UDC: 004.415.052.3 Info M str. 24-31

BUDUĆNOST I IZAZOVI BDD-A
FUTURE AND CHALLENGES OF BDD

Milutin Cvetković, Slađan Babarogić

REZIME: *Behavior-Driven Development* (BDD) predstavlja agilni proces softverskog razvoja. Hronološki i evolutivno dolazi kao nastavka *Test-Driven* razvoja i, kao takav, unosi određene inovacije koje za cilj imaju bolju komunikaciju i razumevanje samog projekta od početka, ali i veći broj sagovornika koji ne obuhvata samo tehnička lica odnosno IT predstavnike. Iako postoje dobre prakse, poredeći ga sa drugim procesima razvoja ali i shvatajući neke njegove nedostatke, BDD nastavlja da se menja i prilagođava. Tema ovog rada je, pored kreiranja detaljnog opisa i suštine kao i poređenja sa drugim uticajnim tehnikama i procesima savremenog softverskog inženjeringa, da ukaže i na potencijalne smernice daljeg usavršavanja. Suština jeste u pravljenju i isporučivanju boljih softverskih rešena takvih da donose pravu vrednost svojim korisnicima - naručiocima i krajnjim klijentima, ali i organizaciji koja ih implementira. Pravljenje pravog softvera na pravi način je bazirano na komunikaciji i razumevanju glavnih razloga traženja rešenja. BDD unosi metodologiju, struktuiranost i formalizam, ali i automatizaciju u procesu komunikacije i implementacije.

KLJUČNE REČI: behavior-driven, test-driven, agilni razvoj softvera, izvršna specifikacija, živa dokumentacija, BDD, TDD

ABSTRACT: Behavior-Driven Development (BDD) is an agile software development process. Chronologically and evolutionarily, it comes as a continuation of Test-Driven development and, as such, introduces certain innovations that aim at better communication and understanding of the project from the beginning, but also a larger number of interlocutors that includes not only technical persons or IT representatives. Although there are good practices, comparing it with other development processes but also understanding some of its shortcomings, BDD continues to change and adapt. The topic of this paper is, in addition to creating a detailed description and essence, as well as comparisons with other influential techniques and processes of modern software engineering, to point out potential guidelines for further improvement. The essence is in creating and delivering better software solutions that bring real value to their users - customers and end customers, but also to the organization that implements them. Creating the right software in the right way is based on communication and understanding the main reasons for looking for a solution. BDD introduces methodology, structure and formalism, but also automation in the process of communication and implementation.

KEY WORDS: behavior-driven, test-driven, agile software development, executable specification, living documentation, BDD, TDD

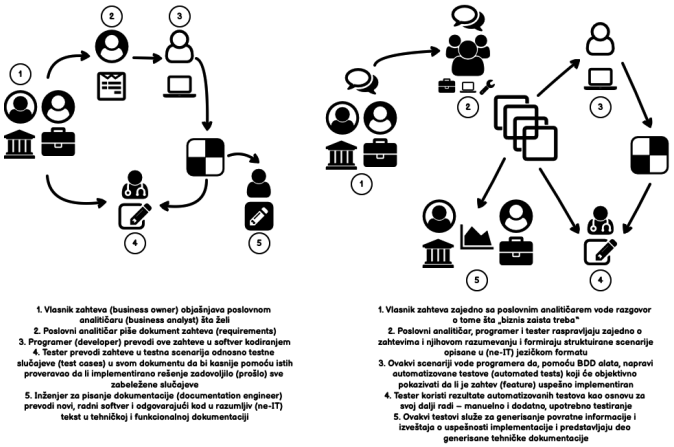
1. UVOD

Dan North je prvi formulisao koncept BDD-a. Početna ideja je bila da adresira postojeće probleme sa *Test-Driven Development*-om (TDD) koji je i sam primenjivao i predavao. Martin Fowler, jedan od potpisnika agilnog manifesta („Agile Manifesto“) i pisac knjige „Refactoring“ (Addison-Wesley Professional, 1999), kao kolega u kompaniji ThoughtWorks, imao je velikog uticaja na Dana. Sintaksnim formalizmom menja reči iz „test“ u „ponašanje“ (eng. „behavior“), piše softver kako bi proverio u praksi svoj pristup (JBehave *framework*) i praktično čini razumljivijim kod „ne-IT“ subjektima. Na Božić 2003. godine registruje jbehave.org domen. Sledeće godine, zajedno sa poslovnim analitičarem Chris Matts-om, razvija *Given-When-Then* rečnik i terminologiju za opisivanje poslovnih scenarija. U magazinu Better Software, 2006. godine Dan North objavljuje članak „Introducing BDD“ u kome objašnjava razloge i načine kojima je praktično uveo novu tehniku a na osnovama TDD-a.

Dodatno razvija *story-level* BDD *framework* za programski jezik Ruby a nazvan RBehave, koji je praktično (sa primerima) integrisan u RSpec projekat. Zajedno sa David Chelimsky-jem kao glavnim autorom, Aslak Hellesoy-jem i ostalim saradnicima pišu knjigu „The RSpec Book: Behaviour Driven Development with RSpec, Cucumber, and Friends“ (Pragmatic Press, 2010).

Tokom intervjua sa Danom na GOTO konferenciji 2013-te godine, Liz Keogh ističe dve bitne stvari – BDD koristi primere da opiše ponašanje aplikacije kao i da je BDD zapravo vođenje konverzacije o tim primerima [1].

Tradicionalni razvojni proces softvera uvodi mogućnosti za nerazumevanje i pogrešnu komunikaciju na više mesta, praktično u svakoj svojoj fazi. Bez obzira da li je uvođenje nove funkcionalnosti na postojeće softversko rešenje ili definisanje jednog funkcionalnog zahteva, svodi se na sliku u nastavku.



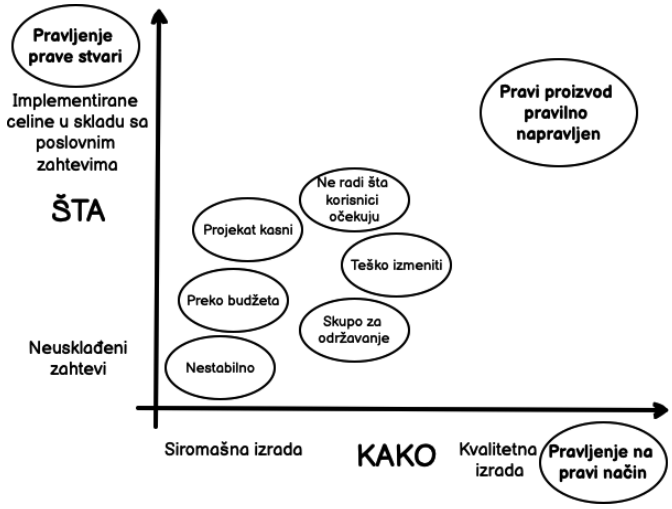
Slika 1 – *Tradicionalni Vs. BDD pristup*

U slučaju BDD-a, ceo proces se oslanja na direktnu i konstantu komunikaciju između svih učesnika, čime se pozitivno utiče na mogućnost gubljenja informacija ili nerazumevanje.

Softverski projekti ne uspevaju iz mnogo razloga, ali su dva najznačajnija:

1. Softverska rešenja se ne prave na pravi način
2. Ne prave se prava softverska rešenja

To je lepo ilustrovano na sledećoj slici:



Slika 2 – *Ilustracija različitih tipova projekata i proizvoda [2]*

Na projektu, kao i u realnom životu, na početku se zna vrlo malo. Kako vreme prolazi, nove informacije dolaze i razumevanje raste. Neke zahteve treba modifikovati ili izbaciti. Kod velikih projekata, ideja nije da se smanji nesigurnost i nepoznanice tako što potroši dosta vremena na samom početku a da bi se što više zahteva prepoznalo i definisalo. BDD nudi tehnike za „blagovremeno" upravljanje nepoznanicama i smanjivanje rizika zbog dinamičnosti takvog procesa odnosno „odbijanja da na startu prepoznate sve zahteve". Nepoznanice se, u ovoj varijanti, otkrivaju progresivno u šta se ubraja i definisanje i implementiranje zahteva koji jasno definišu poslovnu vrednost koju potpomažu ili donose.

Iako ne postoji jedinstven recept za pravljenje softvera visokog kvaliteta, koji se lako održava i proširuje, studije su pokazale da je procenat uspešnosti veći kod projekata izvedenih po uticajem *Lean* i *Agile* praksi.

2. BDD – OUTSIDE IN RAZMIŠLJANJE

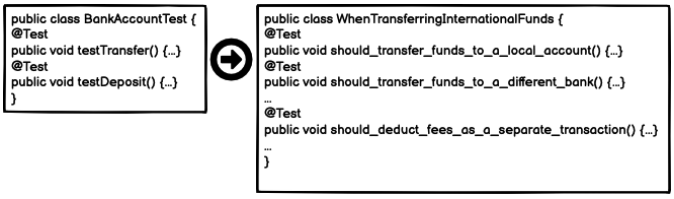
BDD je originalno zamišljen i dizajniran kao poboljšanje *Test-Driven Development*-a.

Kent Beck je u povoju agilne metodologije, osmislio TDD koji koristi unit testove da specificira, dizajnira i verifikuje aplikacijski kod. Glavni proces se sastoji iz tri dela:

1. Prvo se napiše test koji ne prolazi (*failing test*) koji opisuje zahtev (*feature*) koje treba implementirati.
2. Zatim se napiše minimalno dovoljno koda da bi test prošao.
3. Na kraju se izanalizira kod i eventualno poboljša a kako bi se lakše održavao.

Ova jednostavna ali jako efikasna tehnika omogućava i ohrabruje softverske inženjere da pišu čistiji i bolje dizajniran aplikacijski kod koji se lakše održava i ima manje defekata. Posledica tehničke implementacije *unit* testova jeste činjenica da su oni vezani (mapirani) za metodu ili funkciju koju proveravaju a ne za ono šta bi pripadajući kod trebalo da podrži u poslovnom smislu. Samim tim, kada bi se radila revizija koda (eng. *refactoring*) ili otklanjala greška u implementaciji, zbog svoje „nedeskriptivnosti" teže je ispraviti same testove ili zameniti sva mesta (uključujući i imena) gde je ispravka nužna.

North je primetio da imenovanje *unit* testova pomoću deskriptivnih rečenica i korišćenjem reči „should" (trebalo bi) doprinosi čitljivosti, razumevanju, efikasnijem pisanju i ispravljanju koda *unit* testova.



Slika 3 – *Primer uvođenja reči „should" [1].*

Testovi pisani na prethodno pomenuti način se čitaju više kao specifikacija nego kao „suvi" *unit* testovi i služe da opišu i verifikuju određenu vrstu (slučaj) **ponašanja**. Takođe je primetio da reč „test" u svojim predavanjima na temu TDD-a dosta zbunjuje sagovornike i učenike, a da su standardna pitanja vezana za TDD polaznike počela da nestaju kada je upotreba pomenute reči zamenjena rečju „behaviour". Kao posledica, počeo je i sa izbacivanjem reči „test" iz template-a testnih metoda.

2.1. GHERKIN

Zajedno sa svojim kolegom, poslovnim analitičarem Chris Matts-om, Dan North pokušava da implementira jezik koji će Chris-ove kolege moći da koriste za definisanje zahteva (opisivanje ponašanja sistema), a koji se lako može transformisati u automatizovane kriterijume za prihvatanje softvera (acceptance tests).

Počeli su od *story template*-a u kompaniji koji su koristili:

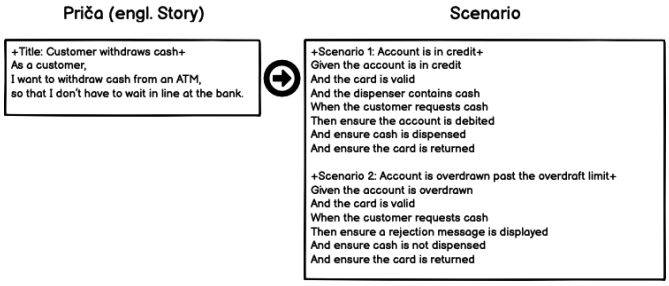
***As a** [X]*
***I want** [Y]*
***so that** [Z]*

gde je X – uloga (rola), Y – feature, a Z – benefit ili vrednost tog *feature*-a. *Given-When-Then* format je pomirio razumljivost i upotrebu od strane ne-IT lica (poslovnih analitičara i *stakeholder*-a) sa jedne strane i, struktuiranost i razbijanje priča na fragmente koja se može automatizovati dalje a za potrebe programera i testera sa druge strane.

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

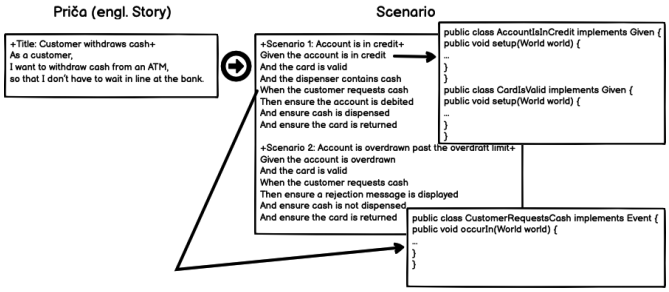
Ponašanje priče (*story*-ja) jeste kriterijum prihvatanja te priče.

Da bi sproveli ovaj zaključak u delo, počeli su da deskriptivno izražavaju *acceptance* test kriterijume za korisničke priče (eng. *user stories*) poznate kao **scenarije**.



Slika 4 – *Primer kreiranja scenarija iz priče [1]*

Scenariji se najčešće čuvaju kao ".feature" datoteke. Delovi scenarija ***givens***, ***events*** (when) i (*then*) mogu se direktno prikazati u softverskom kodu. JBehave definiše objektni model koji omogućava mapiranje delova scenarija u ***Java klase***.



Slika 5 – *Primer mapiranja scenarija u Java klase [1]*

Ovako izmapirani klase se mogu ponovo iskoristiti kod drugih scenarija i priča.

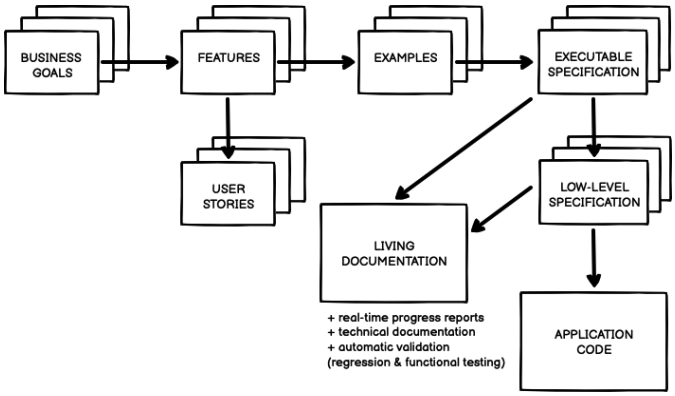
Prethodno navedena forma opisivanja je prihvaćena kao Gherkin. Ona se dalje može lako iskoristiti za pravolinijsko pisanje (generisanje) automatizovanih kriterijuma za prihvatanje softvera sa automatizovanim izvršavanjem, kad god je potrebno.

2.3. PRINCIPI I PRAKSE

BDD se danas koristi u mnogim uspešnim organizacijama i kompanijama svih veličina. Gojko Adžić je u svojoj knjizi „Specification by Example" naveo preko 50 studija slučaja odnosno organizacija [3].

Jedno od pogrešnih shvatanja jeste da je TDD pa samim tim i BDD još jedan od pristupa testiranju softvera. Testovi su koristan „nus-proizvod", ali pre svega, mera kontrole obima i kvaliteta koda. TDD je prvi značajno napravio vezu između *test-first* programiranja i dizajnerskog pristupa kroz kontinualno refaktorisanje koda. Efekat jeste da se može sa sigurnošću manje dizajnirati naspram *over-engineering*-a, odnosno, da se kod isporuke ima taman onoliko dizajna i odgovarajućeg koda koji odgovaraju trenutnim zahtevima.

BDD je holistički pristup razvoju softvera sa namerom da se optimizuje razvojni proces i uštedi razne resurse bazirajući svoju logiku na nedvosmislenim poslovnim ciljevima i vrednostima koje razvoj direktno pomaže. Na Slici 6 se mogu videti glavne aktivnosti i proizvodi (artefakti) BDD-a.



Slika 6 – *Glavne aktivnosti i proizvodi BDD-a [2]*

Polazna tačka jeste u identifikaciji poslovnih ciljeva (eng. *business goals*) i prepoznavanju koje funkcionalnosti treba implementirati (eng. *features*) a kako bi se direktno zadovoljili ti ciljevi. Kolaboracija sa korisnikom ali i glavnim *stakeholder*-ima (naručioc, vlasnik proizvoda – interno ili neko ko odlučuje) u smislu komunikacije i proveravanja trenutnog razumevanja ciljeva i zahteva, se vrši preko konkretnih primera koji ilustruju zahteve. Kad god je moguće, ovi primeri se automatizuju u formi izvršne specifikacije (eng. *executable specifications*). I primeri i izvršna specifikacija validiraju implementaciju koda i automatski nadograđuju tehničku i funkcionalnu dokumentaciju. Ista logika pomaže i programerima za pisanje kvalitetnijeg, bolje testiranog i dokumentovanog koda koji je lakše koristiti i održavati.

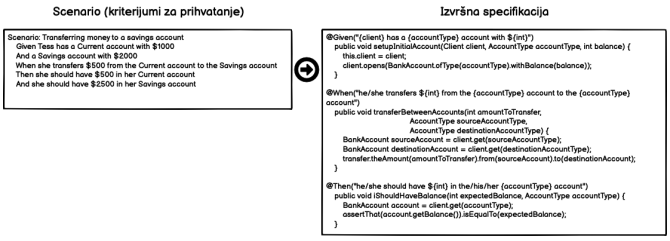
U knjizi „BDD in Action", John Ferguson Smart ističe sledeće principe:

1. Fokusirati se na funkcionalnosti koji donose poslovnu vrednost
2. Zajednički rad na specifikaciji funkcionalnosti
3. Prihvatiti nepoznanice
4. Ilustrovati zahteve konkretnim primerima
5. Ne pisati automatizovane testove već izvršnu specifikaciju
6. Ne pisati *unit* test-ove već specifikaciju niskog nivoa (eng. *low-level specification*)
7. Isporučiti „živu" dokumentaciju (eng. *living documentation*)
8. Koristiti *living documentation* kako bi se podržali tekući poslovi održavanja softvera [2]

Priče (eng. *stories*) i primeri (eng. *examples*) čine osnovu specifikacije koju *developer* koristi za implementaciju. Scenariji nastali iz priča, služe kako za kriterijume prihvatanja implementacionih celina, tako i kao smernica softverskim inženjerima šta treba da se napravi. Gde god je moguće, pretvaraju se u automatizovane testove za prihvatanje (eng. *acceptance tests*) odnosno izvršne specifikacije (eng. *executable specifications*). Na taj način, kriterijumi za prihvatanje postaju izvršni.

Ovo je generalni princip, a kako to izgleda u slučaju određenog scenarija sa konkretnim vrednostima (prebacivanje novca sa jednog na drugi račun), prikazano je u nastavku [4].

Kriterijumi za prihvatanje (eng. *acceptance criteria*) dobijaju konkretne vrednosti.



Slika 7 – *Primer izvršne specifikacije [2]*

Kao što se vidi, svakom koraku iz scenarija (*Given-When-Then* sa ili bez *And*-ova) :

Given Tess has a Current account with $1000 odgovara kod iz izvršnih specifikacija i to:

1. Deo za mapiranje (koji je korak u pitanju i koji su parametri)

   @Given("{client} has a {accountType} account with ${int}")

   public void setupInitialAccount(Client client, AccountType accountType, int balance) {

2. Deo za pozivanje aplikacijskog koda (sa odgovarajućim, prenetim parametrima)

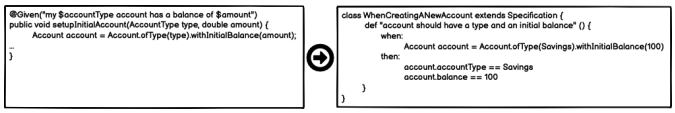
   this.client = client;

   client.opens(BankAccount.ofType(accountType).withBalance(balance));

Na ovaj način sve je predstavljeno uobičajenim poslovnim jezikom (sintaksom) koju ceo tim razume. Pozitivna je još jedna stvar – ako se zahtevi promene, izvršne specifikacije se direktno update-uju na jednom mestu.

Izvršne specifikacije predstavljaju specifikacije visokog nivoa (eng. *high-level specifications*). Najpopularniji BDD alati kojima se kreiraju executable specifications su: JBehave, Cucumber i SpecFlow. BDD koristi outside-in pristup. Kada se programira implementaciona celina (eng. *feature*), polazi se od kriterijuma za prihvatanje (eng. *acceptance criteria*) i ide se "na dole" u smislu da se kodira sve što je potrebno da se kriterijumi zadovolje. Kao što se ni jedan *feature* ne implementira ukoliko ne doprinosi identifikovanom poslovnom cilju, tako se ni ne kodira ništa ako ne pomaže da se prođe *acceptance* test. Za razliku od TDD-a, programeri neće razmišljati o kreiranju *unit* testa za određenu klasu, već o kreiranju tehničke specifikacije koja opisuje kako aplikacija treba da se ponaša, tj. šta da vrati za zadate unose ili kako da odreaguje u datoj situaciji.

Specifikacija niskog nivoa (eng. *low-level executable specification*) proističe iz specifikacije visokog nivoa i praktično pomaže sofverskim inženjerima da dizajniraju i dokumentuju aplikacijski kod u kontekstu isporučivanja *high-level feature*-a.



Slika 8 – *Primer specifikacija niskog nivoa [2]*

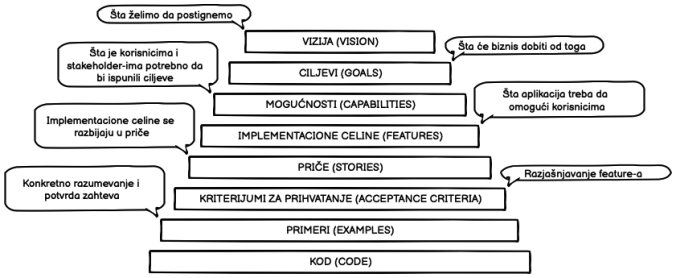
Sve ovo podseća na pisanje detaljne dizajn dokumentacije sa puno primera. *Low-level executable specifications* se mogu pisati pomoću *unit*-testing alata (JUnit ili NUnit) ili BDD specijalizovanih alata (Spock ili RSpec).

Kada se *executable specifications* izvrše, nastaje *living documentation*. Generisani izveštaji nisu samo tehničke prirode i za programere. Obe vrste izvršne specifikacije prave određenu formu dokumentovanja proizvoda i nastali izveštaji se mogu koristiti od strane celog tima (vlasnik proizvoda, menadžer projekta, poslovni analitičar, tester, inženjer) kao i samog korisnika. Lako se čitaju i koriste i imaju široku primenu – od nadgledanja progresa do načina implementacije. Živa dokumentacija (eng. *living documentation*) je uvek najsvežija (eng. *up-to-date*). Studije su pokazale da 40% do 80% troškova softvera idu na održavanje istog. U mnogim organizacijama posle puštanja u produkciju, softversko rešenje i prateća dokumentacija se predaju drugim timovima. Izvršna specifikacija visokog nivoa pomaže da se shvate poslovni ciljevi i procesi. Izvršna specifikacija niskog nivoa na *unit-testing* nivou obezbeđuje radne primere za način na koji su zahtevi implementirani.

Postojanje i korišćenje izvršnih specifikacija nije magično rešenje koje rešava sve probleme tradicionalnog tehničkog dokumentovanja. Kod dodatnog kodiranja, često su potrebni i drugi tehnički, arhitekturalni i funkcionalni dokument.

Poslovni analitičari će pomoći razvojnom timu da implementira „prave stvari", ako utvrdi sledeće:

1. Zašto se softversko rešenje pravi (vizija projekta)?
2. Kako je projekat koristan organizaciji (koji su poslovni ciljevi)?
3. Ko su *stakaholder*-i i kako projekat utiče na njih?
4. Kako će im projekat omogućiti da ostvare poslovne ciljeve što efikasnije?



Slika 9 – *BDD artefakti [2]*

Tehnike koje su opisane u nastavku, pomažu utvrđivanju pomenutih stavki.

„**Feature Injection**" tehnika služi da se izbaci minimalan set *feature*-a koji omogućavaju najviše benefita stakeholderima u smislu postizanja poslovnih ciljeva [5]. To je proces iz tri koraka: „Lovi" vrednost (eng. *Hunt the Value*) - Ubaci *fea-*

ture-e (eng. *Inject the Features*) - Uoči primere (eng. *Spot the Examples*). Suprotno od vodopad (eng. *waterfall*) pristupa gde se pravi lista zahteva i implementacionih celina (*feature*-a) na startu, ovde se prvo utvrđuje šta se tačno od sistema očekuje i koje poslovne vrednosti treba da donese, a onda se identifikuju *feature*-i (minimalan set) koji to najbolje ostvaruju.

Primer:

***Cilj**: povećanje prodaje karata*

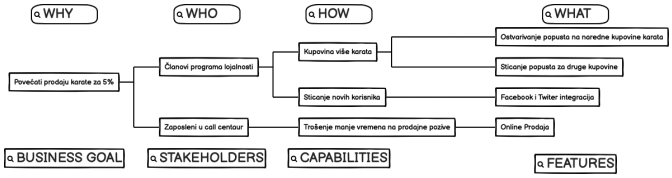
***Featue 1**:Povećanje prodaje uvođenjem programa lojalnosti koji omogućava dodatne popuste na naredne kupovine*

***Feature 2**:Povećanje prodaje uvođenjem online prodaje koja uključuje i rezervacije*

Ubacivanjem *feature*-a u smislu kratkog opisa, dobija se samo delimičan „osećaj" kako bi to izgledalo/radilo. Primeri su intuitivan način da se postigne razumevanje i usaglašavanje svih strana, dodatno ispitaju nepoznanice ali i dobije bolja procena obima posla.

„**Impact Mapping**" - u istoimenoj knjizi, Gojko Adžić opisuje interesantan, vizualan pristup usaglašavanja *feature*-a sa poslovnim ciljevima [6]. Impact Mapping je način vizualizacije relacija između ciljeva projekta, učesnika (*stakeholder*-a) i *feature*-a koji omogućavaju projektu da donese željene rezultate. Serija razgovora se vodi na relaciji *stakholder*-i – razvojni tim, a dijalozi gravitiraju oko četiri pitanja: *Why* – Zašto pravimo softver?; *Who* – Ko su sve lica (*stakeholder*-i) na koje projekat ima uticaj?; *How* – Kako *stakeholder*-i (uključujući i krajne korisnike) doprinose ostvarenju poslovnih ciljeva?; *What* – Šta aplikacija treba da ima (eng. *high-level feature*-e) da bi imala uticaje definisane prethodnim pitanjima? U agilnim praksama, *What* pitanje daje kandidate za *high-level* priče (eng. *stories*) negde klasifikovane kao *epics*. Kod *Unified Process* metodologije, ovo je ekvivalentno slučajevima korišćenja (eng. *use-cases*).

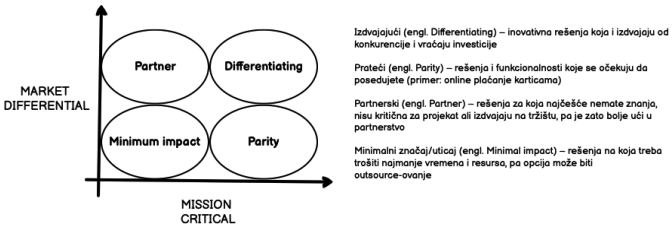
Primer je dat u nastavku.



Slika 10 – *Impact Mapping* primer [2]

„**Purpose-Based Alignment Model**" se bazira na činjenici da svi zahtevi i implementacione celine nisu iste težine. Razvijanje nekih iziskuje inovacije i nova znanja kojima će se organizacija razlikovati od konkurencije. Razvijanje ostalih predstavlja standardne tj. postojeće funkcionalnosti koji mnogi imaju. Prepoznavanje gde ulagati resurse, ne donosi samo poslovnu vrednost nego i povratak investicija (ROI). U knjizi „Stand Back and Deliver: Accelerating Business Agility" detaljno je opisan Purpose-Based Alignment model, koji koristi dva kriterijuma:

1. Kritikalnost poduhvata (eng. *mission critical*) – koliko je zahtev nasušanj/bitan i uticajan za/na poslovanje
2. Izdvajanje na tržištu (eng. *market differentiation*) – da li implementiranje zahteva donosi značajno drugačiji položaj u odnosu na konkurenciju [7].



Slika 11 – *Purposed-Based Alignment Model [2]*

Princip „**Realnih opcija (eng. real options)**" - Chris Matts je pre 15-tak godina identifikovao fundamentalni princip agilnih praksi: odlaganje odluka do „poslednjeg odgovornog trenutka" [8]. Suština jeste da se ne obavezuje dok god se ne mora. Bazira se na tri činjenice:

1. Opcije imaju vrednost
2. Opcije imaju rok trajanja (ističu)
3. Nikada se ne obavezuj rano osim ako znaš zašto

Opcije se ne mogu držati otvorene zauvek. Kod softverskog inženjeringa opcije su vezane za zahteve koji i kada treba implementirati, ali i način kako se implementira. Dakle, jedna opcija implementacije ima svoju cenu u vremenu i novcu, druga ima drugu itd. U slučaju prihvaćenih zahteva, opcije ističu sa poslednjim trenutkom da se započne implementiranje na određeni način a zbog dužine trajanje takve implementacije i unapred definisanog roka za isporuku. U Realnim Opcijama, ovaj treći princip znači odlaganje obavezivanja nekom određenom načinu implementiranja dok se ne zna dovoljno o tome zašto bi se izabrao baš taj način. Donošenje odluke se odlaže, ali ne sistematično i konstantno do „poslednjeg mogućeg trenutka". Ono se pomera do trenutka kada se ima dovoljno informacija da se deluje. Kada se pribave, deluje se što je pre moguće i brže.

Princip „**Namernog otkrivanja (eng. *deliberate discovery*)**" - Originalno predloženo od strane Dan North-a a razvijeno od Liz Keogh, predstavlja naličje principu „Realnih opcija" ali su komplementni [9]. U softverkom razvoju neznanje je ograničenje. Na kraju projekta zna se mnogo više, ali to znanje je neupotrebljivo na završenom projektu. Naučene lekcije (eng. *lesson learned*) služe za buduće projekte. Ovaj princip počinje od pretpostavke da postoje stvari koje se ne znaju. One mogu biti loše a koje nisu mogle biti anticipirane i koje će „iskočiti" u toku projekta, ali i prilike i šanse koje će biti propuštene. „Realne opcije" drže opcije otvorenim dok se ne saznaju informacije. „Namerno otkrivanje" pomaže u pronalaženju tih informacija. Na taj način se smanjuju rizici i donose odluke brže (ranije). Pretpostavimo da se implementira određeni *feature*-a koji se može razložiti na mnogo priča (eng. *stories*). Neke su pravolinijske za implementaciju dok druge nisu. Prirodno je da se krene sa onim što se zna i što je pravolinijski. Međutim, *Deliberate Discovery* zagovara smanjivanje rizika od neizvesnosti i potencijalnog neznanja, što u ovom slučaju znači da se prvo razmotre i procene najsloženije priče za implementaciju – one koje nose najviše neodgovorenih pitanja.

Konverzacije mogu imati različite forme. Standardna forma karakteristična za organizacije koje počinju da primenjuju BDD praksu jeste radionica (eng. *workshop*). Tu su pristuni svi članovi razvojnog tima i *stakeholder*-i. Radionica je dobar način da se počne što pre sa konverzacijom i da se postigne zajedničko razumevanje zahteva i funkcionalnosti koje se implementiraju, kao i da se formira skup visoko kvalitetnih primera. Loša strane je što se teško organizuje i što je skupa u smislu čovek-sati. Mnogo lakša,brža i efikasnija forma konverzacija je pristup „**Tri amigosa (eng. *Three Amigos*)**“ u kome učestvuju: poslovni analitičar ili vlasnik proizvoda, tester i softverski inženjer. Svako od njih, iz domena svoje odgovornosti i stručnosti će naglašavati potrebne detalje. Poslovni analitičar ili vlasnik proizvoda će uočavati i naglašavati relevantnost i relativnu vrednost scenarija. Tester će naglašavati validnosti ali i granične i ekstremne slučajeve, dok će developer stavljati u fokus tehnička razmatranja i tehnološke principe i dobre prakse izrade tj. kodiranja. U isto vreme će i tester i developer poprimati bolje razumevanje celog projekta u smislu poslovnih zahteva i ciljeva, što nije slučaj u drugim praksama razvoja softvera. Obično ovo troje sede za kompjuterom i pišu zajedno inicijalni nacrt (eng. *draft*) automatizovanih scenarija. Odabere se priča, a zatim i skup primera kojim se iliustruju kriterijumi za prihvatanje koji se automatizuju u testove za prihvatanje (eng. *acceptance tests*). Posle inicijalnog padanja testa, programer kodira dok ne prođu svi scenariji jedne priča. Onda se grupno, prelazi na sledeću priču. Nastali i korišćeni testovi za prihvatanje iz prethodnog procesa (više priča), služe testeru kao polazna osnova za dalja i druga testiranja (manuelna ili ne), a različiti *stakeholder*-i mogu da pregledaju isporučenu implementacionu celinu (eng. *feature*) i utvrde da li ispunjava njihova očekivanja.

3. POREĐENJA BDD-A SA SLIČNIM PRISTUPIMA

Tehnički gledano, svi agilni koncepti predstavljaju načine da se softverski dizajn i razvoj poboljša, a zajedničko im je iterativni i inkrementalni pristup sa automatizacijom.

ATDD (eng. *Acceptance Test-Driven Development*) - Manje poznat i kao STDD (eng. *Story Test-Driven Development*) dovodi naručioce i druge relevantne *stakeholder*-e u testno-dizajnerski prostor. Na osnovu dubokog razumevanja šta se razvija, usaglašavaju se automatizovani kriterijumi za prihvatanje (eng. *acceptance criteria*) pre početka bilo kakvog kodiranja. Kao i kod BDD-a kolaboracija je na visokom nivou, a testovi su definisani pojednostavljenom engleskom terminologijom razumljivom i poslovnom svetu.

FDD (eng. *Feature-Driven Development*) - Jeff de Luca i Peter Coad se smatraju izumiteljima FDD-a jer su adresirali probleme *Waterfall* pristupa. Ovde se *feature* tretira kao veliki komad isporučene funkcionalnosti. Definisan je u 5 koraka: 1. Razviti celokupan model – 2. Napraviti listu *feature*-a – 3. Planirati po *feature*-u (jednom) – 4. Dizajnirati po *feature*-u – 5. Implementirati po *feature*-u. Prva tri koraka spadaju u „Startup“ fazu, dok 4. i 5. korak pripadaju „Construction“ fazi. Druga faza se iterativno ponavlja, a prva može biti ponovljena sa modifikacijama u toku trajanja projekta [10].

DDD (eng. *Domain-Driven Design*) – Razmišljanje o široj slici - Izraz je formiran u istoimenoj knjizi „Domain-Driven Design: Tackling Complexity in the Heart of Software“ čiji je autor Eric Evans [11]. Ova tehnika za razvoj softvera se bazira na objektno orijentisanoj analizi i dizajnu. Domen se definiše kao osnovna poslovna aktivnost samog korisnika. Ukratko: objektima poslovnog modela (korisnika) koji se implementira, odgovara set klasa u poslovnom domenu. Domenski objekti sadrže podatke i *setter* i *getter* metode, a menadžeri (kontroleri) implementiraju „ponašanje“ samih objekata. Ovo je nezamislivo bez samog korisnika, jer je nemoguće da softverske organizacije / kompanije poseduju mnogo različitih domenska (ne-IT) znanja. Koristeći jedinstven jezik za opisivanje budućeg sistema, premošćuje se razdor između poslovnih ljudi koji definišu ponašanje sistema (kreiraju zahteve) i programera koji implementiraju to ponašanje. Tako dizajniran sistem poseduje višeslojnu arhitekturu. Tipičan sistem poseduje: Korisnički interfejs (Prezentacioni sloj), Aplikacioni sloj, Domenski sloj (Sloj poslovne logike) i Infrastrukturni i tehničko-servisni sloj (Perzistivni sloj). Osnovni princip raslojavanja kod DDD-a jeste postojanje zavisnosti između slojeva na dole, počev od Prezentacionog sloja. Kolaboracija je i ovde bitna, a pre svega zbog razumevanja i usaglašavanja obe strane i činjenice da razvojni tim ne posedaju poslovna znanja korisnika. Da bi se sistem pravilno dizajnirao i podigla domenska ekspertiza, najviše komunikacije sa korisnikom se dešava na početku projekta.

TDD (eng. *Test-Driven Development*) – *Inside Out* razmišljanje - Osnovna ideja je pisanje testova pre kodiranja. Vuče korene iz test-first koncepta kod *Extreme Programming*-a, a najveći doprinos čini Kent Beck-ova knjiga „Test-driven Development by Example“. Robert C Martin definiše „Zakone TDD-a“:

1. Nije vam dozvoljeno da pišete bilo kakav produkcioni kod ako nije u cilju prolaska testa koji je pao
2. Nije vam dozovoljeno da pišete bilo šta dodatno osim onoliko koliko je potrebno da test padne
3. Nije vam dozvoljeno da pišete bilo šta dodatno od produkcionog koda osim onoliko koliko je potrebno da test koji je pao sada prođe

Suština je da se piše samo onoliko koda koliko je potrebno da se prođu svi definisani testovi. U svetu se za ovakav stil razvoja koriste i izrazi (prakse): KISS (eng. *Keep It Simple, Stupid*) i YAGNI (eng. *You Aren't Gonna Need It*) [12, 13].

Zbog svojih sličnosti, ATDD, BDD i TDD se mogu vrlo lako uporediti po više parametara. Jedna takva tabela je data u nastavku.

parametar	TDD	BDD	ATDD
Definicija	Razvojna tehnika koja se fokusira više na implementaciju feature-a	Razvojna tehnika koja se fokusira na kako sistem treba da se ponaša	Razvojna tehnika veoma slična i BDD-u i TDD-u, ali se fokusira na potpunom razumevanju zahteva
Učesnici	Programeri	Programeri, naručioci i drugi stakeholder-i, testeri	Progremeri, naručioci i drugi stakeholder-i, testeri
Jezik	Pisan u jeziku sličnom onom koji se koristi za razvoj feature-a (npr. Java, Python itd.)	Gherkin (pojednostavljen engleski)	Gherkin (pojednostavljen engleski)
Fokus	Unit Tests	Razumevanje ponašanja sistema	Pisanje Acceptance Tests
Alati	JDave, Cucumber, JBehave, Spec Flow, BeanSpec, Gherkin Concordian, FitNesse	Gherkin, Dave, Cucumber, JBehave, Spec Flow, BeanSpec, Concordian	TestNG, FitNesse, EasyB, Spectacular, Concordian, Thucydides
Agilni koraci	1. Testiraj 2. Kodiraj 3. Refaktoruj (sa ponavljanjem)	Implementiraj funkcionalnosti inkrementalno na osnovu očekivanog ponašanja. Kao ekstenzija TDD-a, isto se pišu testovi koji ne inicijalno ne prolaze na osnovu očekivanog ponašanja.	1. Diskutuj 2. Razvij 3. Isporuči (sa ponavljanjem)
Zahteva automatizaciju	Da	Da	Ne, ali je potrebna zbog regression testing-a
Test mapiranje	Svaka funkcionalnost treba da ima implementiran test.	Svaki feature treba da ima implementiran test „ponašanja“.	Svaka priča (story) treba da ima acceptance test.

Slika 12 – Uporedne karakteristike

4. PREDNOSTI I IZAZOVI

BDD je fokusiran na prepoznavanje i implementiranje samo onih *feature*-a koji donose poslovnu vrednost.

Smanjenje gubitaka - Postoje dva najveća rizika kada se nešto radi nepotrebno i praktično stvaraju gubici za organizacije koje učestvuju u projektu:

1. Kodiranje implementacionih celina (eng. *features*) koje nisu u skladu sa poslovnim ciljevima odnosno ne donose nikakvu poslovnu vrednost. Uzroci mogu biti različiti – od nerazumevanja razvojnog tima do personalnih karakteristika samih programera ili nekog drugog člana tima i logike "ovo je baš lepo da se ima" [2].
2. Kodiranje na način koji poslovanju nije koristan ili je preskup. Kasnije ovo uvodi dodatne gubitke ali i troškove, jer je potreban ponovni rad (eng. *rework*) [2].

Činjenica koja sprečava ostvarenje ova dva rizika navedena je na početku ovog poglavlja.

Smanjenje troškova - Direktna posledica prethodnog i "pravljenja pravog softvera" odnosno smanjenja gubitaka jeste smanjivanje troškova. Povećanje kvaliteta aplikacijskog koda takođe vrši uštede – smanjivanje broja bug-ova je direktno proporcijalan smanjivanju cene održavanja softvera.

Troškovi su svi resursi – od vremena do novca.

Lakše i bezbednije promene - BDD utiče značajno na vršenje promena i proširivanje postojećih aplikacija. Living documentation proistekla iz izvršnih specifikacija (eng. *executable specifications*) koristi terminologiju koju svi *stakeholder*-i razumeju. *High-level executable specifications* sa svojim poslovno-orijentisanim rečnikom olakšavaju razumevanje poslovnih ljudi (ne-IT stručnjaka), dok *low-levele executable specifications* ponašajući se kao tehnička dokumentacija omogućavaju različitim razvojnim timovima i tehničkim licima lakše razumevanje postojećih kodnih baza. Visok stepen automatizacije kako testova za prihvatanje (eng. *acceptance tests*) tako i *unit* testova, značajno smanjuju rizik od regresije (eng. *regression*) u slučajevima bilo kakvih dodatnih izmena.

Brže puštanje u produkciju - Automatizovanje različitih tipova testova direktno smanjuje vreme potrebno da se osigura što bezbednije puštanje softverskih rešenja u produkciju. Ono sa druge strane pravi dodatni prostor testerima da se posvete drugim oblicima manualnog testiranja (istraživačkim i netrivijalnim).

Visok stepen angažovanosti i saradnje samog biznisa - S obzirom da je razumevanje implementacionih zahteva i poslovnih vrednosti koje podupiru bazirano na konverzacijama i povratnim informacijama, ukoliko *stakeholder*-i mahom ne-IT lica, nisu voljna ili su sprečena da iste podrže, prednosti BDD-a neće biti iskorišćena kako su originalno zamišljene.

BDD postiže najbolji efekat u agilnom ili iterativnom okruženju - BDD podrazumeva da je skoro nemoguće definisati u potpunosti sve implementacione zahteve na početku. Oni će se "kristalisati" tokom životnog veka projekta i samim tim znači da je potrebno višestruko vraćanje i proveravanje istih. Ovaj proces je u skladu sa agilnim ili iterativnim pristupom i metodologijama.

BDD ne radi dobro u silosima - Mnoge organizacije još uvek imaju silosno organizovno interno poslovanje, od razvoja do testiranja. Dodatno su opterećene ugovaranjem kao procesom ali i klijentskim pristupom kod naručivanja i načinom kako vide ceo projekat. Prihvatanje BDD kao pristupa bi trebalo da se dešava na nivou cele kompanije odnosno, benefite ne bi trebalo da osećaju parcijalni timovi. Na primer: BDD praksa je prisutna kod razvojnog tima, ali kada kod pređe u održavanje postoji druga (eng. *waterfall-like*) praksa. Najznačajnije jeste da se poslovni analitičari i softverski inženjeri drže principa BDD-a, ali je sličan mod kolaboracije potreban i sa ostalim stakeholder-ima, na prvom mestu naručiocima zahteva [2].

Loše napisani testovi vode do većih troškova održavanja - Pisanje automatizovanih kriterijuma i testova, naročito u slučajevima kompleksnih aplikativnih rešenja, zahteva određene veštine i iskustvo. Ovo drugo je naročito izazovno na startu primenjivanja BDD-a. Nivo apstraktnosti i izražajnosti raste pravilnom upotrebom BDD praksi ali zahteva vreme kako bi se izbegle zamke i troškovi slabije napisanih testova. Vremenom, napisani testovi postaju glomazni u smislu njihovog volumena ali se od njihovog prilogaođavanja (eng. *refactoring*-a) ne sme odustajati, pored nužnog prilgođavanja i menjanja kodne baze.

S obzirom da BDD direktno zavisi od aglinih praksi, deo potencijalnih izazova leži i u pravilnom primenjivanju istih: komunikacija mora biti sveprisutna i transparentna ali dozirana. Informacije i zajednički ciljevi moraju biti jasni svima, ali na sastanke treba zvati samo potrebne ljude. I kod repetitivnih sastanaka, učesnici treba da dolaze pripremljeni. Broj sastanaka treba da bude minimalno potreban kako bi se ograničilo ukupno vreme provedeno na istim. Distribuiranost timova i razlike u časovnim zonama mogu uticati na kolaboraciju, komunikaciju i povratne informacije, ono na čemu se i sam BDD bazira.

Treba istaći još tri bitna izazova: postojanje QA testera sa iskustvom u kodiranju, kvalitet testiranja nije samo odgovornost testera i značaj pravilnog formatiranja BDD scenarija [14]. Prvi izazov proizilazi iz činjenice da je BDD visoko automatizovana metodologija i primena iste se na može bazirati samo na manuelnom testiranju. Definisanje i pisanje testnih scenarija je proizvode zajedničke kolaboracije. Kod definisanja scenarija treba imati u vidu sledeće:

1. Držati se poslovne logike a ne korisničkog interfejsa
2. Ne mešati više domena u jednom scenariju. Primer: ako scenario opisuje funkcionalno ponašanje, ne treba automatizovanim testovima potvrditi i performanse, sigurnost, sadržaj ili stil.
3. Ne treba imati previše koraka u jednom scenariju (idealno do 8, ne više od 15)

Na osnovu prikupljenih informacija i „Studije o karakteristikama BDD-a", primećeno je da BDD alati (xBehave familije, xSpec familije, StoryQ, Cucumber i SpecFlow) imaju slabu ili nikakvu podršku za fazu planiranja projekta [15]. xSpec familija i StoryQ zaostaju i sa podrškom za fazu analize. Potencijalne studije pa samim tim i dalji razvoj alata mogu doprineti dodatnoj podršci (struktuiranosti i automatizmu) kolaboraciji između poslovnog i razvojnog tima u ranim fazama projekta (planiranju i analizi).

Treba razmotriti uvođenje dodatnih pravila za mapiranje. Buduće studije bi trebalo da istraže da li je moguće efikasno mapirati čitave setove *feature*-a u *namespace*-ove ili pakete, a gde se već nalaze testne klase odgovarajućih scenarija.

ZAKLJUČAK

Danas se smatra da BDD pruža najbolje iz *Test-Drivent Development*-a, *Domain-Driven Design*-a i *Acceptance Test-Driven Planning* tehnika. U zavisnosti od kompleksnosti sistema koji se implementira, domenskih znanja naručioca (korisnika), internog znanja ali i drugih činjenica vezanih za predstojeći projekat, *Behavior-Driven Development* se može kombinovati sa drugim praksama i metodologijama – hibridni pristup. Primer je ukrštanje sa DDD-om.

Kao što se vidi u ovom radu, BDD je agilni softverski razvoj baziran na nekoliko osnovnih principa:

1. Opisati ponašanje sistema, a ne praviti de-facto specifikaciju rešenja
2. Otkriti ponašanje sistema koje donosi pravu poslovnu vrednost (više njih)
3. Koristiti konverzacije i primere kako bi se utvrdilo šta sistem treba da radi

Konverzacija i kolaboracija jesu ključ ovog pristupa. Različite tehnike i pristupi koji omogućavaju da se uspešno dese ovakvi sastanci, diskutuje i usaglasi oko reprezentativnih primera (sa konkretnim ulaznim i izlaznim vrednostima) su opisani u radu. Interakcija svih strana je obavezna, a struktuiran jezik (jednostavni delovi engleskog jezika) u pisanoj formi su razumljivi i poslovnim i IT licima. Na taj način razumevanje i usaglašavanje se formalizuje, a proistekli primeri su spremni za dalje korišćenje – automatizaciju. Ovo predstavlja rad sa zahtevima visokog nivoa (eng. *high-level requirements*)

Drugi deo BDD-a tj. automatizacija predstavlja takođe veliku vrednost ovakvog pristupa. Primeri (eng. *examples*) napisani u prethodno pomenutoj formi se lako pretvaraju u izvršne specifikacije (eng. *executable specifications*). Najkorišćeniji alati za to su Cucumber i JBehave. Takođe, pisanje izvršne specifikacije na *unit-testing* nivou je pravolinijsko. Proizvod izvršnih specifikacija i visokog i niskog nivoa jeste „uvek svež" (eng. *up-to-date*) dokument zahteva, koji može biti pregledan bilo kada od strane bilo koga. Živa dokumentacija (eng. *living documentation*) je razumljiva svima i vrlo korisna u izveštajnom segmentu.

Sve ovo upućuje na zaključak da *Behavior-Driven Development* dosta racionalizuje razvojni proces, ali i čini održavanja tako implementiranih rešenja, manje skupim i sa manje rizika. Razumevanje poslovnih ciljeva i očekivanog ponašanja sistema koji se pravi, sa BDD struktuiranim tehničkim pristupom, donosi pravu poslovnu vrednost.

LITERATURA

[1] Dan North & Associates, „Introducing BDD", URL: https://dannorth.net/introducing-bdd/ [pristupljeno: avgust, 2020.]
[2] John Ferguson Smart, BDD in Action (Manning, 2014)
[3] Gojko Adžić, Specification by Example (Manning, 2011)
[4] „Sample code for the 2nd Edition of BDD In Action", GitHub, URL: https://github.com/bdd-in-action [pristupljeno: avgust, 2020.]
[5] Chriss Mats, Gojko Adžić, „Feature Injection: three steps to success" (2011), InfoQ, URL: https://www.infoq.com/articles/feature-injection-success/ [pristupljeno: avgust, 2020.]
[6] Gojko Adžić, Impact Mapping (Provoking Thoughts, 2012)
[7] Pollyanna Pixton, Niel Nickolaisen, Todd Little, and Kent McDonald, Stand Back and Deliver: Accelerating Business Agility (Addison-Wesley Professional, 2009)
[8] Chris Matts and Olav Maassen, "'Real Options' Underlie Agile Practices" (2007), InfoQ, URL: http://www.infoq.com/articles/real-options-enhance-agility [pristupljeno: avgust, 2020.]
[9] Dan North, "Introducing Deliberate Discovery" (2010), URL: http://dannorth.net/2010/08/30/introducingdeliberate-discovery [pristupljeno: avgust, 2020.]
[10] Nirav Asar, „Exploring the Subtle Differences Between Agile Paradigms" (2010), AgileConnection, URL: https://www.agileconnection.com/article/exploring-subtle-differences-between-agile-paradigms [pristupljeno: avgust, 2020.]
[11] Eric Evans, Domain Driven Design (Addison-Wesley Professional, 2003).
[12] Jash Unadkat, „BDD vs TDD vs ATDD : Key Differences" (2019), BrowseStack, URL: https://www.browserstack.com/guide/tdd-vs-bdd-vs-atdd [pristupljeno: avgust, 2020.]
[13] „Test-driven development", Wikipedia, URL: https://en.wikipedia.org/wiki/Test-driven_development [pristupljeno: avgust, 2020.]
[14] Evgeny Tkachenko, „11 Reasons Behavior-Driven Development Can Fail" (2019), URL: https://www.stickyminds.com/article/11-reasons-behavior-driven-development-can-fail
[15] Carlos Solis, Xiaofeng Wang, „A Study of the Characteristics of Behaviour Driven Development" (2011), IEEE, URL: https://ieeexplore.ieee.org/document/6068372?arnumber=6068372 [pristupljeno: avgust, 2020.]

Milutin Cvetković, MScEE (Računarska tehnika i informatika) / Senior Konsultant Projektni Menadžer u Endava d.o.o
Kontakt: milutinc@gmail.com
Oblast interesovanja: Digitalna transformacija i platformski proizvodi, Upravljanje projektima i resursima, Tehnike programiranja, Razvoj informacionih sistema, Sistemi za upravljanje poslovnim procesima, Sistemi za podršku odlučivanju, Mašinsko učenje, Otkrivanje zakonitosti u podacima

prof. Sladjan Babarogić, Fakultet organizacionih nauka, Univerzitet u Beogradu
Kontakt: babarogic.sladjan@fon.bg.ac.rs
Oblast interesovanja: Razvoj IS vođen modelima, Poslovna analiza i Modelovanje poslovnih procesa, Metodologije razvoja informacionih sistema i Baze podataka