcu prvotné informácie o chybách spolu zo základnými informáciami (názov testu, identifikátor pahýľa a vstupného objektu).

Test: order-method-normal.compute | **Functions:**

Stub ID	Identifier	Exception	
2628	58	Count 55 should be 301	☑

Obrázok 8.9: Tabuľka zobrazuje informácie o chybách v danom teste [vlastné]

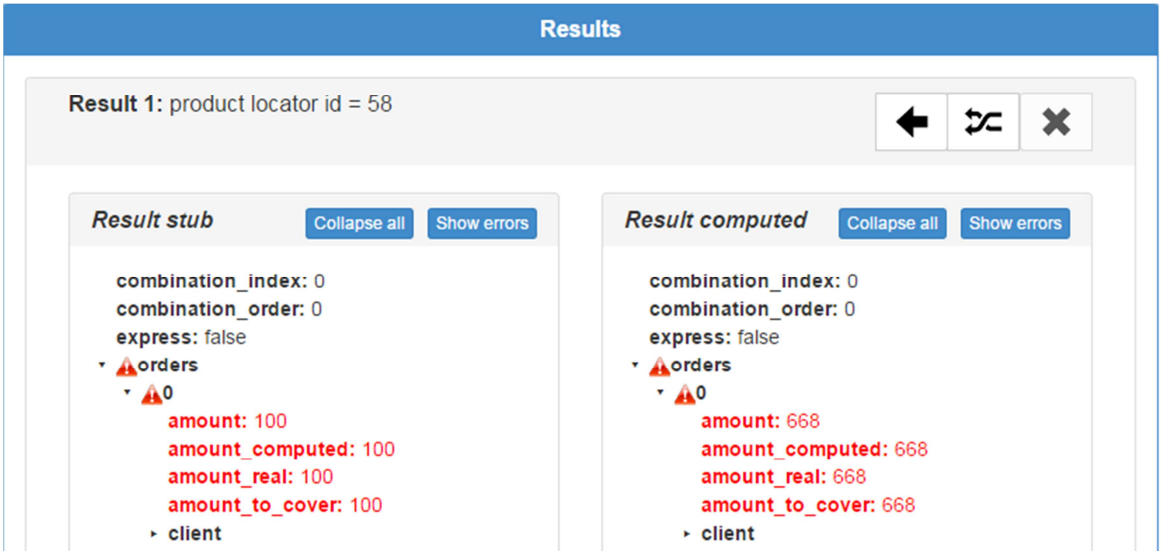
[18] Viac informácií na URL: https://testanything.org/tap-version-13-specification.html

Grafické zloženie tabuľky sa prispôsobuje na základe typu chyby. Rozlišujeme chybu typu `Tester\\AssertException`, ktorá prezentuje nezhodu dát vo výpočte objednávok a ostatné chyby, ako sú napríklad chyby v zdrojových kódoch. Cez stĺpec Exception je prístupný odkaz zobrazujúci podrobnejšie informácie o chybe. Ak ide o nešpecifickú chybu dostaneme výstup – príloha C Obrázok C.2. U chyby `Tester\\AssertException` je možné zobraziť graf reprezentujúci vypočítane dáta a dáta z pahýľa – príloha C Obrázok C.3. V pravom rohu tabuľky si môžeme všimnúť stĺpec s názvom Functions. Tento stĺpec obsahuje funkciu na vytvorenie nového pahýľa na základe nových dát:

```
public function recreateStubs($ids, $uuid)
{
    $stubDatabase = $this->sm->loadColumnsFromDatabase($ids, array('*'));
    foreach ($stubDatabase as $value)
    {
       $this->sm->recreateStub($value, $uuid);
    }
    $insert =  $this->transpose($stubDatabase);
    unset($insert['id']);
    $this->sm->insertRecordToDatabase($insert);
}
```

Programová reprezentácia tejto stránky je v súboroch *ErrroTable.php* a *ErrorTableGUI.php*.

Ďalšia stránka je prístupná cez odkaz v stĺpci Stub ID. Táto stránka prezentuje dáta a špecifikuje chyby v nich. Stránka sa prispôsobuje na základe typu testu. Pri testoch objednávok sa zobrazí graf a strom reprezentujúci dáta objednávky.



Obrázok 8.10: Stromová reprezentácia chýb v dátach výpočtu objednávok [vlastné]

Stromovú štruktúru dát je možné ovládať zrkadlovo a zobraziť si len chybné alebo všetky dáta. Nad celým pahýľom sú k dispozícii tri operácie prístupné v pravom hornom rohu rozhrania. Prvou funkciou je nahratie vypočítaného pahýľu na databázový server, ktorý bude reprezentovať nový

správny výsledok. Tiež môžeme pridať do pahýľa nový výsledok a ak pahýľ obsahuje viac výsledkov je možné výsledky odstrániť ale vždy musí ostať aspoň jeden.

Grafickú kostru stránky obsahuje súbor *ErrorCompareGUI.php*. Nasledujúci kód na základe testovacieho súboru a nastavenia parametrov v konfiguračnom súbore *config.neon* rozhodne aké správy (angl. reports) sa zobrazia:

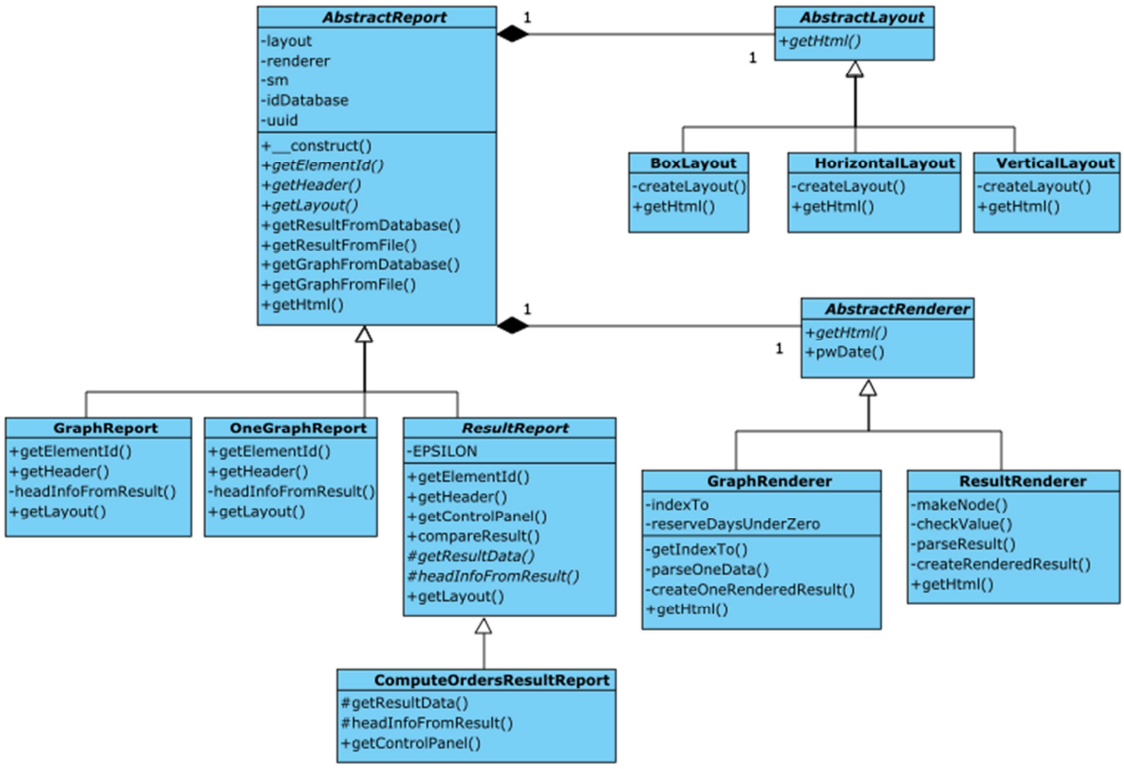
```
$test = $_GET['test'];
$reports = $container->parameters['reports'][$test];
$services= array();

foreach ($reports as $value)
{
    $services[] = $container->getService($value);
}
```

Ukážka nastavenia z *config.neon*, vyčleňuje aké správy sa majú zobraziť pre objednávkovú metódu *order-method-normal*:

```
reports:
  order-method-normal.compute:
    - computeOrdersResultReport
    - oneGraphReport
```

Štruktúra správ je vytvorená tak, aby bolo jednoduché vytvárať ďalšie. Správa je rozdelená na usporiadanie (angl. layout) a renderovanie (angl. rendering). Usporiadanie sa stará o rozloženie boxov, ktoré nesú dáta. Momentálne sú k dispozícií rozloženie horizontálne, vertikálne a schránka. Renderovanie slúži na upravenie predaných dát a ich zobrazenie v požadovanom formáte. Dáta zobrazujúce objednávky je možné zobraziť vo forme grafu alebo stromovej štruktúry. U stromovej štruktúry dochádza v abstraktnej triede `ResultReport` k porovnaniu výsledkov pomocou metódy `compareResult($expected, $actual, $key = NULL,$re = array()).` Metóda vráti pole nerovnajúcich sa dát, ktoré sú využité pri renderovaní chýb.



Obrázok 8.11: Diagram tried reprezentujúci grafické zobrazenie objednávkových dát [vlastné]

8.2 Implementácia priebežnej integrácie

Cieľom implementácie priebežnej integrácie je automatizácia testovania a kontrola prepočtu produktu Plannig Wizard na vzorke reálnych dát. Pre implementáciu priebežnej integrácie bol zvolený voľne dostupný nástroj Jenkins, ktorý je vhodný pre projekty implementované v jazyku PHP. Viac informácií o tomto produkte sa nachádza v kapitole 3.1.

8.2.1 Štruktúra Jenkinsu

Popíšme si najprv štruktúru Jenkinsu. Jenkins sa skladá z úloh, ktoré môžeme chápať ako projekt. Aktuálne sú k dispozícii štyri typy úloh z ktorých sa dá vybrať:

1. **Základný projekt** – ide o najčastejší typ úloh určených pre jednoduchšie projekty a akcie k vykonaniu.
2. **Maven projekt** – ide o projekty založené na nástroji Maven[19].
3. **Externá úloha** – táto úloha slúži na prezentovanie údajov z iných systémov automatizácie.
4. **Mnohonásobne nastaviteľný projekt** – vhodné pre projekty, kde je potreba veľkého množstva konfigurácií, napríklad testovanie v rôznych prostrediach.

[19] Viac informácií na URL: https://maven.apache.org/

V rámci týchto úloh sa vykonávajú jednotlivé zostavenia (angl. build). Pod zostavením si môžeme predstaviť jednotlivé operácie, ktoré chceme aby sa vykonali pre dané zdrojové súbory ako napríklad testy, statická analýza, atď.

Pre vytvorenie základného projektu je potrebné prejsť nasledujúcimi **piatimi krokmi nastavenia**:

1. Nastavenie názvu, popisu projektu, spolu zo zadefinovaním údaju o uchovávaní informácií o zostaveniach.
2. Nastavenie repozitára zdrojových súborov.
3. Nastavenie akým spôsobom sa budú spúšťať zostavenia.
4. Ďalej nastaviť samotné akcie zostavenia, napríklad spustenie testov.
5. Na záver sa definujú akcie, ktoré sa majú realizovať po zostavení, napríklad vyhodnotenie testov.

8.2.2 Architektúra Jenkinsu v spoločnosti Logio

Architektúra Jenkinsu je implementovaná podľa návrhu z kapitoly 7.2. Jenkins je nainštalovaný ako služba na serveri s operačným systémom CentOS 7. Pre beh Jenkinsu je potrebné mať nainštalovanú Javu. Inštalácia je jednoduchá a je možná pomocou balíčkového systému *yum*[20]. Po inštalácii nám tento server slúži ako správca, zobrazovateľ informácií o úlohách a je prístupný na porte 8080.

Dané zostavenia sa vykonávajú na otrokovi. Otrok predstavuje samostatný server, taktiež s operačným systémom CentOS 7 a je riadený správcovským serverom. Takéto rozčlenenie zabezpečuje efektívne využitie výpočtovej sily a v prípade potreby je možné pridať ľubovoľný počet ďalších otrokov. Prepojenie medzi správcom a otrokom zabezpečuje správca a je potrebné len nastaviť prístupy k otrokovi, a musí byť na ňom k dispozícii súbor *slave.jar*[21]. Na otrokovi musíme mať nainštalované nástroje, ktoré sa budú používať pri zostavení ako napríklad Selenium server.

Na otrokoch sa o zostavenia starajú exekútori. Ich počet sa nastavuje v závislosti na výkone daného otroka. Čím viac exekútorov tým je možné vykonávať viac zostavení súčasne.

V rámci diplomovej práce bol implementovaný jeden otrok so štyrmi exekútormi. A na základe potrieb spoločnosti Logio boli vytvorené dva typy úloh zabezpečujúce pravidlá priebežnej integrácie, ktoré si podrobnejšie popíšeme v nasledujúcich dvoch kapitolách.

8.2.3 PW-Commit

PW-Commit je úloha typu **základný projekt**. Tým, že je produkt Planning Wizard rozsiahli je potrebné pridržiavať sa pravidiel priebežnej integrácie. Z toho dôvodu sa táto úloha považuje za hlavné (primárne) zostavenie, ktoré by sa malo pohybovať v časovom rozmedzí desať minút. Na základe časového obmedzenia bolo preto potrebné si nadefinovať tie najdôležitejšie akcie zostavenia a ostané akcie prenechať do vedľajšieho (sekundárneho) zostavenia.

[20] Ide o voľne dostupný nástroj, slúžiaci k správe balíčkov linuxových systémov.

[21] Viac informácií na URL: https://wiki.jenkins-ci.org/display/JENKINS/Step+by+step+guide+to+set+up+master+and+slave+machines

Nastavenie úlohy na základe piatich bodov z predchádzajúcej strany je nasledujúce:

1. Zostavenia sa uchovávajú z dvoch posledných dní.
2. Ako repozitár zdrojových kódov je použitý nástroj SVN Subversion. Do úvahy sa berú aj externé zdrojové súbory.
3. Jenkins v pätnásť minútových intervaloch kontroluje repozitár a na základe zistenia zmeny v zdrojových kódoch spúšťa zostavenie.
4. Pri zostavení sa vykoná spustenie jednotkových testov a statickej analýzy.
5. Na záver sú spustené nástroje vyhodnocujúce testy a statickú analýzu.

Zdrojové súbory a nastavenia pre beh tejto úlohy sa nachádzajú v testovacom rámci PwTester v zložke *jenkins*.

Zloženie zostavenia

Pre spúšťanie akcií zostavenia sa využíva nástroj Ant od spoločnosti Apache. Ant dokáže automatizovať radu činností a definuje jednotný zápis akcií, ktoré sa majú vykonať. Skripty sa píšu v jazyku XML. V rámci práce sú vytvorené rozkúskované skripty, ktoré obsahujú spúšťanie samostatných akcií, ako je napríklad nastavenie prostredia alebo spustenie jednotkových testov. Tento koncept uľahčuje tvorbu nových úloh v Jenkinse, kde stačí poskladať potrebné akcie a nie je potreba upravovať skripty. Ukážku skriptu, ktorý slúži pre počiatočne nastavenie prostredia pred zostavením sa nachádza v prílohe D.

Tento skript pred každým spustením vyčistí prostredie, vytvorí adresárovú architektúru pre uskladnenie výstupov spustených nástrojov a mal by byť základom každej úlohy v Jenkinse. Taktiež spustí shellový skript *jenkins-setup.sh*, ktorý obnoví knižnice pomocou nástroju Composer a inicializuje sa prostredie pre beh produktu Planning Wizard.

Po inicializácii prostredia sa môžu vykonať jednotkové testy, ktorých výsledok definuje stav zostavenia. Pre spustenie testov slúži skript *build-unit.xml*, ktorý inicializuje PHP skript *run-unit-tests.php*:

```
use Kdyby\Curl;
require_once(__DIR__.'/../../../composer/vendor/autoload.php');

preg_match("/\/var\/www\/html\/(.*)jenkins/", __DIR__, $output);
$server = new Curl\Request("http://localhost/$output[1]/api?runUnitTests");
$server->setTimeout(10000000);

try
{
    $response =  $server->post(array('paths'=> array('shared', 'modules', 'specific'),
'output' => 'tap'));
    echo $response->getResponse();
}
catch(Curl\CurlException $e)
{
    echo $e->getMessage();
}
```

Tento skript pomocou HTTP požiadavku uvedie do chodu PwTester pre jednotkové testy. PwTester automatický spustí všetky jednotkové testy, na ktoré narazí v zadefinovaných zložkách a vytvorí

výstup vo formáte TAP. Ak dôjde k chybe u jednotkových testov zostavenie je ukončené a vyhodnotené ako neúspešné.

Po úspešných testoch je na rade statická analýza. O jej spustenie sa stará skript *build-static_analyze.xml*. Pre rýchlejšie vykonanie analýzy je vhodné spustiť nástroje paralelne to môžeme dosiahnuť pomocou nasledujúceho nastavenia v skripte:

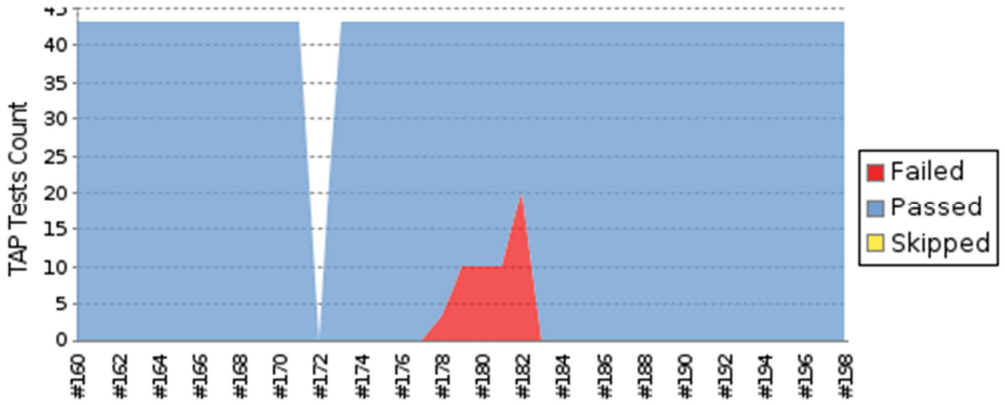
```
<target name="tools-parallel" description="Spusti nastroje paralelne.">
<echo message="Build part static analyze"/>
<parallel threadCount="4">
    <antcall target="phpcpd-ci"/>
    <antcall target="phpcs-ci"/>
    <antcall target="phploc-ci"/>
</parallel>
</target>
```

Statická analýza zdrojových súborov produktu Plannig Wizard je založená na týchto nástrojoch:

- **php** – pomocou parametru -l (lint) je možné vykonať syntaktickú kontrolu zdrojových súborov. Pri syntaktickej chybe dôjde k ukončeniu zostavenia.
- **PHPLOC** – meria veľkosť a analyzuje štruktúru PHP projektu.
- **PHP Copy/Paste Detector (PHPCPD)** – kontroluje duplicitné riadky v projekte.
- **PHP_CodeSniffer** – na základe zadefinovaných kódovacích noriem vie vyhľadať a opraviť prípadne narušenia pravidiel. Sadu pravidiel je potrebné zadefinovať do súboru vo formáte XML. Pre produkt Planning Wizard sú pravidlá zadefinované v súbore *pw_ruleset.xml*. Nastavené je pravidlo pre kontrolu zakázaných a zastaraných funkcií v PHP, kontrola predávania parametrov do funkcie cez referenciu a kontrola pravidiel pre použitie premenných.
- **PHP_CodeBrowser** – generuje prezerateľnú reprezentáciu PHP kódu spolu so zvýraznením chýb získaných z nástrojov PHPCPD a PHP_CodeSniffer.

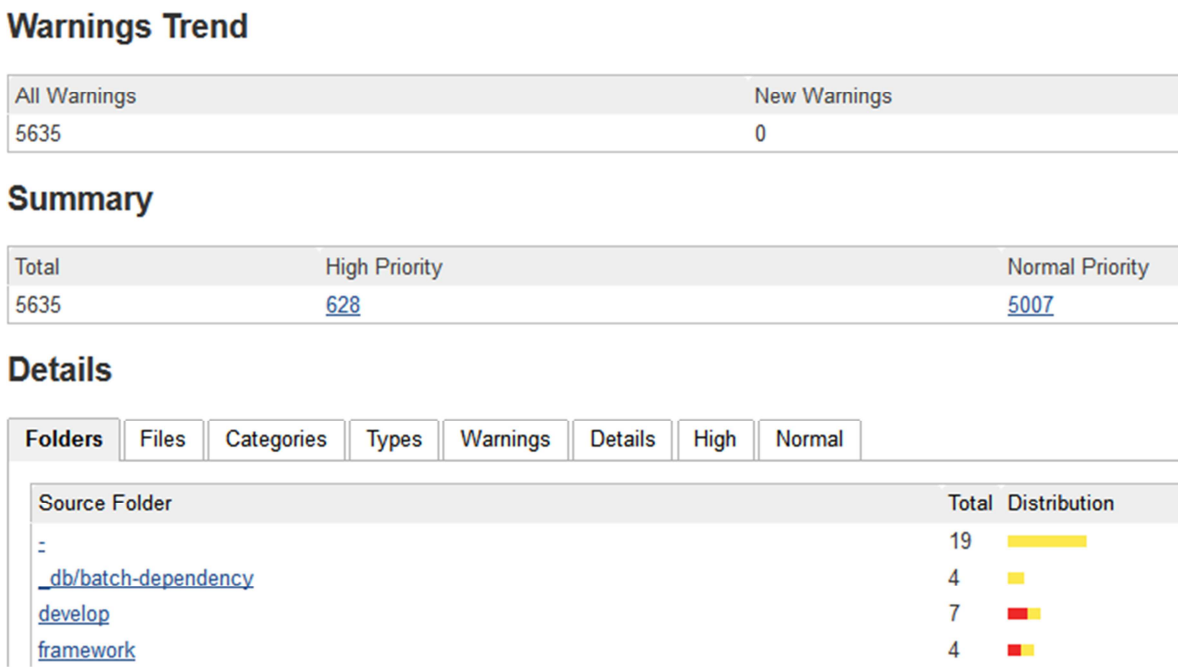
Výsledky nástrojov sú následne uspôsobené v poslednej etape zostavenia a to prevažne pomocou zásuvných modulov dostupných pre Jenkins. Tieto moduly sú schopné sformátovať výstupy nástrojov a zobraziť ich vo forme pochopiteľnej pre človeka.

Na výstup jednotkových testov je použitý modul **Publish TAP Result** pre spracovanie súborov vo formáte TAP, ktorý vytvorí výstup vo forme grafu a textu.



Obrázok 8.12: Graf zobrazujúci výsledky jednotkových testov z posledných 38 zostavení [vlastné]

Na zobrazenie výstupov z nástrojov PHPCPD a PHP_CodeSniffer sa používajú moduly **Publish Checkstyle analysis results** a **Publish duplicate code analysis results**, ktoré pracujú na rovnakom princípe. Zobrazujú štatistiky chýb v grafe a jednotlivé chyby je možné prezerať priamo v zdrojových súboroch.



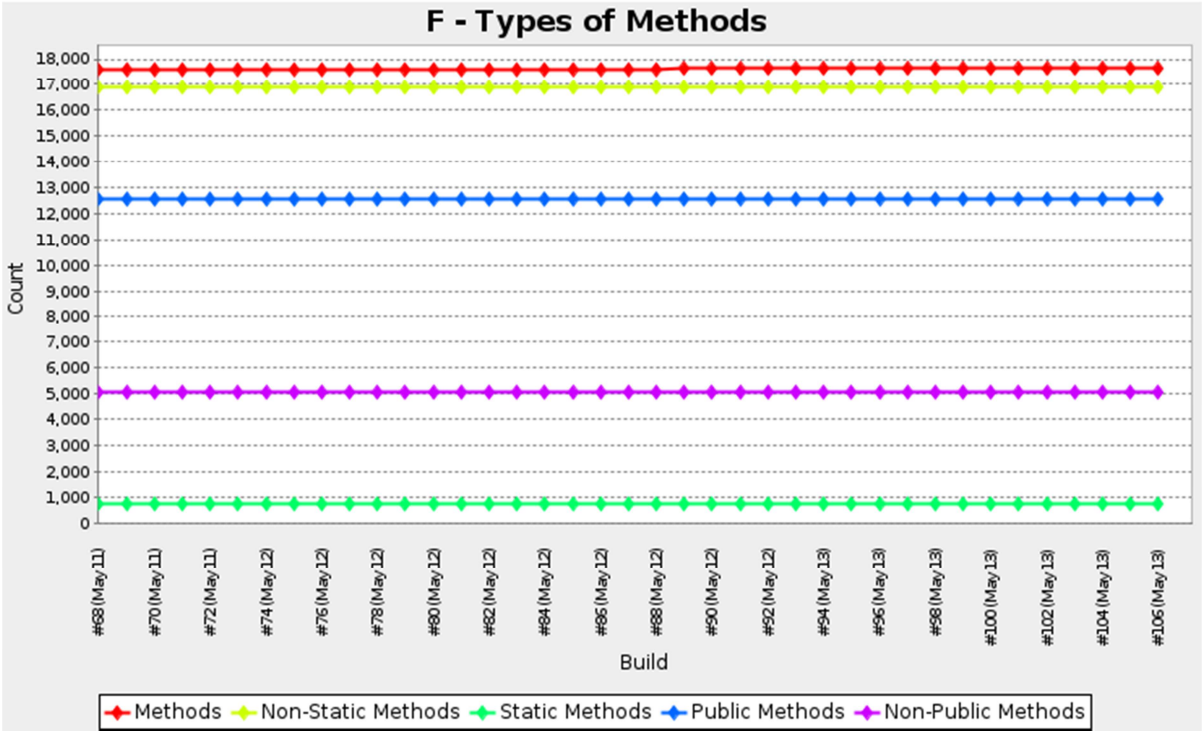
Obrázok 8.13: Grafické rozhranie modulu Checkstyle analysis results [vlastné]

Rovnakým rozhraním disponuje modul **Scan workscpace for open tasks** určený pre vyhľadanie otvorených úloh v projekte. Modulu stačí zadefinovať typy súborov, v ktorých hľadať a značky reprezentujúce otvorenú úlohu. Najčastejšie ide o značky vo formáte FIXME a TODO, ktoré sa líšia svojou prioritou.

Veľmi užitočným zásuvným modulom je **Publish HTML reports**, ktorý dokáže prezentovať HTML správy priamo v rozhraní Jenkinsu. Postačuje nastaviť cestu a názov k index súboru spolu

s názvom výslednej správy. Tento modul sa používa na zobrazenie výstupu z nástrojov PHPLOC a PHP_CodeBrowser.

Na základe výstupov nástroja PHPLOC pri každom zostavení sme schopný pomocou grafov sledovať aj štatistiky ako sú počty okomentovaných a neokomentovaných riadkov kódu, priemerný počet riadkov metódy a mnoho ďalších.



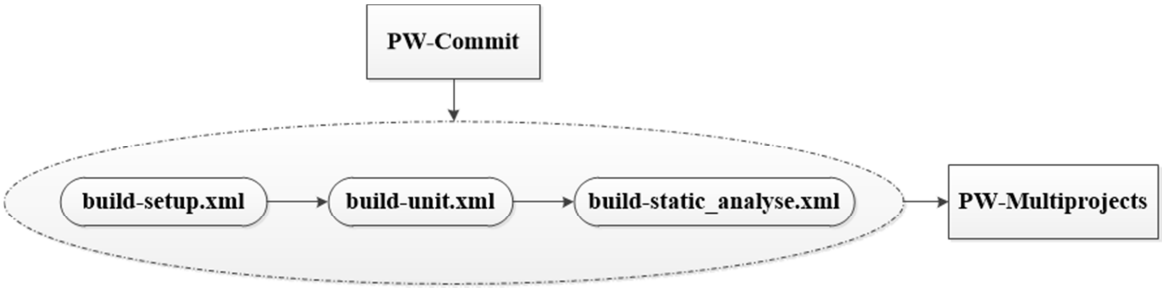
Obrázok 8.14: Graf zobrazujúci typy metód na základe výstupu z nástroja PHPLOC [vlastné]

Pre kontrolu správneho vykonania zostavenie používame ešte modul **Scan for compiler warnings**, ktorý kontroluje prípadne chyby v behu PHP.

V určitých prípadoch je konzolový výstup zostavenia príliš neprehľadný a na lepšie orientovanie sa v ňom je použitý modul **Console output parsing**, ktorý na základe vopred predefinovaných pravidiel vie rozčleniť výstup. Pravidlá sú zadané pomocou regulárnych výrazov, ktoré stanovujú návesti pre navigovanie vo výstupe.

Pre upozornenie užívateľov o výsledku zostavenia je najvhodnejší email. Preto je použitý modul **Editable Email Notification** umožňujúci definovať formát, obsah a pravidlá pre odoslanie emailu.

Záverečnou akciou úlohy PW-Commit je pri úspešnom zostavení spustenie druhej vedľajšej, doplňujúcej úlohy PW-Multiprojects s požadovanými parametrami. Týmto medzi nimi vzniká závislosť.



Obrázok 8.15: Závislosť úlohy PW-Multiprojects na PW-Commit [vlastné]

8.2.4 PW-Multiprojects

PW-Multiprojects je úloha typu **mnohonásobne nastaviteľný projekt** s parametrom. Tento typ úlohy bol zvolený pre schopnosť vykonať v rámci jedného zostavenia viac dielčích zostavení s rovnakým alebo rôznym nastavením.

Samotná úloha je spustená na základe úspešného vyhodnotenia úlohy PW-Commit a ide o pokračujúce (sekundárne) časovo náročné zostavenie. Ako parameter je zadaný reťazec vo formáte `projects=raven abb ges`, kde `raven`, `abb` a `ges` predstavujú projekty, ktoré sa majú zostaviť. Účelom úlohy je otestovať základnú funkčnosť produktu Planning Wizard na reálnych dátach.

Nastavenie úlohy na základe piatich bodov zo strany 56 je rozšírené ešte o dve nastavenia závislé na danom type úlohy:

1. Zostavenia sa uchovávajú z dvoch posledných dní.
2. Potreba nastaviť parameter pre zostavenie. Ide o textový parameter s názvom `projects`.
3. Ako repozitár zdrojových kódov je použitý nástroj SVN Subversion. Do úvahy sa berú aj externé zdrojové súbory.
4. Inicializácia úlohy je na základe úspešného vykonania PW-Commit.
5. Nastavenie matice úlohy.
6. Pri zostavení sa spustí prepočet produktu Planning Wizard, spustia sa integračné testy a testy grafického rozhrania.
7. Na záver sú spustené nástroje vyhodnocujúce testy a prepočet systému.

Matica úlohy

Mnohonásobne nastaviteľný projekt umožňuje definovať aké akcie sa majú duplikovať a vytvoriť viac osí graf zostavení pre spustenie.

Úloha PW-Multiprojects nastavuje pomocou parametru `projects` dynamickú os s názvom `project`. Dynamická znamená, že je nastavená cez premennú prostredia. To nám povoľuje absolvovať paralelne zostavenia pre všetky projekty v parametri na rovnakom slede akcií.

Zdrojové súbory a nastavenia pre beh tejto úlohy sa nachádzajú v testovacom rámci PwTester v zložke *jenkins*.

Zloženie zostavenia

Na začiatku zostavenia je rovnako ako u úlohy PW-Commit spustení skriptu *build-setup.xml* nastavujúci prostredie. Po nastavení prostredia nasleduje spustenie skriptu *build-batch.xml*, ktorý zavolá PHP skript *jenkins-batch.php*, kontrolujúci funkčnosť produktu Planning Wizard.

Pre správne vyhodnotenie funkčnosti je potrebné overiť dve operácie. Prvou operáciou je vykonanie rozdielových databázových skriptov. O túto operáciu sa stará metóda `setUpEnviroment`:

```
private function setUpEnvironment()
{
    echo exec("chmod -R 777 cache/");
    echo exec("scp -i {$this->keyFile} -P port ipadresa.{$this->project}.sql.gz
/tmp/{$this->project}.sql.gz ");
    $this->createDb();
    dibi::connect(array(
       'driver'   => 'mysql',
       'host'     => 'localhost',
       'username' => 'root',
       'password' => 'heslo',
       'database' => "$this->projectDatabase",
       'charset'  => 'utf8',
    ));
    $this->createPwm();
    $log = file_get_contents("updateAutomaticallyLog&batch");
    $logRows = explode("<br>", $log);
    $this->createLog("build/updater/updater.html",$log);
    if($logRows[0]=="Error")
    {
       echo "UPDATER FAILED!\n";
       exit(1);
    }
    $this->setUpBatchSelection();
}
```

Metóda nastaví prostredie, aby mohla byť vykonaná kontrola funkčnosti. To zahŕňa získanie skriptu pre naplnenie databáz z centrálneho databázového serveru, vytvorenie pripojenia k primárnej databáze a vytvoriť súbor s nastavením pre Planning Wizard. Po nastavení prostredia je možné spustiť nástroj updater, ktorý je súčasťou produktu a zabezpečuje vykonanie databázových skriptov. Na základe výstupu updateru rozhodneme o správnosti operácie. Ak došlo k chybe, zostavenie sa ukončuje ako neúspešné.

Druhou operáciou kontroly je spustenie prepočtu produktu. Spustenie prepočtu zabezpečuje metóda `runBatch`:

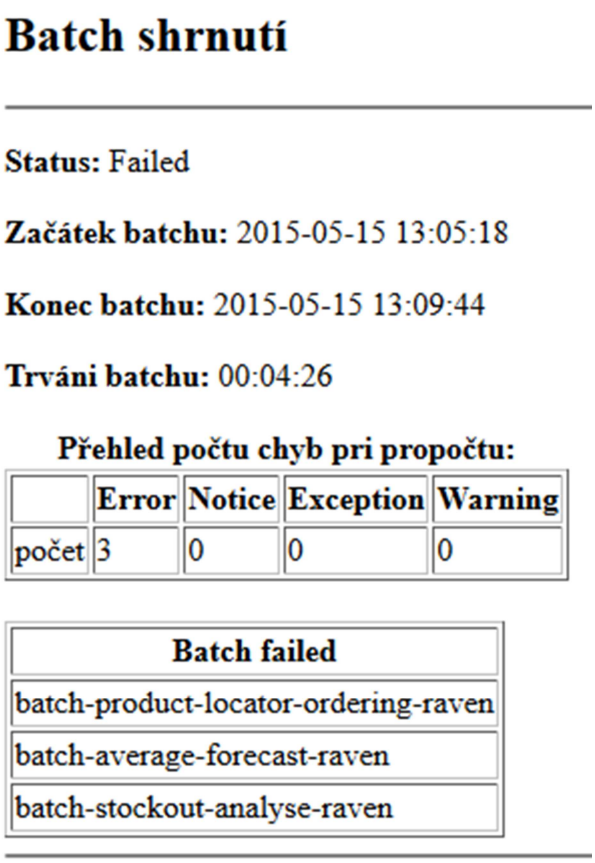
```
public function runBatch()
{
    file_get_contents("=runBatchFromCron&batch&parallel");

    while($this->isBatchRunning())
    {
       sleep(10);
    }
    $batchTimes = $this->getBatchTime();
```

```
    $failedBatch = $this->checkedBatch();
    $this->createBatchReport($batchTimes,$failedBatch);

    if(count($failedBatch) > 0)
    {
       echo "BATCH FAILED!\n";
       exit(1);
    }
}
```

Metóda spustí prepočet na vopred zadefinovaných moduloch systému a v desať sekundových intervaloch kontroluje, či sa prepočet stále vykonáva. Po ukončení prepočtu sa zo záznamov z databázy získajú údaje o časoch prepočtu a zoznam modulov, kde sa vyskytli chyby. Tieto údaje sú použité na vytvorenie správy o zhrnutí prepočtu.



Batch shrnutí

Status: Failed

Začátek batchu: 2015-05-15 13:05:18

Konec batchu: 2015-05-15 13:09:44

Trváni batchu: 00:04:26

Přehled počtu chyb pri propočtu:

	Error	Notice	Exception	Warning
počet	3	0	0	0

Batch failed
batch-product-locator-ordering-raven
batch-average-forecast-raven
batch-stockout-analyse-raven

Obrázok 8.16: Ukážka správy o chybnom prepočte [vlastné]

K tomuto zhrnutí sa pripája celkový výstup získaný z prepočtu systému vo formáte HTML. O zobrazenie výstupov z operácií sa stará modul **Publish HTML reports.** Ak dôjde počas prepočtu k chybe ukončuje sa zostavenie a vyhodnotí sa ako neúspešné.

Pre lepšie identifikovanie dôvodu ukončenia sa do konzolového výstupu vypisuje informácia o pôvodcovi chyby. Tieto predefinované výstupy sa odchytávajú modulom **Failure Cause**

Management. Modul filtruje konzolový výstup a pomocou regulárnych výrazov odchytí správy, ktoré zobrazí v zhrnutí o zostavení.

Po úspešnom vykonaní prepočtu sú spustené integračné testy – *build-stub.xml*. Inicializovaný skript *run-stub-test.php* pomocou HTTP požiadavku zavolá PwTester. Testovací rámec spustí testy pre všetky dostupné pahýle v centrálnej databáze v prípade, že sú k dispozícií špecifické zdrojové súbory projektov. Neúspešné testy neukončujú zostavenie len ho identifikujú ako nestabilné.

Potom sú spustené testy grafického rozhrania (selenium testy) pomocou skriptu *build-selenium.xml*:

```
<target name="phpunit-ci" depends="">
  <exec executable="${toolsdir}phpunit" failonerror="false">
    <arg value="--configuration"/>
    <arg path="${path}/tools/tester/jenkins/settings/phpunit.xml"/>
  </exec>
</target>
```

Na spustenie testov je použitý aplikačný rámec PHPUnit s nastavením definovaným v súbore *phpunit.xml*. Ten určuje umiestnenie výstupu testov a testovaciu súpravu, teda zložku kde sa nachádzajú testy spolu s príponou testovacích súborov. Viac informácií o selenium testoch sa nachádza v kapitole 8.2.5. Rovnako ako pri integračných testoch ani selenium testy pri neúspechu neukončujú zostavenie.

Výsledky všetkých druhov testov vyhodnocuje modul **Publish TAP Result.** Na získanie podrobnejších informácií o výsledkoch testov grafického rozhranie je použitý modul **Publish xUnit test result report**. Konkrétne je modul nastavený na spracovanie výstupu pre PHPUnit testy.

Integračné testy generujú svoje vlastné rozhranie pre zobrazenie výsledkov spísané v kapitole 8.1.2 a je prístupné pomocou modulu **Publish HTML reports.**

Rovnako ako pri úlohe PW-Commit sú spustené ešte moduly **Scan for compiler warnings** - kontrola behu PHP**, Console output parsing** - rozčlenenie konzolového výstupu, a **Editable Email Notification** - upozornenie o stavu zostavenia**.**

Obrázok 8.17: Závislosť akcií zostavenia úlohy PW-Multiprojects [vlastné]

8.2.5 Implementácia testov grafického rozhrania

Testy grafického rozhrania v produkte Planning Wizard sú použité na simuláciu reálneho užívateľa a sú najvhodnejšie pre automatizáciu, kvôli svojej časovej náročnosti. Pomocou týchto testov sme schopný otestovať aj funkčnosť produktu v rôznych webových prehliadačoch.

Na implementáciu testov je použitý aplikačný rámec PHPUnit a Selenium server. PHPUnit je do PwTesteru zakomponovaný ako balíček pomocou Composeru. Aby bolo možné testy spúšťať na webových prehliadačoch je potrebný aj zobrazovací server implementujúci protokol X11[22]. Na to je

[22] Viac informácií na URL: http://search.cpan.org/~smccam/X11-Protocol-0.56/Protocol.pm

použitý zobrazovací server Xvfb[23]. Xvfb a Selenium server bežia na otrokovi ako služby a to na základe skriptov *selenium* a *xvfb*.

Konvencia pomenovania a umiestnenia testov je nasledujúca:

- testové súbory sú uložené v testovacom rámci PwTester v zložke *tester/jenkins/selenium-tests*,
- konvencia pomenovania je *<názov-testovacieho-scénara>.<**selenium**>.**php.***

Ukážka testu na existenciu stránky s prihlasovacím formulárom:

```
class PwSelenium extends PHPUnit_Extensions_SeleniumTestCase
{
    protected $jobName;
    protected $captureScreenshotOnFailure = TRUE;
    protected $screenshotPath;
    protected $screenshotUrl;

    public static $browsers = array(
                array(
                        'name'    => 'Linux Firefox',
                        'browser' => '*firefox',
                        'host'    => 'localhost',
                        'port'    => 4444,
                        'timeout' => 30000,
                    ),
                array(
                        'name'    => 'Linux Chrome',
                        'browser' => '*chrome',
                        'host'    => 'localhost',
                        'port'    => 4444,
                        'timeout' => 30000,
                    )
                );

    protected function setUp()
    {
      $this->jobName = str_replace("=", "/", getenv('JOB_NAME'));
      $this->setBrowserUrl("http://localhost/");
      $this->screenshotPath = "path";
      $this->screenshotUrl = "url";
    }

    public function testLoginScreen()
    {
      $this->open("/workspace/{$this->jobName}/");
      $this->assertEquals("Přihlásit se", $this->getValue("id=loginBtn"));
    }
}
```

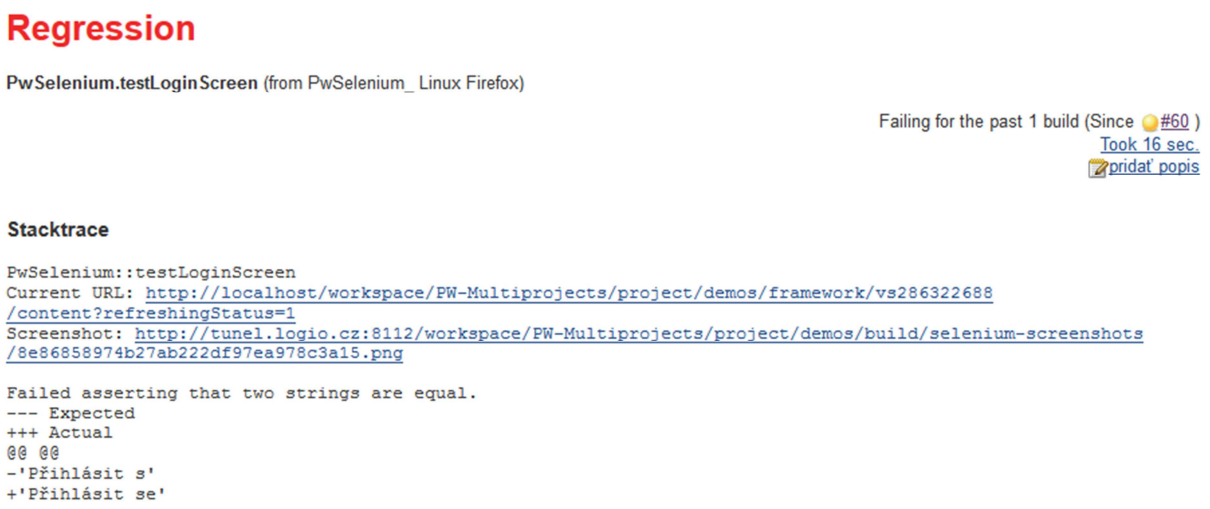
Štruktúra testov je rovnaká ako u jednotkových testov. Každý test musí dediť z triedy `PHPUnit_Extensions_SeleniumTestCase` a obsahovať metódu `setUp`. Táto metóda sa

[23] Viac informácií na URL: http://www.x.org/releases/X11R7.6/doc/man/man1/Xvfb.1.xhtml

spúšťa pred každým testom. Nastaví sa v nej adresa prehliadača pre testovanie a získa sa názov projektu, ktorý sa aktuálne zostavuje.

Do premennej `$browsers` sa definujú webové prehliadače, na ktorých sa majú testy uskutočniť. PHPUnit tiež dokáže pri chybe zachytiť aktuálny snímok obrazovky. Táto funkčnosť sa povoľuje pomocou `$captureScreenshotOnFailure = TRUE`. Po povolení je potrebné nastaviť cestu kam uložiť obrázky a adresu k ich zobrazeniu. Samotné testovacie metódy musia obsahovať predponu *test*.

Na tvorbu testov je možné použiť vývojové prostredie Selenium IDE a testy si priamo exportovať do formátu pre PHPUnit.



Obrázok 8.18: Ukážka chybového výstupu testu grafického rozhrania v Jenkinse [vlastné]

9 Testovanie a zhodnotenie výsledkov

Kapitola zhodnocuje dosiahnuté výsledky práce, ukážky prípadov odchytených chýb a vytvorených protiopatrení. Celá architektúra priebežnej integrácie a lokálneho testovania pomocou vytvoreného nástroja PwTester momentálne funguje k dátumu 20.5.2015 jeden mesiac v reálnej prevádzke.

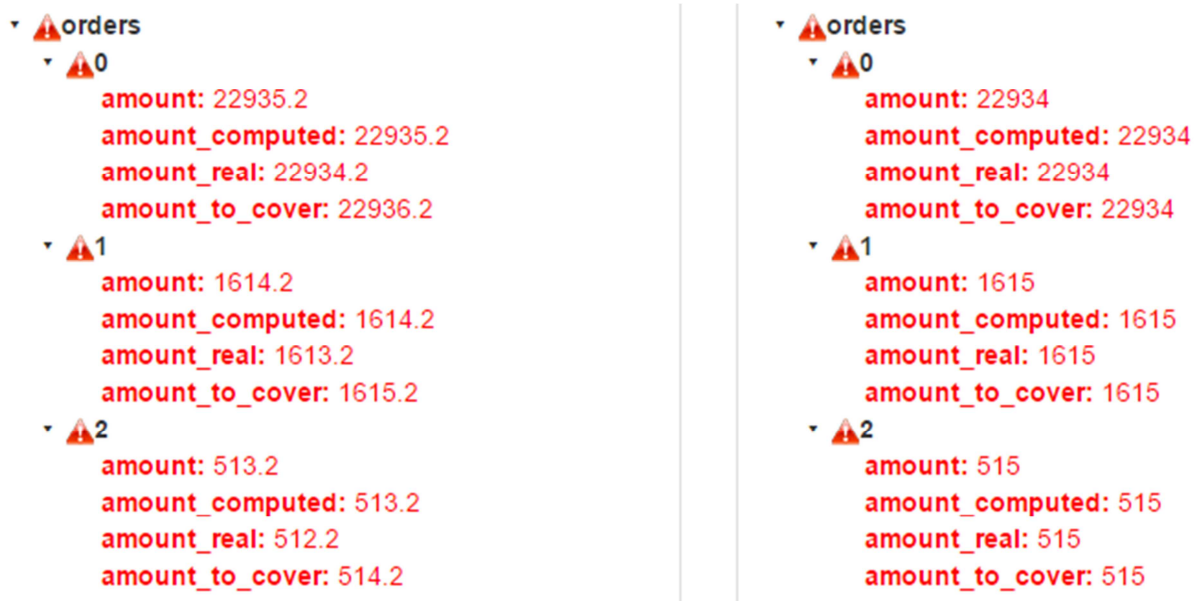
9.1 Ukážky odchytených chýb

Nasledujúce prípady poukazujú na jedny z najčastejších chýb odchytených navrhnutým systémom počas mesačnej prevádzky nad reálnymi dátami. U zistených chýb sa popisuje ich dôvod výskytu a spôsob odchytenia.

Integračné testy – chybné zaokrúhľovanie

Testovací rámec PwTester bol nasadený na reálny zákaznícky server a rovnako aj na testovací server obsahujúci stabilnú verziu kódov pre daný projekt. Na zákazníckom serveri boli vytvorené pahýle, ktoré sa berú ako základné dáta pre následne testovanie. Testy na zákazníckom serveri prebehli v poriadku. Je to z dôvodu, že pahýle sú vytvorené na týchto zdrojových súboroch.

Avšak testy na testovacom serveri neprešli, ich výstup sa líši v objednávanom množstve.



Obrázok 9.1: Ukážka rozdielu vo vypočítanom množstve [vlastné]

Ako môžeme vidieť množstvá sa líšia len v ráde desatinných čísel. Po preskúmaní kódu sa zistilo, že tento rozdiel bol spôsobený zlým zaokrúhlením. Na zákazníckom serveri bol kód pre výpočet objednávok doplnení o špecifickú metódu, ktorá výpočet pozmeňovala nesprávne.

Pomocou manuálneho vykonávania integračných testov sme teda zistili nežiaducu chyba na reálne bežiacom projekte.

Prvotným zámerom je otestovať týmto spôsobom všetky aktuálne projekty a odhaliť prípadné chyby. Po odstránení zistených chýb sú tieto projekty pridané do automatizovaného testovania ako jeden z projektov prepočítavaných úlohou PW-Multiprojects, kde dochádza k automatizovanej kontrole.

Úloha PW-Multiprojects – chybný databázový skript

Prvým dôležitým procesom, ktorý sa kontroluje pri prepočte systému Planning Wizard je aplikovanie rozdielových databázových skriptov. Tieto skripty upravujú vytvorenú databázu tak, aby bol zabezpečený správny beh produktu.

Tvorba týchto skriptov je podmienená požiadavkami zákazníkov a pridávaním rozširujúcich funkčností. Preto je ich úspešné vykonanie veľmi dôležité.

Na základe požiadavky pre novú funkčnosť sa vytvorí databázový skript napríklad pre pridanie stĺpca do tabuľky. Skript je uložený na centrálny repozitár a je otestovaný len na testovacej inštancii. V určitých prípadoch dochádza k odhaleniu chyby až po niekoľkých dňoch, keď dôjde k reálnemu použitiu skriptu, čo je neprijateľné.

Medzi časté zistené chyby patrí zlý zápis skriptu. Ten môže obsahovať buď syntaktickú chybu alebo neviditeľné znaky, čo znemožňuje jeho vykonanie. Syntaktická chyba je spôsobená rôznymi zápismi pre rozdielne verzie databázových serverov.

Všetky tieto chyby boli odchytené pomocou navrhnutej architektúry priebežnej integrácie. Úloha PW-Multiprojects dokáže po vytvorení potrebných databáz spustiť nástroj updater, ktorý je súčasťou Planning Wizardu. Tento nástroj vykoná jednotlivé skripty a informáciu o ich stave vráti Jenkinsu na spracovanie. Výsledkom je rýchlejšia spätná väzba a prehľadné zobrazenie chýb.

```
Error
3544: SQLSTATE[42S22]: Column not found: 1054 Unknown column 'access_righ' in 'field list'
3547: OK
3548: OK
```

Obrázok 9.2: Ukážka výstupu z nástroja updater o stavu vykonaných databázových skriptoch [vlastné]

Úloha PW-Multiprojects – chybný prepočet modulov

Prepočet produktu Planning Wizard zabezpečuje pomocou aktuálnych dát potrebné výstupy pre zákazníkov. Ide o spustenie jednotlivých modulov systému, ako je napríklad predpoveď predaja do nastaveného časového horizontu. Ide o jadro celého systému a neúspešné vykonanie len jedného z množstva modulov znamená zlyhanie systému a k dispozícii sú potom nereálne výsledky.

Prepočet celého systému nad všetkými dátami je časovo náročný. Z toho dôvodu je veľmi zložité zmenu otestovať ihneď po jej implementácii. K otestovaniu dochádza až na konci pracovného dňa a kontroluje sa väčší počet zmien. Tento postup je neefektívny pre dosiahnutie rýchleho vývoja.

Cieľom úlohy PW-Multiprojects je dané zmeny otestovať ihneď. Spustí prepočet systému nad všetkými modulmi okrem dvoch, ktoré slúžia na import a export dát do databáze, ktoré nie sú pri tomto prepočte potrebné. Na prepočet je použitá menšia reálna vzorka dát, ktorá umožní beh produktu a tiež skráti samotnú dobu prepočtu. Je dôležité, aby boli v tejto vzorke dát zastúpené všetky druhy dát, čím zabezpečíme vykonanie celého prepočtu a skontrolujú sa všetky zákutia systému.

Úloha PW-Multiprojects sa aktuálne spúšťa na základe úspešného zostavenia úlohy PW-Commit, ktorá sa inicializuje pri zistení zmien v repozitári. Tým dosiahneme pravidelnú kontrolu prepočtov na zvolených projektoch.

Najčastejšími zistenými chybami u modulov boli nesprávne napísané databázové príkazy. Je to dôsledok toho, že systém je primárne založený na prácu s dátami.

Výsledkom je pravidelná kontrola zmien na celom systéme, rýchlejší vývoj, časová úspora a prehľadný záznam o chybách v jednotlivých moduloch.


METHOD = processbatch, MODULE = batch-after-batch, CTIME = May 15 2015 13:00:42., FILE = batch-after-batch_processBatch_pid884998.txt

URL = /workspace/PW-Multiprojects/project/abb/framework/run.php?module=utils-processmanager&method=runProcess&batch&&params=a%3A1%3...

Zpráva:

Table 'tmp_product_locator_avg_amount' already exists
STACK TRACE:

Výjimka nastala v modulu: **dotnet**
Výjimka nastala v souboru: **/var/www/html/workspace/PW-Multiprojects/project/abb/modules/source/dotnet/dotnet/dotnet.php** na řádku **500**
Trace:
1) /var/www/html/workspace/PW-Multiprojects/project/abb/modules/source/dotnet/dotnet/dotnet.php (541) : CDotnet->processCallbacks(CControllingDisplayAbb)
2) /var/www/html/workspace/PW-Multiprojects/project/abb/modules/source/forecast/controlling-display/controlling-display.php (300) : CDotnet->callMethod(CControllingDisplayAbb, 'reportHistoryGraph.Execute', array('monthsLookBack' => 12, 'accuracyFunctionType' => 5))
3) /var/www/html/workspace/PW-Multiprojects/project/abb/modules/source/batch/batch-after-batch/batch-after-batch.php (35) : CControllingDisplay->generateControllingReportDotNet()
4) /var/www/html/workspace/PW-Multiprojects/project/abb/modules/source/batch/batch-after-batch/batch-after-batch.php (88) : CBatchAfterBatch->computeReportControlling()
5) /var/www/html/workspace/PW-Multiprojects/project/abb/specific/source/abb/extensions/batch-after-batch-abb/batch-after-batch-abb.php (21) : CBatchAfterBatch->compute()
6) /var/www/html/workspace/PW-Multiprojects/project/abb/shared/batch/batch/batch.php (139) : CBatchAfterBatchAbb->compute()
7) UNKNOWN (UNKNOWN) : CBatch->processBatch()
8) /var/www/html/workspace/PW-Multiprojects/project/abb/shared/utils/utils-processmanager/utils-processmanager.php (376) : call_user_func_array(array(0 => CBatchAfterBatchAbb, 1 => 'processBatch'), array())
9) UNKNOWN (UNKNOWN) : CUtilsProcessManager->runProcess(array('id' => '884998', 'type' => 'local', 'historyId' => NULL, ...))
10) /var/www/html/workspace/PW-Multiprojects/project/abb/framework/run.php (78) : call_user_func_array(array(0 => CUtilsProcessManager, 1 => 'runProcess'), array(0 => array('id' => '884998', 'type' => 'local', 'historyId' => NULL, ...), ...))

Obrázok 9.3: Ukážka zobrazuje príčinu chyby v module batch-after-batch [vlastné]

Úloha PW-Multiprojects – testy grafického rozhrania

Pomocou testov grafického rozhrania sme schopný odchytiť chyby nielen v logike systému ale aj v jeho zobrazovaní. To, že prepočet systému prebehol v poriadku ešte nemusí znamenať plnú funkčnosť.

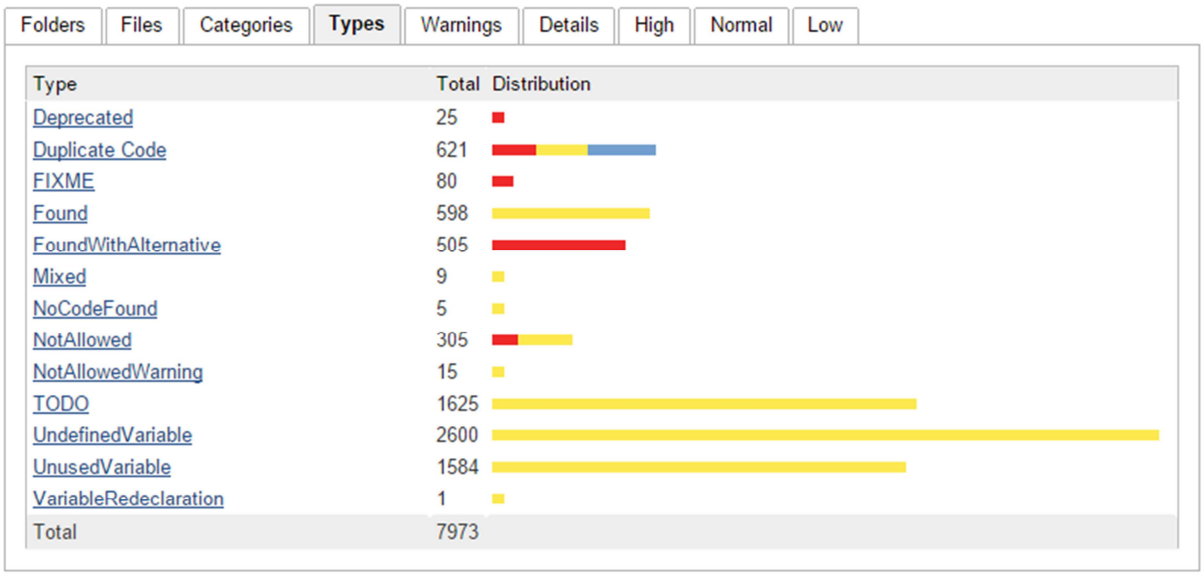
Počas reálnej prevádzky sa podarilo odchytiť chybné jquery skripty a chybné databázové príkazy pre získanie dát a ich následne zobrazenia v príslušných tabuľkách.

Výsledkom je časová úspora, rýchla spätná odozva a k dispozícii sú snímky obrazovky s chybami.

Úloha PW-Commit – statická analýza

Predpoklady o zlom stave Planning Wizardu z pohľadu statickej analýza sa potvrdili. Aktuálne sa v projekte nachádza celkovo 7 973 chýb. V rozdelení podľa priorít ide o 875 chýb s vysokou prioritou, 6 841 s normálnou a 257 s nízkou prioritou. Typy chýb a ich počet sa nachádza na obrázku na nasledujúcej stránke.

Výsledkom je detailný prehľad o type a počte chýb. Vieme rozlíšiť priority a zasiahnuť v najakútnejších prípadoch.



Obrázok 9.4: Prehľad statickej analýzy podľa typu chýb [vlastné]

9.2 Zhodnotenie stavu a výsledkov

Aktuálne sa po mesačnej prevádzke ešte nedá kvantifikovať prínos riešenia, ten bude možný s väčším časovým odstupom.

Testovací rámec je však nasadený na piatich reálnych zákazníckych serverom, s tým že pre všetky tieto projekty existujú stabilné inštancie na testovacom serveri. Je to z dôvodu, aby sme boli schopný otestovať pahýle aj na zdrojových súboroch z repozitára a otestovať prípadné zmeny manuálne.

Z týchto piatych projektov boli vytvorené vzorky dát, ktoré slúžia na automatizovanú kontrolu celého prepočtu produktu Planning Wizard. Vďaka tejto architektúre sme schopný odchytiť chyby v rámci hodín čo je mnohonásobné zlepšenie oproti predchádzajúcemu spôsobu. Tiež je pokrytá väčšia časť systému vďaka jednotkovým, integračným testom, testom grafického rozhrania a prepočtu systému.

Pomocou úlohy PW-Commit je kontrolovaná každá zmena v hlavnej vývojovej vetve repozitára. Programátori teda získavajú rýchlu spätnú odozvu každých pätnásť minút na všetky svoje zmeny, spolu s informáciami o statickej analýze. Získali sme tak efektívnejší vývoj a povzbudili programátorov v snahe písať prehľadnejší a z pohľadu noriem čistejší kód.

Celý testovací rámec a architektúra priebežnej integrácie je spracovaná tak, aby bolo jednoduché pridávať ďalšie testy a vytvárať nové úlohy v nástroji Jenkins. Samotný programátori sú aktuálne schopný vytvárať si vlastné jednotkové testy. V prípade integračných testov ide o zložitejšiu implementáciu ale už vzniklo pár požiadaviek na ich rozšírenie.

Celkový prínos je zo strany spoločnosti zhodnotený pozitívne. Dosiahlo sa aktuálne veľkej časovej úspory na strane programátorov a povzbudila sa ich snaha testovať svoj kód. Testami sa podarilo pokryť najdôležitejšie časti produktu Planning Wizard. K dispozícii je prehľadný zoznam chýb a štatistík z rôznych časových období.

10 Možné rozšírenia

Kapitola popisuje možné nové funkčnosti vhodné pre vytvorený testovací rámec PwTester a zavedenú priebežnú integráciu. Nasledujúce rozšírenia boli konzultované s programátormi v spoločnosti Logio, teda najčastejšími užívateľmi celého systému.

10.1 Rozšírenia PwTester

Najbližšími rozšírením pre rámec PwTester bude doplnenie integračných testov ďalšími testami na algoritmy v produkte Planning Wizard, ako je napríklad predpoveď predaja či výber vhodného dodávateľa. Aktuálne sa pracuje na testoch, ktoré budú pokrývať ďalšiu dôležitú časť produktu a to import zákazníckych dát pomocou nového vyvíjaného nástroja.

V pláne je doplniť testovací rámec o ďalšie úrovne testovania vychádzajúce z V-modelu ako sú systémové a akceptačné testy. Zámerom je pokryť testami čo najväčšiu časť produktu Planning Wizard.

Čo sa týka doposiaľ implementovaných integračných testov pre výpočet objednávok vznikla potreba na implementáciu odchýlky. Dôvodom je, že v niektorých prípadoch, keď sa napríklad objednávky na produkt líšia povedzme v množstve tri, nepovažuje sa tento rozdiel za chybu. Rozdiel v množstve môže byť spôsobený špecifickým výpočtom, dodávateľom a atď. Odchýlka bude závisieť vždy na špeciálnych nastaveniach a eventualitách.

Pre uľahčenie vyhľadávania chyby v zdrojových súbor by bolo vhodné ku dátovej štruktúre pahýľ prikladať aj zdrojové súbory. To by nám umožnilo porovnať súbory, ktoré pahýľ vytvárali spolu so súbormi, na ktorých sú spustené testy a zobraziť ich rozdiely.

10.2 Rozšírenia pre priebežnú integráciu

U priebežnej integrácie ide o implementáciu ďalších rozširujúcich vlastností a technik. Tieto nové funkčnosti sú plánované s postupom času.

Prvým rozšírením by malo byť zabezpečiť prepočet produktu aj na zdrojových súboroch nachádzajúcich sa na zákazníckych serveroch. Dôvodom sú špecifické úpravy kódov priamo na serveri a ich absencia v repozitári Subversion. Pre aplikovanie tejto funkčnosti je potreba pozmeniť architektúru a pravidlá používané v spoločnosti Logio čo je z časového hľadiska aktuálne nereálne.

Častým návrhom na rozšírenie od programátorov je mať možnosť pomocou jednoduchého rozhrania vytvoriť a nastaviť si úlohu pre systém Jenkins. Túto funkcionalitu by využívali pri implementácií nových projektov. Umožnilo by im to rýchlejší a bezpečnejší vývoj. Rozhranie pre nastavenie úlohy by bolo implementované ako ďalší nástroj v Planning Wizarde.

Rovnako sa počíta s postupným rozširovaním techník priebežnej integrácie a ako najbližší krok je vhodná technika priebežného nasadzovania. Tým sa umožní automatizované nasadzovanie produktu priamo ku zákazníkovi. Výhodou tejto techniky je aj automatizácia vytvárania stabilných verzií a ich verzovanie v repozitári. V spoločnosti Logio sa o tvorbu týchto verzií starajú aktuálne dvaja programátori. Samotné zostavenie a testovanie voči všetkým reálne bežiacim projektom musia

vykonávať manuálne čo je časovo náročné. K uvoľňovaniu stabilných verzií dochádza raz za dva mesiace čo je pre zavedenie rýchlejšieho vývoja neprípustné. Túto situáciu by mala práve zvrátiť technika priebežného nasadzovania, ktorá dokáže znížiť časovú náročnosť na tvorbu verzií a zrýchlenie ich zverejňovania.

Pre zaistenie čo najrýchlejšej reakcie na chybné zostavenia a efektívnejšiemu získavaniu informácií sme sa po dohode s vedením spoločnosti Logio zhodli na zaistení systému, ktorý by to umožnil. Ako najvhodnejšie riešenie sa ponúka zabezpečiť televízor a umiestniť ho na viditeľné miesto. Jenkins priamo disponuje zásuvnými modulmi pre takéto riešenia.

11 Záver

Cieľom práce je zoznámenie sa s použitím praktik a nástrojov vhodných pre implementáciu priebežnej integrácie a testovania v zázemí spoločnosti Logio. Následne zhodnotiť situáciu spoločnosti v oblasti problémov pri vývoji produktu Plannig Wizard a na základe potrieb, a požiadaviek implementovať získané znalosti.

Druhá kapitola práce spisuje základné techniky a pravidlá pre implementáciu priebežnej integrácie, spolu s výhodami jej použitia pri vývoji. Ďalej sú spísané dôvody kvôli, ktorým sa implementuje táto metodika v softvérových spoločnostiach.

Tretia kapitola popisuje v súčasnej dobe vhodné nástroje podporujúce priebežné integráciu pre programovací jazyk PHP.

Jednou z hlavných úloh priebežnej integrácie je testovanie. Preto sa v ďalšej kapitole s číslom štyri práca zaoberá popisom rôznych testovacích prístupov a spôsobov testovania. Kapitola obsahuje základne rozdelenie a druhy testov spolu s V-modelom životného cyklu vývoja, ktorý najlepšie definuje rozdelenie na rôznych úrovniach vývoja.

Zmyslom priebežnej integrácie je úlohy automatizovať čo sa týka aj testovania. Preto piata kapitola obsahuje definíciu ich automatizácie a výhody z nej plynúce spolu s nástrojmi, ktoré sú najvhodnejšie pre vytvorenie automatizovaných testov v jazyku PHP.

Šiesta kapitola popisuje vykonanú analýzu problémov pri vývoji a potrieb v spoločnosti Logio, ktoré stoja za možnosťou užitia metodiky priebežnej integrácie.

Na základe získaných poznatkov a analýze nárokov vznikol návrh architektúry priebežnej integrácie a štruktúry jednotlivých projektov s využitím nástroja Jenkins. Spolu s rozdelením a definíciou testov vhodných pre produkt Planning Wizard. Celý návrh sa nachádza v siedmej kapitole tejto práce.

Ôsma kapitola sa skladá z dvoch hlavných častí implementácie. Prvá časť popisuje implementáciu testovacieho rámca PwTester určeného pre manuálne spúšťanie jednotkových a integračných testov. Druhá časť definuje štruktúru a obsah zloženia priebežnej integrácie určenej pre spoločnosť Logio. Popisujú sa jednotlivé použité zásuvné moduly a nástroje pre vykonanie testov a statickej analýzy v Jenkinse.

Deviata kapitola zhodnocuje dosiahnuté výsledky práce a ukážky najčastejších odchytených chýb počas mesiaca reálnej prevádzky.

V predposlednej desiatej kapitole sú návrhy na možné rozšírenia rámca PwTester a pokračovanie vo využití techník a metód priebežnej integrácie. Rozšírenia boli prediskutované s reálnymi užívateľmi celého systému. Prípadná implementácia týchto rozšírení by mala ešte viac zvýšiť automatizáciu procesov a zefektívniť vývoj.

Implementované riešenie prináša zrýchlenie vývoja a spätnej väzby u produktu Planning Wizard. Štruktúra testov a priebežnej integrácie bola navrhnutá tak, aby bolo možné daný systém ľahko doplniť o ďalšie techniky a typy testov. Autor sa bližšie zoznámil s technikou priebežnej integrácie a možnosťami testovania softvéru spolu s reálnym aplikovaním získaných znalostí.

Literatura

[1] BECK, Kent a Cynthia ANDRES. *Extreme programming explained: embrace change.* Vyd. 2. Boston, MA: Addison-Wesley, 2005. ISBN 03-212-7865-8.

[2] BOOCH, Grady. *Object-oriented analysis and design with applications*. Vyd. 2. Massachusetts: Addison-Wesley, 1994. ISBN 0-8053-5340-2.

[3] FOWLER, Martin. *Continuous Integration* [online]. 2006-05-01 [cit. 2014-11-13]. Dostupné z: http://www.martinfowler.com/articles/continuousIntegration.html.

[4] DUVALL, Paul M, MATYAS, Steve a Andrew GLOVER. *Continuous Integration: improving software quality and reducing risk*. Upper Saddle River, NJ: Adison-Wesley, 2007. ISBN 03-213-3638-0.

[5] FARLEY, Jez Humble a David, MATYAS, Steve a Andrew GLOVER. *Continuous Delivery*. Upper Saddle River, NJ: Addison-Wesley, 2011. ISBN 03-216-0191-2.

[6] WIEST, Simon. Hudson – Your Escape from "Integration Hell". *Software development magazine: software testing, project management, Agile, Scrum, UML, programming, requirements* [online]. 2010 [cit. 2014-11-15]. Dostupné z: http://www.methodsandtools.com/tools/tools.php?hudson

[7] KAWAGUCHI, Kohsuke. Meet Jenkins. *Welcome to Jenkins CI!* [online]. 2013 [cit. 2014-11-14]. Dostupné z: https://wiki.jenkins-ci.org/display/JENKINS/Meet+Jenkins

[8] JENKINS. *Jenkins 1.580.1* [software]. [prístup 20. decembra 2014]. Dostupné z: http://jenkins-ci.org/

[9] TeamCity :: Distributed Build Management and Continuous Integration Server – Features. *Continuous Integration for Everybody – TeamCity* [online]. © 2000-2015 [cit. 2014-11-15]. Dostupné z: https://www.jetbrains.com/teamcity/features/

[10] BALLIAUW, Maarten. Getting started with PHP. *TeamCity Documentation – TeamCity 9.x Documentation* [online]. 2014 [cit. 2014-11-20]. Dostupné z: https://confluence.jetbrains.com/display/TCD9/Getting+started+with+PHP

[11] PICHLER, Manuel. PhpUnderControl - Continuous Integration for PHP. *PhpUnderControl* [online]. 2010 [cit. 2014-11-15]. Dostupné z: http://phpundercontrol.org

[12] MYERS, Glenford J. *The art of software testing*. Vyd. 2. Hoboken: Wiley, 2004. ISBN 978-0-471-46912-4.

[13] KANER, Cem. Exploratory Testing. In: *Improving the Education of Software Testers* [online]. 2006 [cit. 2015-01-03]. Dostupné z: http://www.kaner.com/pdfs/ETatQAI.pdf

[14] What is fundamental test process in software testing?. *ISTQB Exam Certification* [online]. © 2015 [cit. 2015-01-03]. Dostupné z: http://istqbexamcertification.com/what-is-fundamental-test-process-in-software-testing/

[15] DIJKSTRA, Edsger W. *The Humble Programmer* [online]. 1972 [cit. 2015-01-03]. Dostupné z: https://www.cs.utexas.edu/~EWD/ewd03xx/EWD340.PDF

[16] MINDUCA, Arthur. Quality assurance in software development: When should you start the testing process?. *Arthur Minduca .NET programming and related topics in desktop and web development* [online]. 2014 [cit. 2015-01-04]. Dostupné z: http://arthurminduca.com/

[17] LEWIS, William E a Gunasekaran VEERAPILLAI. *Software testing and continuous quality improvement.* Vyd. 2. Boca Raton: Auerbach Publications, 2005. ISBN 08-493-2524-2.

[18] PATTON, Ron. *Testování softwaru*. Vyd. 1. Praha: Computer Press, 2002. ISBN 80-722-6636-5.

[19] BOROVCOVÁ, Anna. *Testování webových aplikací.* Praha, 2008. Diplomová práca. Univerzita Karlova v Praze, Matematicko-fyzikální fakulta. Vedúci práce Mgr. Vladan Majerech, Dr. Dostupné z: https://is.cuni.cz/webapps/zzp/detail/46536/

[20] GRAHAM, Dorothy, VEENENDAAL, Erik van, EVANS, Isabel a Rex BLACK. *Foundations of software testing: ISTQB CERTIFICATION.* Vyd. 2. London: Cengage Learning EMEA, 2008. ISBN 978-1-84480-989-9.

[21] AMMANN, Paul. *Introduction to software testing.* New York: Cambridge University Press, 2008. ISBN 978-0-521-88038-1.

[22] FARRELL-VINAY, Peter. *Manage software testing.* Boca Raton: Auerbach Publications, 2008. ISBN 978-0-8493-9383-9.

[23] PAVLÍK, Vladimír. *Testing Software Systems.* 2014 [2015-01-10]. Dostupné z: www.cs.vsb.cz/jezek/vyuka/tss/download/Testing%20Software%20Systems%20CZ.pptx

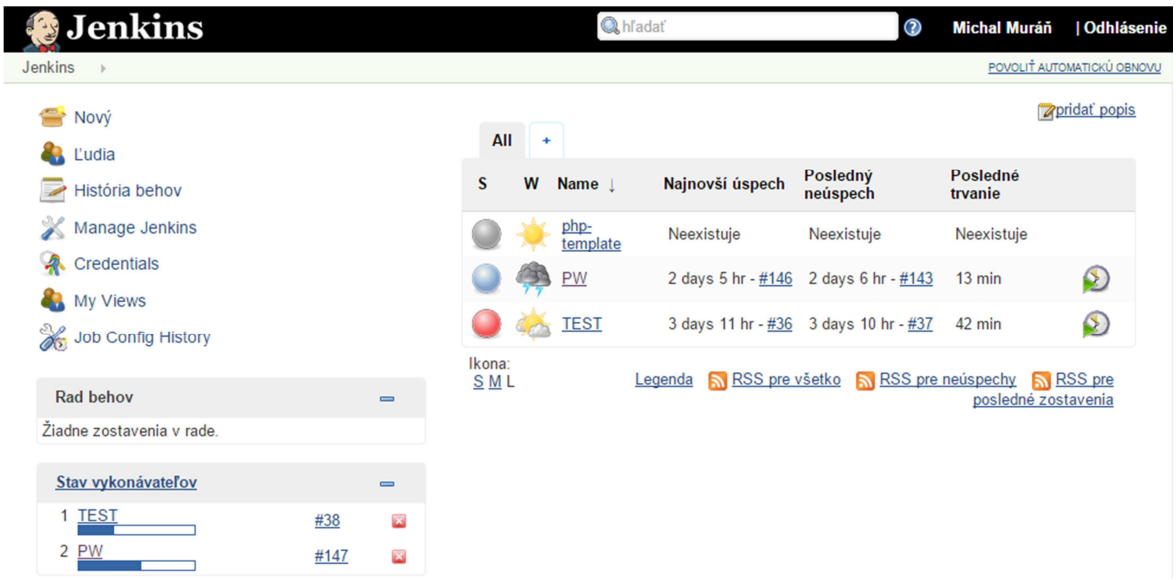
[24] BERGMANN, Sebastian. Documentation for PHPUnit. *Welcome to PHPUnit!* [online]. © 2005-2015 [cit. 2015-01-7]. Dostupné z: https://phpunit.de/manual/current/en/index.html

[25] Nette Tester – pohodové testování. *Rychlý a pohodlný vývoj webových aplikací v PHP* [online]. © 2008, 2015 [cit. 2015-01-7]. Dostupné z: http://tester.nette.org/

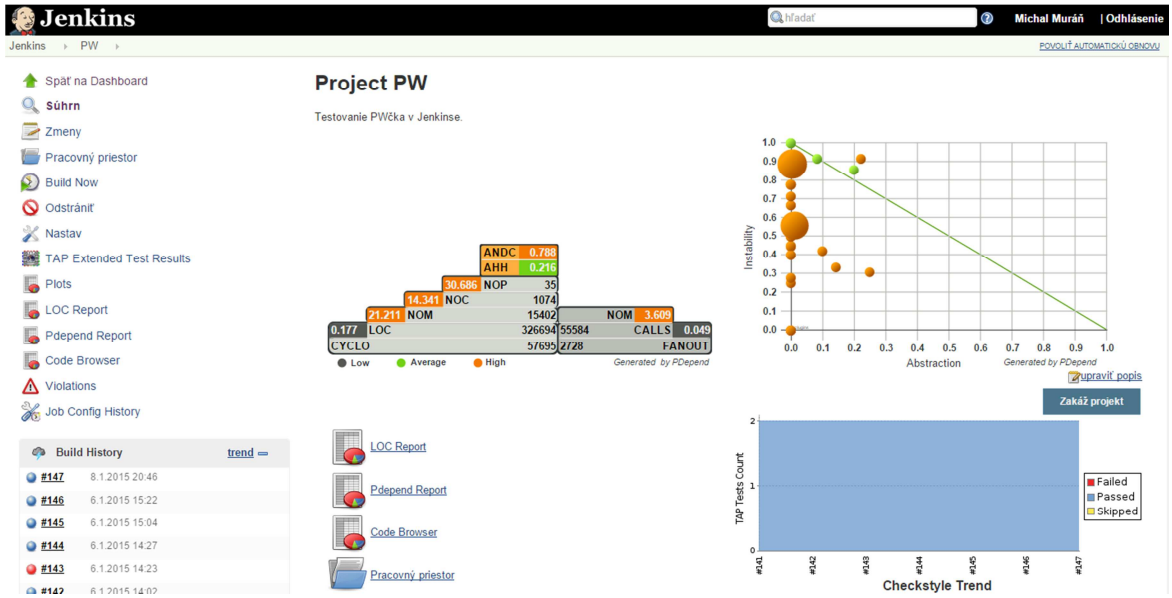
[26] Selenium Documentation. *Selenium - Web Browser Automation* [online]. ©2008-2012 [cit. 2015-01-7]. Dostupné z: http://www.seleniumhq.org/docs/

Zoznam príloh

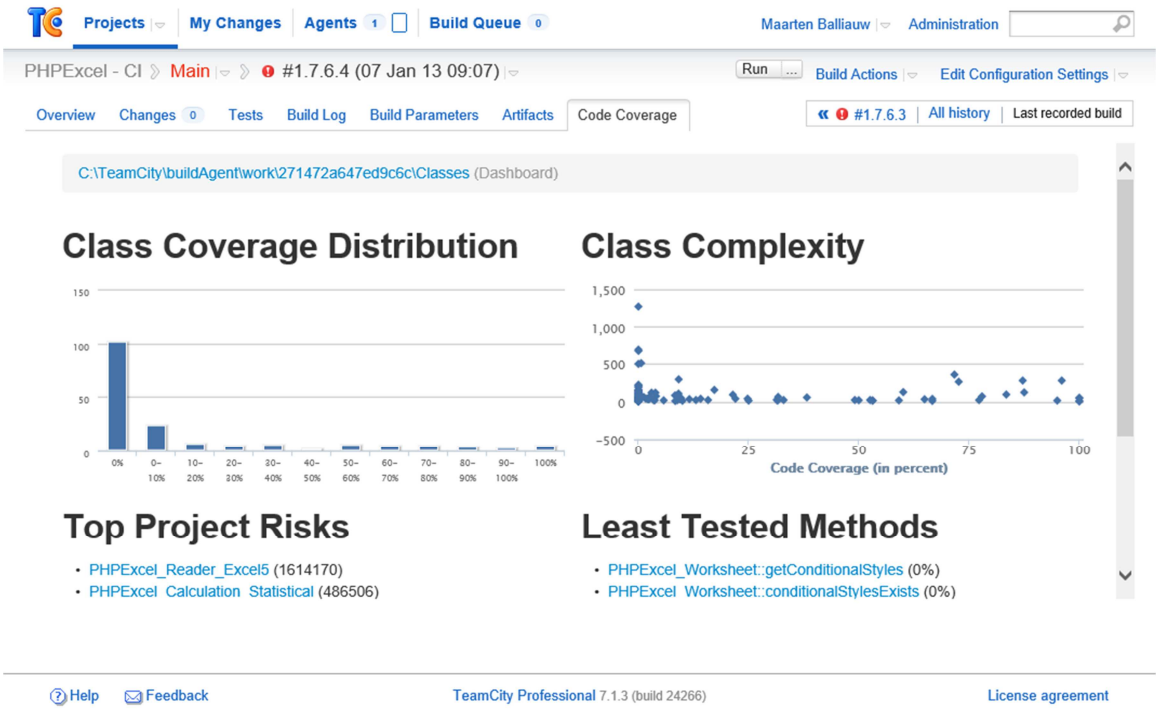
Príloha A. Obrázky nástrojov priebežnej integrácie



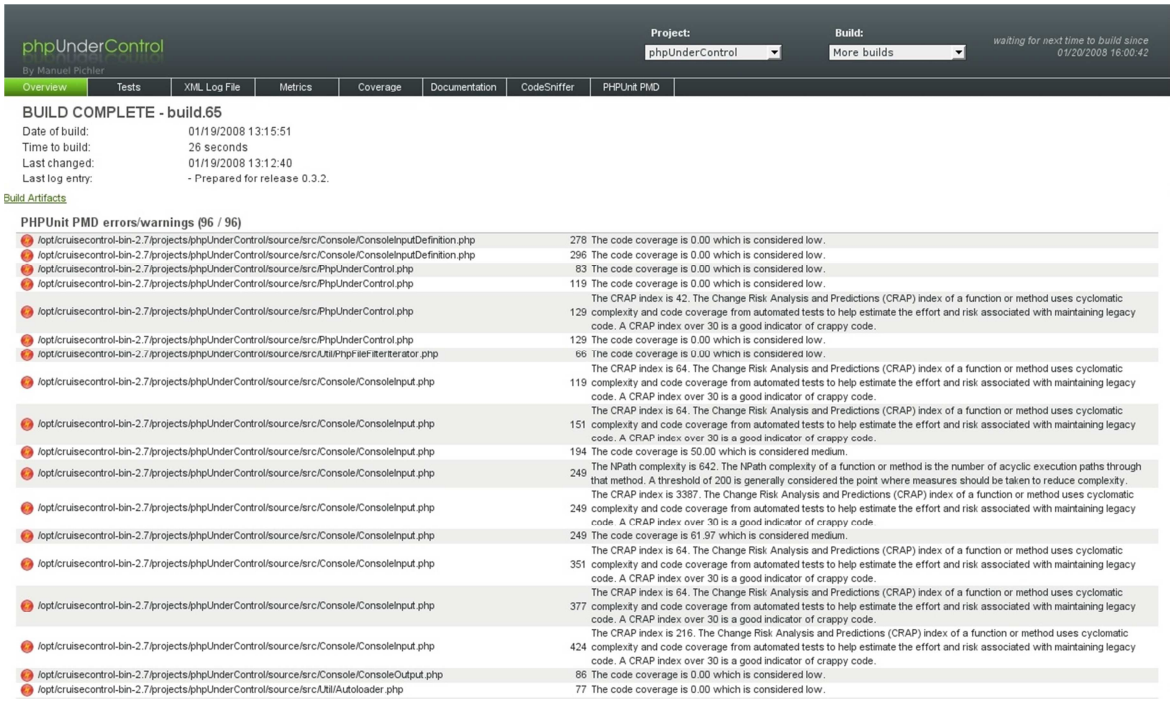
Obrázok A.1: Jenkins úvodná obrazovka s jednotlivými projektmi [8]



Obrázok A.2: Jenkins ukážka projektu s dostupnými základnými informáciami [8]

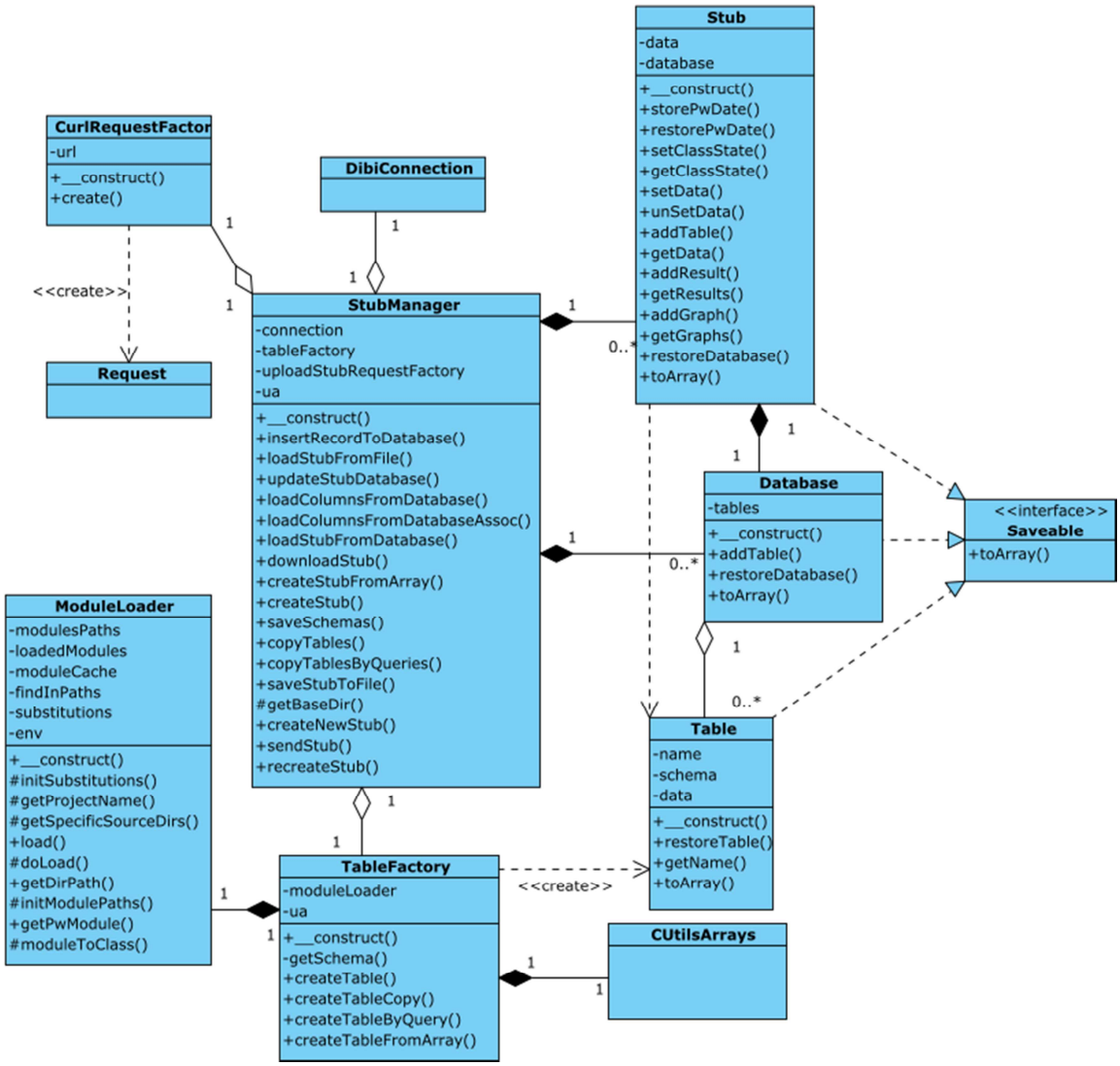


Obrázok A.3: TeamCity zobrazenie metrík projektu napísaného v PHP [10]



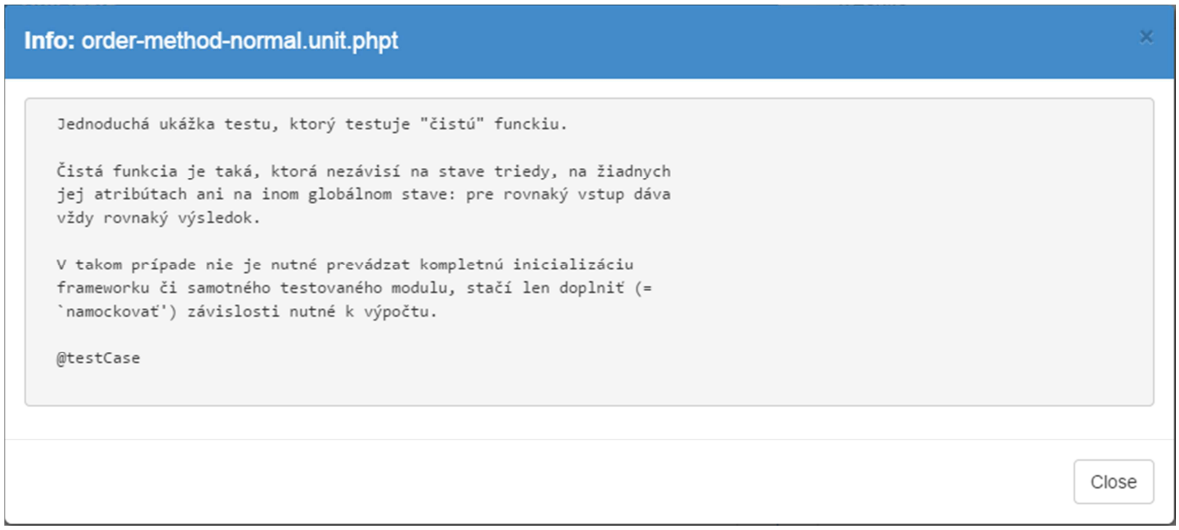
Obrázok A.4: PhpUnderControl ukážka výsledku zo zostavenia [11]

Príloha B. Diagramy tried

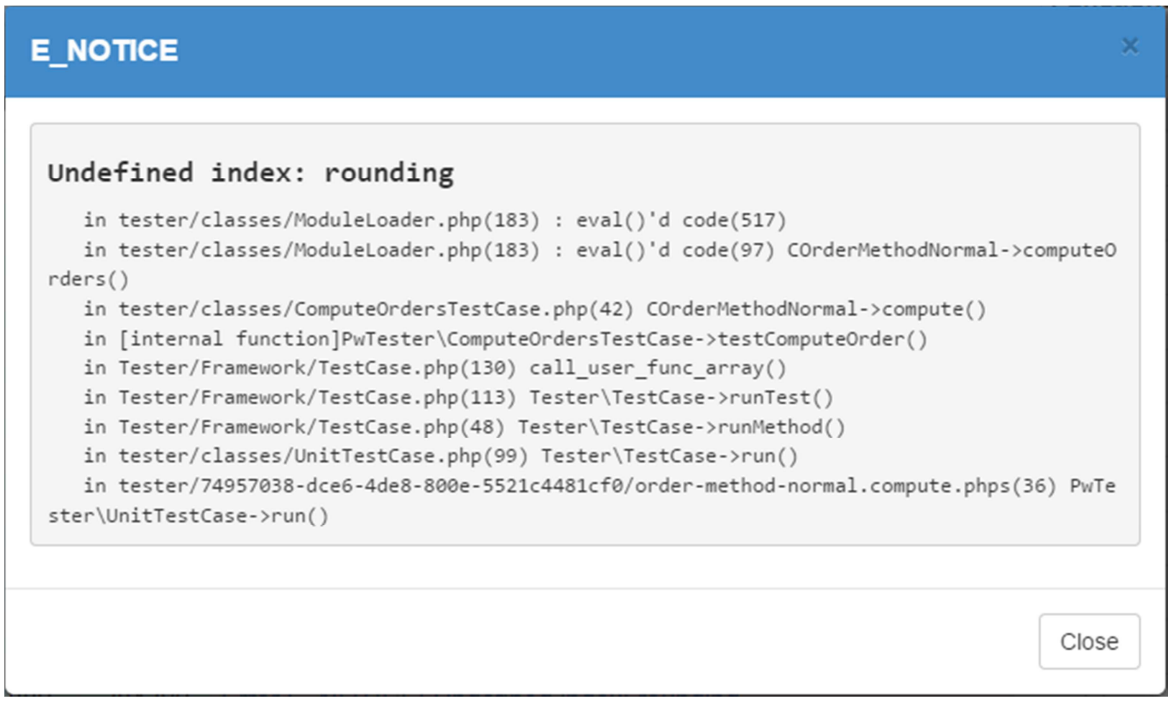


Obrázok B.1: Diagram tried reprezentujúci prácu s pahýľom [vlastné]

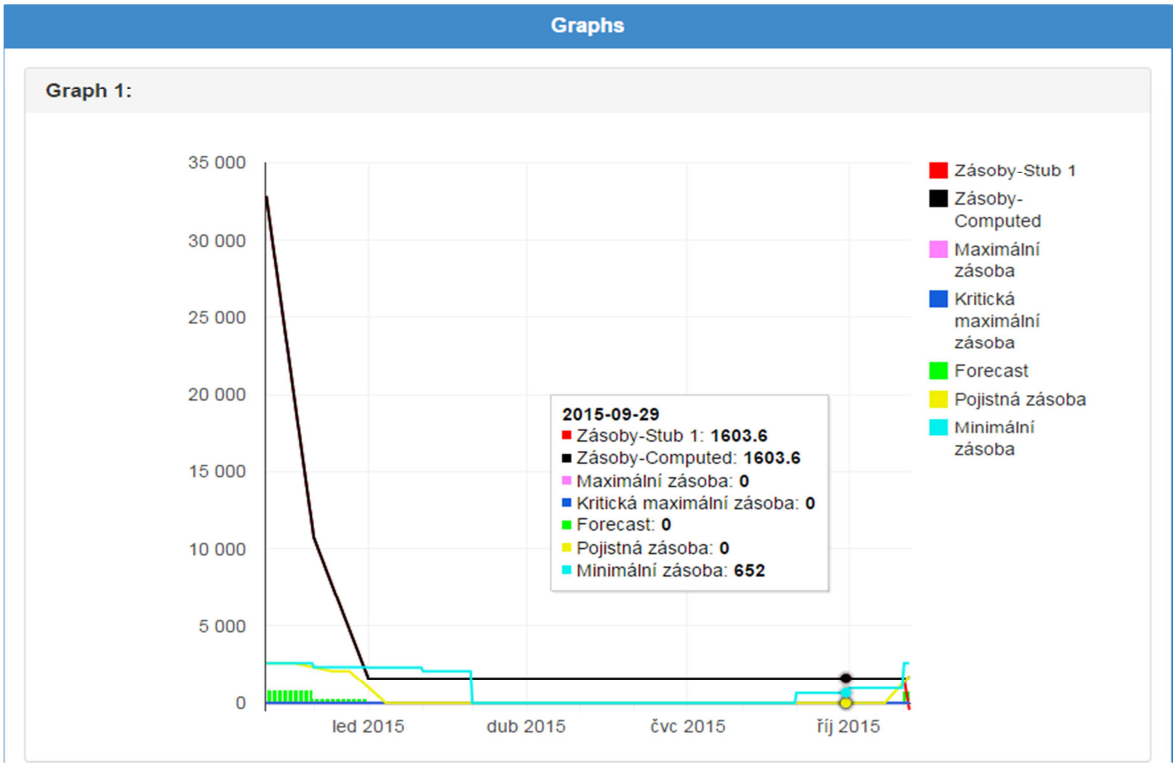
Príloha C. Obrázky grafického rozhrania



Obrázok C.1: Dialógové okno obsahujúce informácie o testovacej triede [vlastné]



Obrázok C.2: Dialógové okno nesúce bližšie informácie o chybe [vlastné]



Obrázok C.3: Reprezentácia dát objednávok pomocou grafu [vlastné]

Príloha D. Ukážky zdrojových kódov

Ukážka skriptu *build-setup.xml*, ktorý nastavuje prostredie pred zostavením:

```
<?xml version="1.0" encoding="UTF-8"?>
<!--Inicializacia prostredia pre jenkins-->
<project name="Setup" default="build">
    <!--Cesta ku korenovenu adresaru-->
    <property name="path" value="${basedir}/../../.."/>
    <!--Cesta k nastrojom-->
    <property name="toolsdir" value="${path}/composer/vendor/bin/"/>
    <echo message="Build part setup"/>

    <!--Build-->
    <target name="build" depends="prepare,Setup-PW" description=""/>

    <!--Clean-->
    <target name="clean" unless="clean.done"
         description="Cleanup build artifacts">
        <delete dir="${path}/cache/tester/taps"/>
        <delete dir="${path}/build/logs"/>
        <delete dir="${path}/build/selenium-screenshots"/>
        <delete dir="${path}/build/tap"/>
        <delete dir="${path}/build/batch"/>
        <delete dir="${path}/build/updater"/>
        <delete dir="${path}/build/code-browser"/>
        <delete dir="${path}/build/pdepend"/>
        <delete dir="${path}/build/api"/>
        <property name="clean.done" value="true"/>
    </target>

    <!--Prepare-->
    <target name="prepare" unless="prepare.done" depends="clean"
             description="Prepare for build">
        <mkdir dir="${path}/build/logs"/>
        <mkdir dir="${path}/build/selenium-screenshots"/>
        <mkdir dir="${path}/build/tap"/>
        <mkdir dir="${path}/build/batch"/>
        <mkdir dir="${path}/build/updater"/>
        <mkdir dir="${path}/build/code-browser"/>
        <mkdir dir="${path}/build/pdepend"/>
        <mkdir dir="${path}/build/api"/>
        <property name="prepare.done" value="true"/>
    </target>

    <!--Spustenie skriptu pre nastavenie PW-->
    <target name="Setup-PW">
        <exec dir="${path}" executable="/bin/sh">
            <arg value="tools/tester/jenkins/jenkins-setup.sh"/>
        </exec>
    </target>
</project>
```

Príloha E. Návod k zdrojovým kódom testovacieho rámca PwTester

Testovací rámec PwTester je primárne určený pre testovanie produktu Planning Wizard od spoločnosti Logio. Pre správny beh testovacieho rámca je preto potrebné ho prepojiť s produktom Planning Wizard.

Samotný PwTester sa používa na lokálne testovanie a automatizované testovanie pomocou priebežnej integrácie v nástroji Jenkins.

Adresárová štruktúra odovzdaných zdrojových kódov:

- **composer** – nástroj pre správu závislostí v PHP. Súbor composer.json obsahuje zoznam potrebných balíčkov.
- **order-method-by-basestock** – ukážka integračného testu pre objednávkovú metódu typu basestock.
- **order-method-normal** – ukážka integračného a jednotkového testu pre objednávkovú metódu typu normal.
- **tester** – zložka obsahuje jadro testovacieho rámca PwTester spolu so zložkou *jenkins* obsahujúcou skripty a nastavenia pre nástroj priebežnej integrácie Jenkins.
- **utils-arrays** – pomocný modul pre prácu s poliami.
- **utils-reflection** – pomocný modul na využitie spätného inžinierstva. Využíva sa na prístup ku triedam a získavanie informácií z nich.
- **utils-stub-button** – ide o modul, ktorý vytvára tlačítko pre vytváranie pahýľov.
- **utils-tester** – modul zaobaľujúci PwTester na možné použitie v produkte Planning Wizard.
- **selenium** – skript starajúci sa o beh Selenium Grid na servery ako služba.
- **xvfb** – skript starajúci sa o beh Xvfb na servery ako služba.
- **ukazkove-kody.php** – ukážky časti kódu ako je vytvorenie tlačítka pomocou modulu utils-stub-button, kód vytvárajúci pahýľ a kód starajúci sa o beh testu v testovanom algoritme.

1. Lokálne testovanie

Pre správne fungovanie lokálneho testovania je potreba zdrojové súbory zakomponovať do programového rámca produktu Planning Wizard.

Požiadavky pre spustenie PwTesteru pre lokálne testovanie:

- prepojenie s produktom Planning Wizard,
- minimálne PHP 5.3.0,
- potreba pripojenia na internet,
- PwTester je založený na nástroji Nette Tester, treba pomocou Composeru nahrať balík,
- ostatné pomocné balíky sú zadefinované v composer.json, spustiť príkaz `composer update`
- databázový server, ktorý sa stará o dátové pahýle (stub).

2. Priebežná integrácia – Jenkins

Pre automatizáciu testovania využívame priebežnú integráciu konkrétne nástroj Jenkins, ktorý je verejne dostupný pod licenciou MIT. Viac informácii o tomto nástroji je v diplomovej práci v kapitole 3.1 Jenkins. Štruktúra jednotlivých úloh navrhnutých pre správne fungovanie v spoločnosti Logio nad produktom Planning Wizard je popísaná v diplomovej práci v kapitole – 8.2 Implementácia priebežnej integrácie.

Všetky potrebné skripty a nastavenia sa nachádzajú v zložke *jenkins*. Súbory s predponou *build* sú skripty spúšťané nástrojom na zostavenie ant cez Jenkins.

Pre vykonanie testov grafického rozhrania je potrebné na servery disponovať Selenium Grid a Xvfb, ktoré sú zaregistrované ako služby pomocou skriptov *selenium* a *xvfb*.

Integračné testy sú založené na základe pahýľov, ktoré sa nachádzajú na centrálnom databázovom servery. Nutný je teda prístup k tomu serveru.

Na databázovom serveri sa tiež nachádzajú skripty databáz projektov. Tieto skripty obsahujú reálne dáta, na ktorých sa spúšťajú prepočty, ktoré budú potrebné pri prepočítavaní týchto projektov.

Obsah zložky *jenkins*:

- **build-setup.xml** – inicializuje prostredie (vytvorí adresárovú štruktúru).
- **build-batch.xml** – spúšťa skript na spustenie prepočtu produktu Planning Wizard.
- **build-static_analyze.xml** – spúšťa nástroje, ktoré vykonajú statickú analýzu nad zdrojovými súbormi.
- **build-stub.xml** – spustenie integračných testov.
- **build-unit.xml** – spustenie jednotkových testov.
- **build-selenium.xml** – spustenie testov grafického rozhrania.
- **jenkins-batch.php** – skript obsahuje logiku zabezpečujúcu vykonanie databázových skriptov, prepočtu produktu Planning Wizard spolu s ich kontrolou.
- **jenkins-setup.php** – skript nastaví prostredie pre chod produktu Planning Wizard.
- **run-stub-tests.php** – logika pre spustenie integračných testov.
- **run-unit-tests.php** – logika pre spustenie jednotkových testov.
- **settings** – zložka obsahuje súbory s nastaveniami pre nástroje PHPUnit a PHP_CodeSniffer.
- **selenium-tests** – zložka obsahuje testy grafického rozhrania.

Príloha F. Obsah CD

- `/Návod k zdrojovým kódom/` – návod k zdrojovým kódom vo formáte pdf a docx.
- `/Zdrojové kódy/PwTester/` – zdrojové kódy vytvoreného testovacieho rámca s názvom PwTester.
- `/Zdrojové kódy/Zip/` – zdrojové kódy PwTesteru zabalené v .zip.
- `/Písomná správa v PDF/` – písomná správa vo formáte pdf.
- `/Zdrojový tvar písomnej práce/` – písomná práca v zdrojovom tvare formát .docx.