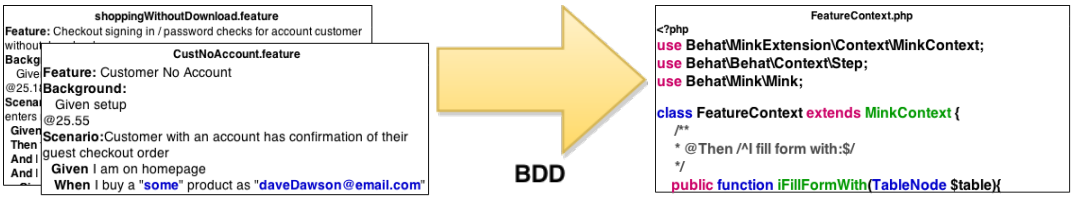
Excel@FIT 2015
http://excel.fit.vutbr.cz

Případ užití procesu BDD v testování aplikací v PHP

Farida Kudaiberdiyeva



Abstrakt

Tato práce je zaměřena na odhalení hlavních výhod a nevýhod použití chováním řízeného vývoje firmami jak v České Republice, tak i v zahraničí. Věnuje se odlišnostem a mezerám v tomto přístupu testování webových aplikací oproti klasickému přístupu. Za klasický přístup se považuje použití nástroje **Selenium** při tvorbě testovacích sad. Z těchto předpokladů vyplynul **cíl** této práce, kterým je *zjistit, zda použití BDD procesu je vhodné a efektivní oproti klasickému přístupu testování software*.

Pro dosažení cíle byly porovnány 2 nástroje: **PHP Selenium Client od Nearsoft** a **Behat**. Kritéria, zvolená pro porovnání, jsou: odhalené chyby, způsob zápisu a vykonání scénáře, srozumitelnost pro počítačově nezaložené lidi, čas, potřebný pro napsání testů, pravděpodobnost zanesení chyby při vytváření testu a úroveň abstrakce testu.

Zkoumání tohoto problému stále probíhá, z předběžných výsledků však lze stanovit hlavní výhody a nevýhody použití každého z těchto nástrojů. Nejefektivnějším přístupem je kombinace obou, tedy použití nástroje Behat s přidáváním vlastních vět a zdrojového kódu. Nejméně efektivní je způsob testování využitím pouze implicitních vět nástroje Behat.

Smyslem a přínosem této práce je zjištění, zda je BDD vhodný a efektivní pro společnost **Dixons Carphone** (dříve *Dixons Retail*). Poskytnutí nezávislého pohledu na tento problém, založeným na reálných příkladech a ověřených argumentech.

Klíčová slova: Automatické testování — Chováním řízený vývoj — Selenium — PHP — Behat

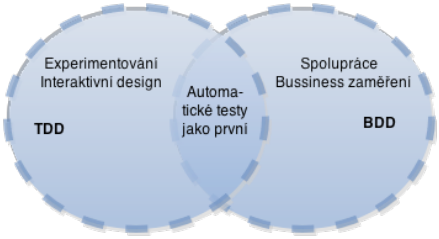
Přiložené materiály: Testovací stránka Currys — Testovací stránka PC World — Společnost Dixons Carphone

*xkudai00@stud.fit.vutbr.cz, *Fakulta Informačních Technologií, Vysoké Učení Technické v Brně*

1. Úvod

Tato práce vznikla jako reakce na použití chováním řízeného vývoje (angl. *Behavior Driven Development* dále jen BDD) firmami jak v České Republice, tak i v zahraničí. Tato ještě poměrně nová technika se v ČR teprve začíná prosazovat a zatím stále zůstává něčím ne moc prozkoumaným. Jedním z představitelů použití chováním řízeného vývoje v praxi v České Republice je firma **Dixons Carphone** (dříve *Dixons Retail*), která mi poskytla možnost se blíže seznámit s projekty založenými na BDD. Zajímalo mě, jaké jsou hlavní výhody a nevýhody použití této techniky, jaké jsou odlišnosti a mezery (pokud nějaké jsou) v tomto přístupu testování oproti klasickému přístupu.

Tato práce je zaměřená na porovnání BDD procesu s klasickým přístupem testování webových aplikací.



Obrázek 1. Podstata BDD, TDD.

Za klasický přístup je považováno použití nástroje **Selenium** při tvorbě testovacích sad. Porovnání je provedeno na testovací sadě napsané pomocí dvou nástrojů: **Behat** (BDD nástroj, jazyk – PHP) a **PHP Selenium Client od Nearsoft** (Selenium, jazyk – PHP). Následně jsou porovnány:

a) odhalené chyby
b) způsob zápisu a vykonání scénáře
c) srozumitelnost pro počítačově nezaložené lidi
d) čas, potřebný pro napsání testů
e) pravděpodobnost zanesení chyby při vytváření testu
f) úroveň abstrakce testu

Výsledky porovnání naleznete v tabulce 1 (kapitola 4).

Při pohledu do historie se dozvíme, že agilní technika BDD, vznikla jako odpověď na již existující - testy řízený vývoj (angl. *Test Driven Development*) - dále jen TDD, viz také obrázek 1.

TDD byl docela skepticky vnímán programátory. Z jejich pohledu nedávalo smysl psát testy dříve než byl napsán alespoň nějaký zdrojový kód [1]. BDD řeší tenhle problém a navíc spojuje zákazníka, QA týmy, vývojáře a business analytiky dohromady. Protože procesu vytvoření specifikace účastní každý z nich, všichni vědí o co se jedná. Následující kroky vývoje jsou přesně definovány již od začátku. Stejné platí o testovacích scénářích. Přirozené názvy testů a Gherkin syntaxe (podkapitola 2.1) se používají proto, aby jednotlivým testům a scénářům mohl porozumět jak zákazník, tak i business analytik bez nutnosti chápání zdrojového kódu testů. Podrobněji o BDD přečtete v kapitole 2.

Můj přínos spočívá v porovnání obou přístupů testování podle zmíněných dříve kritérií a) až f). Postup testování je popsán v kapitole 3. Podkapitoly 3.1 a 3.2 přibližují dva možné způsoby práce s BDD nástrojem Behat. To jsou: testování využitím pouze implicitních vět Behatu a testování s přidáváním vlastních vět a zdrojového kódu. Podkapitola 3.3 je věnována klasickému přístupu testování. Výsledné porovnání a hodnocení najdete v kapitole 4.

2. Chováním řízený vývoj

Proces chováním řízeného vývoje lze popsat následujícími kroky:

1. Zákazník popisuje business analytikovi své požadavky na výsledný systém.
2. Business analytik, vývojář a tester (někdy také zákazník) dávají požadavky na software dohromady a strukturují je do podoby scénářů.
3. Vývojář využívá tyto scénáře při vytváření SW.
4. Tester využívá tyto scénáře pro testování SW.

Tenhle postup je velice efektivní v případě, že nedošlo k žádné chybě při specifikaci produktu nebo při vytváření scénářů. Jinak následující kroky na konci mohou být zákazníkem úplně znehodnoceny. Což lze považovat za velké riziko při volbě BDD.

Pokud se podíváme na proces vytváření scénářů a testů, setkáme se s první odlišností při práci s BDD a to jsou přirozené názvy testů, které jsou obvykle reprezentovány větami. Tedy název každého testu vyjadřuje jeho účel, díky čemuž, tomu, co ten test vlastně provádí může porozumět i člověk, který nemá moc zkušeností s programováním nebo testováním. Například test, který ověřuje, zda uživatel zadal povinný údaj (jméno), by mohl mít název: `testSelzePriChybejicimJmene()` [2].

S časem autoři BDD přišli na to, že pouze přirozený název testu, nestačí pro vyjádření jeho chování a důvodu, proč ten test byl napsán. Proto vzniká další odlišnost BDD procesu od jiných a to jsou *given-when-then scénáře*, které podle zdroje [2] mají podobu:

> **Given** *some initial context (the givens),*
> **When** *an event occurs,*
> **Then** *ensure some outcomes.*

Klíčové slovo **Given** vyjadřuje počáteční podmínky pro spuštění testu. Může mít následující podobu:

```
Given I am on "example.com" page.
```

Tedy testování začíná na stránce *example.com*.

Za **When** a **Then** obvykle následuje popis události a jejího důsledku. Pokud je potřeba popsat více událostí a/nebo důsledků lze je spojit klíčovým slovem **And**. Např.:

```
When I fill in "name" with "MyName"
And I press "formSubmit"
Then I should be on"submitPage.com"
And I should see "Your name is My-
Name"
```

2.1 Gherkin

Gherkin je jazyk, který je založen na BDD principech. Je to jazyk, který využívá odsazení textu pro definici struktur. Každá věta ukončená symbolem konce řádku

se považuje za *krok*. Většina řádků napsaných v Gherkinu začíná jedním z klíčových slov: *Feature:, Scenario:, Given, When, Then, And, But, Background* atd. Testovací sady jsou rozděleny do `.feature` souborů. Obsahem těchto souborů jsou *příběhy* (angl. *stories*). Příklady příběhů najdete v kapitole 3.

Spolu s tímto jazykem vznikají různé nástroje (*Behat*, *Concordion*, *Cucumber*, *JBehave* atd.), které využívají pevně definovanou syntaxi Gherkin. Více informací o nástrojích najdete v podkapitole 2.2.

2.2 Nástroje

BDD je mostem spojující přirozený jazyk (příběhy) s automatickými testy (JUnit apod.). Specifikace testů by měla být napsána pro business analytiky ve srozumitelné formě. To stejné platí pro případy, kdy test z nějakého důvodu selže. *Člověk, který nemá zkušenosti s programováním, by tomu měl rozumět.* To je hlavním cílem BDD nástrojů - poskytnout srozumitelnost a přívětivost pro počítačově nezaložené lidi.

Behat je jedním z BDD zaměřených nástrojů, který pro vytvoření scénářů a příběhů využívá syntaxi jazyka *Gherkin* (viz podkapitola 2.1), založený na *given-when-then scénářích*. Implementačním jazykem je PHP. Příběhy se ukládají do `.feature` souborů, zdrojové kódy jsou potom ve `FeatureContext.php` [3].
Při vytváření nových vět ve scénářích nám Behat automaticky nabídne šablonu pro nový test, kde název testu odpovídá zadané větě. Základní vztah je: 1 řádek = 1 krok. Při dodržení tohoto vztahu lze jednoduše vystopovat, kde a proč došlo k chybě bez náhledu do zdrojového kódu. Tomu by potom mohl porozumět i člověk, který nemá moc společného s programováním.
Nabízí nám také možnost vytvoření příběhů v 10 různých jazycích včetně češtiny. V českém jazyce to může mít následující podobu[1]:

Scénář: *Vymazání paměti agentovi*
Pokud `existuje agent "A"`
A také `existuje agent "B"`
Když `vymažu paměť agentovi "B"`
Pak `by měl existovat agent "A"`
Ale `agent "B" by existovat neměl`

Cucumber Původně implementačním jazykem je Ruby. Při použití pluginu *Cuke4Duke Maven* od Maven je podporována Java [1] (Cucumber - JVM). Proces vytváření testů se řídí Gherkin standardy. Příběhy se ukládají do `.feature` souborů. Byl vyvinut především pro akceptační testování. Je považován za předchůdce Behatu.

Concordion Hlavní odlišností nástroje *Concordion* od ostatních nástrojů je způsob zápisu specifikací. V tomto případě se specifikace zapisuje pomocí HTML kódu a implementace samotných testů v Javě. Podobně jako *Cucumber*, podporuje hodně jiných jazyků a platforem. Aktuálně existují verze pro .NET, Python, Fantom, Scala a Ruby.
Není vázán na BDD syntaxi (Gherkin). Tester a business analytik v tomto případě mají volnost při zápisu specifikace, díky čemuž může být specifikace napsána přirozeným jazykem. Lze také do specifikace přidávat obrázky či tabulky [4].

JBehave je nástroj podporující BDD, pokročilý TDD a také akceptační - TDD (angl. *Acceptance - Test Driven Developement*) - dále jen ATDD [5]. Autorem JBehave je Dan North - jeden z autorů BDD [2]. Implementačním jazykem je Java.

Každý z nastudovaných nástrojů má své výhody a nevýhody, avšak můj výběr se zastavil na dvou nástrojích: *Cucumber* a *Behat*. Jsou si hodně podobné a stejně efektivní. Oba dva nástroje plně odpovídají Gherkin standardům, což pro mě bylo klíčovým při rozhodování o volbě nástroje. Nakonec z důvodu většího množství zkušeností s jazykem PHP než s Ruby moje volba padla na nástroj *Behat*.

3. Testovací sada

Vytvořená testovací sada je založena na scénářích z dokumentu poskytnutého firmou *Dixons Carphone*, který obsahuje kompletní specifikaci projektu *Guest Checkout* (anonymní nákup na e-shopu implementovaný na stránkách Currys a PC World) a testovací scénáře. Jako jazyk pro vytvoření příběhů byla zvolena angličtina, protože je *Dixons Carphone* anglickou firmou a celá specifikace je napsana také v tomto jazyce.

Pro dosažení cíle byla stejná testovací sada napsána několika způsoby a byla porovnána efektivita každého z přístupů. Zvolené způsoby jsou:

- Napsat testovací sadu využitím pouze implicitních vět nástroje Behat - podkapitola 3.1
- Napsat testovací sadu využitím implicitních vět nástroje Behat s přidáváním vlastních vět a zdrojového kódu - podkapitola 3.2
- Napsat testovací sadu využitím nástroje Selenium - podkapitola 3.3

3.1 Testovací sada bez přidávání vlastního zdrojového kódu

Hlavní výhodou využití implicitních vět pro tvorbu testovací sady je zjednodušení práce pro testera. Na-

[1] Příklad převzat z: bin/behat –lang=cs –story-syntax

příklad je nám k dispozici věta, která zvolí jednu z možností (option) z výběrového pole (select)[2]:

When `/^(?:|I ) select "(<option>)" from "(<select>)"`

V tomto případě, tester nemusí mít příliš hluboké znalosti programování. Důležité je, aby se dokázal orientovat v HTML kódu stránky a správně identifikovat potřebný element. V závislosti na větě, kterou použije, má možnost se odkázat na element pomocí *id, name, lable, value, css selectoru* apod.

Potom *příběh* může vypadat následovně (podle syntaxe Gherkin):

@25.21
Scenario: *Customer Eric Evans with download product and small box product in basket enters recognised email address. System displays "Welcome back Eric" and prompts for password. Customer enters wrong password. System shows invalid password message. Guest checkout option is greyed out and customer enters valid password and system progresses to delivery options screen[6].*

...
When `fill in "search" with "iphone 5c tough naked case - clear"`
Then `I press "search_bttn"`
And `I follow "CASE-MATE iPhone 5c-Tough Naked Case - Clear"`
...
And `I should see an`[3] `"div.btn.btnDisabled" element`
...

Nevýhoda tohoto přístupu vyplývá z omezeného počtu implicitních vět, které navíc neumí pokryt všechny možné stavy systému. Scénář, napsaný tímto způsobem, se odlišuje pevnými kroky a pevnou definici elementů, které při testování používá. Problém nastává v případech, kdy například potřebujeme vyhledat libovolné zboží ze seznamu. Pomocí vět lze odkázat pouze na konkretní zboží, podle názvu, id apod. (viz věta **And** `I follow "CASE-MATE iPhone 5c Tough Naked Case- Clear"`). Může ale nastat případ, kdy se toto zboží přestane prodávat. Což potom povede k selhání testu. Tedy pomocí implicitních vět neumíme říci, že potřebujeme *nějaké* zboží, které má *nějaké* vlastnosti (je skladem, lze dovést domu apod.). Umíme odkázat pouze na konkretní zboží. Z toho vyplývá, že úroveň abstrakce implicitních vět je hodně nízká.

3.2 Testovací sada s přidáváním vlastního zdrojového kódu

Důležitou výhodou přidávání vlastního kódu a vlastních vět do scénářů je urychlení napsání testovacích kroků.

[2]Seznam všech dostupných vět: `bin/behat -di`
[3]Neurčitý člen *an* je součástí implicitní věty.

Tak například proces vyplnění formuláře s osobními údaji pomocí implicitních vět lze rozepsat na 7 až 10 řádků, kde každý z nich se liší od předchozího pouze ID elementu, do kterého se zapisuje a textem, který se do tohoto elementu zadává. Níže naleznete část jednoho z takovýchto testů:

@EnterDetDave_25.29
Scenario: *Customer Dave Dawson with small box product in basket (test 25.18) selected guest checkout option. Dave enters guest details where name, delivery address and billing address are all the same as his account already holds* [6].

...
Then `I press "guest_checkout"`
And `I fill in "sFirstname" with "Dave"`
And `I fill in "sLastname" with "Dawson"`
And `I fill in "sDelPostalCode" with "AB10 1AG"`
And `I fill in "sDelAddressLine" with "address 123"`
...
Then `I press "formSubmit"`
...

Tento způsob zápisu je funkční a správný, není však efektivní. Pomocí přidání vlastního kódu a vytvoření nové věty, lze výše uvedený příklad zapsat následujícím způsobem:

@EnterDetDave_25.29
Scenario: *Customer Dave Dawson with small box product in basket (test 25.18) selected guest checkout option. Dave enters guest details where name, delivery address and billing address are all the same as his account already holds* [6].

...
Then `I press "guest_checkout"`
And `I fill form with:`

firstName	lastName	postCode	address
Dave	Dawson	AB10 1AG	address 123

Then `I press "formSubmit"`
...

Sémantika obou dvou testovacích případů zůstala stejná, avšak se změnila efektivita zápisu. Cenou za efektivitu je ale menší přehlednost při selhání testu. V případě implicitních vět je moc dobře vidět, co se nepodařilo a na jakém kroku test spadl. V druhém případě to tak jasné není. Pomoci v tomto případě může snímek obrazovky (který je třeba také ručně nastavit pomocí zdrojového kódu) nebo studování zdlouhavé výjimky. Jinak se v případě selhání zčervená celý krok **And** `I fill form with:` a nebude z toho patrné, co přesně chybu způsobilo.

3.3 Testovací sada s využitím nástroje Selenium

První problém se kterým jsem se setkala byla volba nástroje Selenium pro jazyk PHP. Oficiální verze nástroje Selenium pro tento skriptovací jazyk neexistuje.

Proto jsem musela volit jeden z několika[4] neoficiálních. První moje volba nebyla úspěšná. Nástroj **PHPUnit Selenium** nepodporoval některé funkce, které jsem potřebovala. Z tohoto důvodu jsem byla nucena hledat dále. Nakonec jsem zvolila **PHP Selenium Client** od Nearsoft.

Odlišnost práce s PHP Selenium Clientem spočívá hlavně v tom, že testerovi nejsou k dispozici žádné pomocné funkce, pokud nebudeme brát v úvahu vyhledávaní elementů apod. Což znamená, že než se tester pustí do pohodlnějšího a rychlejšího testování, musí vydělit nějaký čas na vytvoření takových funkcí (metod). Potom se proces vytvoření samotných testů v PHP Selenium Clientu (dále jen PHP SC) už moc neliší od procesu vytvoření stejných testů v Behatu, kde přidáváme vlastní kód.

Níže je příklad jedné z pomocných funkcí, kterou jsem vytvořila pro vyplnění formuláře na základě hodnot asociativního pole `$form`, ve kterém klíč je ID elementu, hodnota - hodnota, která bude vložená do políčka:

```
public function fillForm($form){
 if(is_array($form)){
  foreach ($form as $key => $value){
   if($key === "submit")
    $this->clickOnElement(By::id($value)
        );
   else
    $this->fillField(By::id($key),$value
        );
  }
 }
 else{
  throw new Exception("Array required"
      ,1);
 }
}
```

Funkce využívá také další mnou vytvořené pomocné metody `fillField()` a `clickOnElement()`. Použití výše uvedené funkce v testu vypadá následovně:

```
/* Customer with an account has
    confirmation of their guest checkout
     order */
public function test25_55(){
...
 $this->form["sFirstname"] = "Dave";
 $this->form["sLastname"] = "Dawson";
 $this->handler->fillForm($this->form);
...}
```

Vyplnění formuláře jsem prováděla třemi způsoby. Jeden z nich je pomocí asociativního pole, tedy uvedeny dříve. Druhý způsob využívá mou vlastní funkci `fillFormAs()`, která má jméno a příjmení jako parametry. Tato funkce byla vytvořena z důvodu, že virtuální zákazníci, kteří byli použiti pro testování, mají stejné osobní údaje s výjimkou jména a příjmení a také bylo třeba pokaždé vyplnit stejný formulář. Tedy v této funkci jsou ID políček a hodnoty zadány napevno. Třetí způsob je použití funkce `fillField()`, kde definujeme každé políčko zvlášť (podobně jako test @EnterDetDave_25.29 - podkapitola 3.2, kde se využívá pouze implicitní věta Behatu `I fill in "" with ""`). Poslední způsob ztrácí efektivitu v případech, kdy jedná testovací třída obsahuje několik testů, kde je třeba vyplnit nějaký formulář a kde se mění jen málo hodnot. V tomto případě místo opisování stejného kódu lze použít asociativní pole, kde změníme pouze ty hodnoty, které se liší. Navíc metoda `fillForm()` umožňuje vyplnit jakýkoliv formulář.

Stejné funkce lze bez problémů implementovat i v Behatu. To vede k závěru, že co se týká doplnění vlastního kódu do `FeatureContext.php`, moc velký rozdíl ve složitosti není patrný. Pokud ale je pro nás důležité dodržení BDD praktik (aby testům porozuměli také programátorsky nezaložení lidé), musíme navíc za každým krokem přidávat výpis toho, co se přesně provedlo. To potom lze považovat za komplikaci, která navíc znepřehledňuje zdrojový kód testů.

4. Závěr

Tato práce se zaměřuje především na rozdíly mezi BDD a klasickým přístupem testování. Důležité bylo zjistit jaké, každý z přístupů, má slabé stránky, jaký přístup je vhodnější a pro jaké účely.

Podle dosavadních výsledků a závěrů, je patrné, že záleží především na tom, na co klademe důraz při vývoji a testování SW. Každý z přístupů má své klady i zápory. Hodnocení třech přístupů testování (Behat bez přidávání vlastního zdrojového kódu – Behat-vk, Behat s přidáváním vlastního zdrojového kódu – Behat+vk, testování pomocí Selenia — PHP SC) podle níže uvedených kriterií je znázorněno v tabulce 1.

Kritéria hodnocení:

a) odhalené chyby
b) způsob zápisu a vykonání scénáře
c) srozumitelnost pro počítačově nezaložené lidi
d) čas, potřebný pro napsání testů
e) pravděpodobnost zanesení chyby při vytváření testu
f) úroveň abstrakce testu

Pokud je pro společnost důležité, aby zákazník, business analytik, tester a vývojář spolupracovali a aby každý měl přehled o tom, co se vyvíjí a v jakém je

[4] SeleniumPHP Od: Chibimagic, Lukasz Kolczynski, Facebook, Adam Goucher, Nearsoft, PHPUnit Selenium

	Behat – vk	Behat + vk	PHP SC
a)	Nešlo pokrýt všechny případy	Byly odhaleny stejné chyby	
b)	(+)přehledné kroky (-)neefektivní zápis	(-)nepřehledné kroky (+) efektivní zápis	(-) manuální výpis (+) efektivní zápis
c)	**Nízká:** při použití css selektorů	**Vysoká:** vlastní věty	**Hodně nízká:** bez výpisů kroků
d)	Rychlé vytvoření testů	+ čas na vlastní kód	+ čas na vlastní kód + pomocné funkce
e)	**Nízká:** bez vlastního kódu	**Střední:** chyby ve vlastním kódu	**Vysoká:** chyby ve vlastním kódu + pomocné funkce
f)	**Nízká:** bez vlastního kódu	**Vysoká:** použití vlastního kódu	

Tabulka 1. Hodnocení přístupů testování podle kritérií a) až f).[5]

vývoj aktuálně stadiu, BDD je dobrou volbou. Další otázka, která nastává, je do jaké míry chceme BDD v praxi využít. Podle pravidel BDD [7] se nesmí ve specifikaci a scénářích použít žádný technický název elementu, což znamená, že v případě Behatu přicházíme o možnost použít některé implicitní věty. To potom může vést k tomu, že tester bude nucen přepisovat implicitní větu do vlastní, s tím rozdílem, že vlastní věta místo technického názvu elementu bude obsahovat přirozený. Z hlediska efektivity a času tento přístup nelze považovat za nejlepší řešení. Jako druhé možné řešení je použití přirozených názvů jako identifikátorů při vytváření HTML kódu stránky. V tomto případě se vyžaduje větší spolupráce s vývojáři.

Pokud si zvolíme nekomplikovat práci testerům a vývojářům, můžeme zkombinovat BDD s klasickým přístupem tak, že necháme prostor pro technické názvy elementů. Je z tabulky 1 patrné, že kombinace implicitních vět i vlastního kódu, tedy testovací metoda **Behat + vlastní kód**, je nejefektivnější.

Volba PHP SC pro testování znamená především přenechávání vytvoření všech pomocných funkcí na testerech. To potom zvyšuje pravděpodobnost zanesení chyb při tvorbě testů.

Když se na nástroje Behat a PHP SC podíváme ze stránky podporování funkcí Selenia z oficiálních verzí, např. pro jazyk *Python*, oba dva nástroje v některých případech selhávají. Např. u Behatu (verze 2.4 stable) není možné přepínat mezi okny. PHP SC s tímto problém nemá. Komplikace ale nastávají při využití funkce návratu v historii `back()`. Nelze se totiž odkázat na dříve načtené elementy ze stránky 1 po zpětném návratu. Bez ohledu na nezměněnou cestu k elementu PHP SC považuje tuto definici za neplatnou a vyžaduje znovu načtení elementů. Což, naopak, Behatu problém nedělá.

Z důvodu stálého vývoje obou frameworků, lze narazit i na další situace, které se řeší obtížněji v Behatu na rozdíl od PHP SC i opačně. Proto, záleží na cíli, který si klademe ještě před začátkem testování. Pokud pro nás není BDD nějakým způsobem atraktivní a business hodnota není pro nás až tak důležitá, je jednodušší pro testování využít PHP SC. V opačném případě se pustit do studování Behatu.

V případě firmy *Dixons Carphone* považuji jejich volbu BDD a nástroje Behat za vhodnou, hlavně z důvodu, že primárním zaměřením této společnosti je obchod a ne vývoj SW.

Do budoucna plánuji zkoumat i další možnosti využití nástroje Behat. Chci se zaměřit na testování aplikací bez uživatelského grafického rozhraní pomocí tohoto nástroje.

[5] **Poznámka k tabulce:** Barva políčka vyjadřuje úspěšnost testovací metody. Zelená znamená vysokou úspěšnost, žlutá - průměrnou a červená - nízkou. Označení "(+)/(-)" nese význam kladu/záporu. "+" znamená "navíc".

Literatura

[1] Matt Wynne and Aslak Hellesoy. *The Cucumber Book: Behavior-Driven Development for Testers and Developers*. Pragmatic Programmers, LLC, 2012. ISBN 978-1-934356-80-7.

[2] Dan North. INTRODUCING BDD. Dostupné z: http://dannorth.net/introducing-bdd/, 2006. [vid. 2014-09-18].

[3] WWW stránky. Behat documentation. Dostupné z: http://docs.behat.org/en/v2.5/. [vid. 2014-12-12].

[4] WWW stránky. Concordion documentation. Dostupné z: http://concordion.org/. [vid. 2014-12-17].

[5] WWW stránky. Jbehave documentation. Dostupné z: http://jbehave.org/introduction.html. [vid. 2014-12-17].

[6] Mike Bleasdale. *Request For Change 2433 Guest Checkout*. Dixons Carphone, Březen 2014.

[7] Daniel Dec. Behaviour-Driven Development. Dostupné z: http://www.future-processing.pl/blog/behaviour-driven-development/. [vid. 2015-03-30].