Behaviour Driven Develompent

Od teórie k praxi

Ivan Nečas, UČO 208305, LaSArIS, 2009

Motivácia: nedostatky Test Driven Develompent

Test Driven Development (TDD) je jednou z praktík aplikovaných pri agílnom vývoji softwaru. Vychádza z pravidla XP metodiky: testovať všetko, čo by sa mohlo mohlo pokaziť. Ak túto myšlienku privedieme do extrému, dospejeme k záveru, že pokaziť sa môže všetko - základ TDD.

Vývoj podla TDD pozostáva z troch fáz:

1. vytvorenie testu,
2. napísanie samotného kódu tak, aby prešiel testom,
3. refaktoring – vylepšenie návrhu kódu bezo zmeny chovania (Fawler, 1999)

Kód vytvorený pomocou TDD má niekoľko výhod. Fakt, že všetky testy po najnovších úpravách kódu prebehli úspešne, nás môže ubezpečiť, že do systému nebola zavedená nová chyba. To urýchľuje integráciu nových zmien do systému, čo je nevyhnutné pri inkrementálnom spôsobe vývoja softwaru. Výsledky testovania je možné pomocou nástrojov *continous integration* sledovať v čase a robiť tak rôzne závery, predikcie a hodnotenia.

Navzdory výhodám, prax ukazuje, že TDD so sebou prináša aj nežiadúce vedľajšie efekty, na ktoré upozorňuje napr. Dave Astels v článku *A New Look at Test-Driven Development* (Astels, 2004). Ako jednu z hlavných nedostatkov TDD vidí jazyk (v zmysle slovnej zásoby), ktorý používa. Ako základná jednotka je používaná tzv. *unit*. Jej rozsah nie je presne definovaný, ale často odráža štruktúru samotného kódu (triedy, metódy). Programátora to vedie k písaniu testu sústredného na určitú časť kódu (testovanie metódy na daných vstupoch...) vo veľmi špecifickom kontexte. Zároveň má tendenciu testovať čiste implementačné detaily, ako napr. určitý typ premennej a pod.

Písanie testov ako záruka toho, že kód funguje správne (test = verifikácia), zdá sa, neprináša také úspechy, ako sa od neho očakáva. Jedným z dôvodov tohto problému môže byť fakt, že testovanie sa sústreďuje na to, čo daný objekt je, než na to, čo daný objekt robí (Chelimsky, 2010).

Možné riešenie: Behaviour Driven Develpment

Dave Astels vo svojom článku (Astels, 2004) naznačil možné riešenie nedostatkov, ktoré so sebou prináša TDD a opiera sa *Sapir-Whorfovu* hypotézu, že existuje prepojenie medzi jazykovými kategóriami jazyka, ktorým človek hovorí, a tým, ako daný človek vníma svet a správa sa v ňom. V našom prípade to teda znamená, že ak chceme zmeniť pohľad, ako sa pozeráme na objekty z pohľadu testovania, musíme najprv zmeniť jazyk, ktorým ich popisujeme.

Na rozdiel od xUnit-orientovaných frameworkov, BDD používa pri vyjadrovaní inú slovnú zásobu, ktorá sa snaží viesť k popisu toho, ako sa má daný objekt správať. Namiesto testovacieho prípadu (test case) používa ako základnú štruktúru *kontext*. Slovo test je nahradené slovom *should*. Namiesto overovania, že objekt sa správa podľa predpokladu (assertEquals(expected, actual)), píšeme špecifikáciu toho, ako sa má objekt správať (shouldBeEqual(actual, expected)). Cieľom týchto zmien v slovníku je teda zmeniť náš pohľad na to, akým spôsobom popisovať kód tak, aby bolo hlavné ťažisko na sústredené na špecifiká jeho správania a nie na jeho vnútornú štruktúru.

Pre ilustráciu rozdielov behaviour a test driven prístupu nám môže slúžiť jednoduchý príklad testovania/špecifikácie triedy zodpovednej za vedenie zoznamu študentov. Pri vytovrení nového zoznamu by mal byť tento zoznam prázdny (metóda empty() by mala vrátiť true). Unit test by vyzeral podobne, ako nasledovná ukážka (syntax Ruby):

```
class StudentsListTest < TestCase
     def test_empty()
          students_list = StudentsList.new
          assert_true(student_list.empty())
     end
end
```

V BDD by bola trieda StudentsList popísaná nasledujúcim spôsomon:

```
describe StudentsList do
  describe "first created" do
     it "should be empty" do
       students_list = StudentsList.new
       students_list.should be_empty
     end
  end
end
```

Výstupom tohto testu pri úspešnom behu je text, ktorý je podobný špecifikácií:

```
StudentList
     first created
          -should be empty
```

Cieľom BDD by mala byť skôr špecifikácia správania, než výsledná kontrola, že funguje. Tým sa testovanie lepšie sústreďuje na požiadavky, ktoré sú na neho kladené, s vynechaním implementačných, na výsledné fungovanie nepodstatných, detailov. Výsledkom tohto procesu by mala byť spustiteľná dokumentácia, teda textový dokument zrozumiteľný odborníkom doménovej oblasti, ktorý je zároveň možné spustiť pomocou vhodných nástrojov so všetkými výhodami, ktoré boli objavené už pri používaní TDD. Je nutné podotknúť, že BDD a TDD sú skôr podobné, ako rozdielne, a veľa osvedčených techník zostáva zachovaných. Rozdiel je hlavne v slovníku a v uhle pohľadu, ktorým sa na problematiku pozeráme.

BDD pohľad na vývojový cyklus

Postupom času sa BDD prepracúva z určitej formy TDD k plnohodnotnej metodike vývoja softwaru s vlastnou filozofiou, procesmi a nástrojmi. *The Rspec book* (Chelimsky, 2010) uvádza tri hlavné princípy, ktorými by sa mal BDD riadiť:

1. **Enough is enough** – pri špecifikácii by sme nemali robiť menej, než je potrebné na to, aby sme mohli začať, ale rovnako ani zbytočne viac. Toto sa uplatňuje aj pri automatizácii procesu vývoja, kde je vhodné automatizovať proces zostavovania a nasadenia systému, nie je však vhodné pokúšať sa automatizovať všetko.
2. **Deliver stakeholder value –** ak robíme niečo, čo priamo nesúvisí s dodaním hodnoty zákazníkovi alebo so zvýšením schopnosti dodávať novú funkcionalitu, mali by sme túto aktivitu ukončiť a sústrediť na to, čo túto hodnotu prináša.

3. **It's all about behaviour –** uvažovanie pri všetkých činnostiach, či už špecifikácii požiadaviek, alebo implementácii, by malo orientované na popis chovania.

Životný cyklus softwaru podľa BDD začína špecifikáciou funkcií, ktoré má software ponúkať, aby poskytol zákazníkovi prínos. Tento prístup sa v základe nelíši od tradičných top-down techník. Kľúčová je aplikácia bodu 1. na túto aktivitu. Fázu analýzy ukončujeme v okamihu, keď máme dostatok podkladov na to začať implementovať. Je dobré myslieť na to, že zákazník za nami prišiel, pretože primárne potreboval riešiť určitý problém. Prílišne detailná špecifikácia požiadaviek môže viesť tomu, že úspech projektu začne byť posudzovaný podľa rozpísaných požiadaviek a nie podľa toho, čo zákazník potrebuje, čo môže viesť k sklamaniu, keď zákazník po dodaní systému zistí, že jeho pôvodný problém zostáva nevyriešený.

BDD používa vo fáze analýzy požiadaviek termíny *feature, story* a *scenario. Feature* je konkrétna dodávaná funkcionalita (napr. autentizácia užívateľov). Zákazník má prvotnú predstavu o tom, čo od systému očakáva, a našou úlohou je zreálniť túto predstavu. Na popis *feature* sa používa sada *stories*, ktoré predstavujú jednotlivé funkčné celky *feature* (napr. prihlasovanie do systému, správa užívateľských účtov, riadenie prístupu...). *Story* by mal byť určitý celok nasaditeľný do produkčného prostredia, resp. predvediteľný zákazníkovi, ktorý nám k nemu dokáže poskytnúť spätnú väzbu pre ďalší vývoj. Skutočné nasadenie danej *story* je potom otázkou business rozhodnutia.

Štruktúra *story* je nasledovná:

- **Názov –** slúži na jej identifikáciu pri komunikácií.
- **Narrative (príbeh) –** Zachytenie podstaty danej *story*. Mal by obsahovať informácie o tom, ktorý užívateľ ju požaduje, jej popis a prínos, ktorý od nej užívateľ očakáva. Existuje niekoľko ustálených formátov *narrative*, ako napr. „*In order to [benefit], a [stakeholder] wants to [feature]*. Umiestnenie prínosu na prvé miesto nám pomáha uvedomovať si fakt, čo je naším hlavným cieľom.
- **Acceptance criteria –** určujú, kedy sme s prácou na *story* hotoví. Je to sada scenáre použitia danej *story*. Každý scenár pozostáva z popisu stavu systému, udalostí a očakávaných reakcií systému.

Aby sme mohli začať danú *story* implementovať, mali by sme mať o nej informácie tohto typu. Pre plánovanie je podstatná predovšetkým informácia o tom, kedy môžeme danú story uzavrieť, najlepšie pomocou automatizovaného spustenia akceptačných kritérií voči aktuálnej implementácii.

Nástroje

Aby mohla byt teória uvedená do praxe, sú potrebné nástroje, ktoré vývojárom umožnia podľa metodiky BDD vyvíjať svoje projekty. Prvé použiteľne nástroje, ktoré tento smer postupujú, sa objavili pre programovací jazyk Ruby, pravdepodobne vďaka pestrej možnosti syntaktických konštrukcií, pomocou ktorej je možné jednoducho vytvárať *domain-specific* jazyky (DSL, viz Rails). Medzi najznámejšie dostupné BDD nástroje patria frameworky *Rspec* a *Cucumber*.

RSpec je pôvodný framework vytvorený ako reakcia na článok Dava Astelsa o problémoch TDD a ich riešení. Testy vytvorené pomocou *RSpec* patria do spoločnej kategórie ako testy jednotiek, teda medzi testy naviazané priamo na zdrojový kód. Ako základná jednotka testovania nie je určitá časť zdrojového kódu, ale určitá časť správania obejtku. Pomocou kľúčového slova *describe* sú objekty vždy uvedené do určitého kontextu,. Kontexty možno do seba vnorovať a vytvárať hierarchiu. V rámci kontextu vývojár popisuje správanie, ktoré od objektu očakáva. Predpoklady sú vyjadrované pomocou príkazu *should* a sú koncipované tak, aby sa ich zápis podobal zápisu prirodzeného jazyka.

Ako príklad si môžeme uviesť popis objektu na digitálne podpisovanie *DigitalCertificate*. Ten je určený podpisovým certifikátom a heslom a má umožniť podpiosovanie dokumentov a mailov.

```
describe DigitalCertificate do
  describe "without exisisting certificate"
    before do
       @digital_certificate = DigitalCertificate.new
       @digital_certificate.file = "non-exiting-file.p12"
    end
    it "shouldn't save properly" do
       @digital_certificate.save.should equal false
    end
  end
  describe "without valid password" do
    before do
       @digital_certificate = DigitalCertificate.new
       @digital_certificate.file = "exiting-file.p12"
       @digital_certificate.password = "invalid password"
    end
    it "shouldn't save properly" do
       @digital_certificate.save.should equal false
    end
  end

  describe "with exising file and valid password" do
    before do
       @digital_certificate = DigitalCertificate.new
       @digital_certificate.file = "exiting-file.p12"
       @digital_certificate.password = "valid password"
    end
    it "should be saved properly"
       @digital_certificate.save sholud equal true
    end
    it "should sign a file" do
       signed_file = @sign_certificate.sigh_file("file_to_sign.txt")
       DigitalCertificate.signed_file?(signed_file) should equal true
    end
    it "should sign a mail" do
       signed_mail = @sign_certificate.sigh_mail("file_to_sign.eml")
       DigitalCertificate.signed_mail?(signed_mail) should equal true
    end
  end
end
```

Tento spôsob zapisu umožní, že po implementovaní požadovanej funkcionality by sme mali vidieť nasledovný výsledok testu:

```
Digital Certificate without existing certificate
     - shouldn't be saved properly
Digital Certificate without valid password
     - shouldn't be saved properly
Digital Certificate  with existing file and valid password
     - should be saved properly
     - should sign a file
     - should sign a mail
```

Ako môžeme vidieť, tento výpis je podobný špecifikácií, ktorou by sme daný objekt popísali. Namiesto toho, aby sme ako jednotku testovania vybrali napr. metódu save a pomocou *assertions* očakávali výsledky pri jednotlivých typoch vstupov, sme vždy nadefinovali určitý kontext správania a v ňom popísali, ako sa má objekt a jeho metódy správať. Podobnosť so špecifikáciou nie je náhodná. Práve táto skutočnosť nám uľahčuje písanie testov pred implementáciou. Pri písaní štandardných unit testov je často očakávané, že poznáme štruktúru kódu, ktorý však ešte nie je naimplementovaný. Tým sa dostávame k problému sliepky a vajca: už počas vytvárania testu robíme rozhodnutia o tom, aké bude mať objekt metódy, a teda de facto implementujeme.

Okrem samotnej štruktúry prináša *RSpec* aj nástroje na vytváranie tzv. s*tubs (mocks),* teda určitých náhrad za skutočné implementácie. Tomuto riešeniu pomáha zase z velkej časti dynamická povaha jazyku Ruby, ktorý jednoducho umožňuje definovať metódy za behu. Príkladom vytvorenia takejto náhrady:

```
customer = Stub.new
customer.stub(:name).and_return(“Peter”)
```

Objektu *customer* sme týmto príkazom vytvorili metódu *name,* ktorá pri jej volaní vráti hodnotu „Peter“. Takto vytvorený objekt môžeme použiť pri komunikácií s objektom, na ktorého chovanie sa sústreďujeme. Je možné taktiež definovať, že chceme, aby testovaný objekt použil určitú metódu *stubu* takto:

```
customer.should_receive(:name).and_return(“Peter”)
```

Táto konštrukcia je však odporúčaná len pre prípady, kde explicitne požadujeme použitie danej metódy (napr. zaznamenanie určitej činnosti do logu aplikácie). V ostatných prípadoch by to znamenalo zbytočné prepojenie so samotnou implementáciou.

Zatiaľ čo *RSpec* pokrýva hlavne testovanie kódu na nižšej úrovni, na písanie integračných testov štýlom BDD slúži *Cucumber*. Jeho cieľom je poskytnúť vývojárovi možnosť písať dokumentáciu zrozumiteľnú zákazníkovi, ktorá je zároveň použiteľná na testovanie správnej funkčnosti. Okrem Ruby je dostupný aj pre ďalšie programovacie jazyky, ako Java, či jazyky z rodiny .NET.

Testovací dokument pre Cucumber má podobnú štruktúru ako *story*, o ktorej je písané v úvodnej časti tohto dokumentu. Obsahuje názov, jednoduchý popis cieľov a akceptačné kritéria. Tieto kritériá sú písané formou scenárov zložených z krokov (*steps)*, čo môžu byť predpoklady, udalosti a očakávania. Typ kroku určuje začiatočné slovo kroku (*given*, *when*, *then*). Príklad Cucumber dokumentu s ukážkou scenára testujúceho prihlasovaciu obrazovku webovej aplikácie môže vyzerať nasledovne:

```
Signing in to the system.

In order to protect sensitive data every user should be prompted to
insert his credentials to log in to the system.

Scenario: a user knows his login and password
Given user with login “test” and password “test”
When I am on the home page
And when I fill “login” with “test”
And when I fill “password” with “test”
And when I press “log in”
Then I should see “User logged in successfully”
```

Text napísaný v takomto formáte dokáže Cucumber spracovať pomocou tzv. *step definitions*, čo je presná definícia, čo sa v danom kroku deje. Výber správnej definície je založený na regulárnych výrazoch a umožňuje v kroku definovať premenné. Definícia kroku

„`Given user with login “test” and password “test”`
by mohla vyzerať nasledovne:

```
Given /^user with login “(.*)” and password “(.*)”$/ do |login, pwd|
     User.create(:login => login, :password => pwd)
end
```

Písanie scenárov je vcelku jednoduché hlavne vďaka veľkému množstvu knižníc s preddefinovanými definíciami krokov. Existujú napr. definície pre nástroj *Webrat* (http://github.com/brynary/webrat), ktorý slúži na testovanie webových aplikácií. V predchádzajúcom prípade ide napr. o kroky pracujúce s formulárom.

Pri integračnom testovaní webovej aplikácie máme niekoľko možností, akým spôsobom testovať:

1. **Testovanie vnútornej štruktúry** – pristupujeme priamo k objektom a metódam aplikácie.
2. **Simulácia webového prehliadača –** napodobňovanie správanie prehliadača, posielanie požiadavkov a testovanie odpovedí. Používa metódu čiernej skrinky. Použiteľný hlavne v prípade, že aplikácia nepoužíva pri svojej práci *javascript* a ostatné nástroje webových prehliadačov.
3. **Automatizácia webového prehliadača –** testovanie prebieha priamo vo webovom prehliadači. S využitím zásuvného modulu prehliadača je simulovaná ľudská činnosť. Pomocou tohto spôsobu je možné odtestovať aj stránky používajúce *javascript*. Nevýhodou je pomalšie a výkonovo náročnejšie spúšťanie týchto testov.

Výhodou frameworku *Cucumber* v spojení s nástrojom *Webrat* je fakt, že výber medzi 2. a 3. možnosťou je z veľkej časti transparentný a scenáre fungujú pri oboch spôsoboch spúšťania testov. Vývojár tak môže určiť, ktoré testy majú používať simulovaný a ktoré automatizovaný prehliadač, a toto rozhodnutie časom zmeniť.

Nástroje *Cucumber* alebo *RSpec* je možné používať priamo z príkazového riadku. Oveľa komfortnejšie je však integrácia v IDE a nástrojoch pre *continuous integration*. Našťastie stále viac výrobcov si uvedomuje silu týchto nástrojov a ponúka tieto funkcie vo svojich vývojárskych produktoch.

Záver

V prvom rade je nutné uviesť, že iniciatíva BDD v žiadnom prípade nezavrhuje techniku TDD. Iba poukazuje na nedostatky, ktoré môže táto technika prinášať, a ponúka možné riešenie. Z veľkej časti z TDD vychádza a stavia na jej základoch. Prínos BDD by mohol byť v pridaní ďalšej motivácie, prečo je písanie automatizovaných testov také dôležité – špecifikácia funkcií. Pri testovaní sa vývojár často stretáva so situáciou, že testovanie považuje za stratu času a má tendenciu sa mu vyhýbať. Vedenie projektu môže taktiež opomínať dôležitosť testovania, hlavne v spojení s blížiacimi sa termínmi odovzdania projektu, kedy je testovanie jedna z prvých činností, ktoré sú vynechávané z každodenných aktivít. Verím, že tieto nové postupy nájdu svoje uplatnenie a budú viesť k lepším produktom, príjemnejšej atmosfére pri vývoji a k potešeniu všetkých, ktorí sa tieto postupy rozhodnú používať.

Bibliografia

ASTELS, Dave. *A New Look at Test-Driven Development* [online]. 2004. [cit. 2009-12-11]. Dostupný z WWW <http://techblog.daveastels.com/files/BDD_Intro.pdf>.

Cucumber [počítačový program, online]. Version 0.5.1. Aslak Hellesøy, 2009 [cit. 2009-12-12]. Dostupný z WWW <http://cukes.info>

FAWLER, Martin. *Refactoring: Improving the Desing of Existing Code.* Toronto: Addison-Wesley Professional, 1999. ISBN 0-201485-67-2.

CHELIMSKY, David, et al. *The Rspec book.* Dallas: The Pragmatic Bookshelf, 2010. ISBN 1-934356-37-9.

RSpec [počítačový program, online]. Version 1.2.9. David Chelimsky, 2009 [cit. 2009-12-12]. Dostupný z WWW <http://rspec.info>.

Webrat [počítačový program, online] Version 0.6.0. Bryan Helmkamp, 2009 [cit. 2009-12-12]. Dostupný z WWW < http://github.com/brynary/webrat >