UDC: 004.4:004.8 INFO M: str. 29-37

AUTOMATIZOVANO TESTIRANJE PRIHVATLJIVOSTI SOFTVERA PRIMENOM FITNESSE-A
AUTOMATED ACCEPTANCE TESTING USING FITNESSE

Sonja Dimitrijević, Saša D. Lazarević

REZIME: Automatizacija testova prihvatljivosti je nova strategija čije je uspešno usvajanje moguće i danas primenom nekih od dostupnih okvira, alata i aktuelnih praksi. Glavni cilj ovog rada je da ukaže na mogućnosti automatizacije testova prihvatljivosti polazeći sa stanovišta da bi oni morali biti korisnički orijentisani. U radu su izloženi neki od najznačajnijih koncepata i rezultata iz ove oblasti i opisan studijski primer realizovan primenom jednog od najpopularnijih okvira i alata – FitNesse. Studijski primer je bio usmeren na ispitivanje pogodnosti često korišćenih stilova FitNesse tabela za kreiranje jednostavnih uniformnih korisnički-orijentisanih testova s obzirom na različite podmodele objektno-orijentisanog domenskog modela sistema koji se testira. Pokazano je da FitNesse fleksibilan okvir tako da kreiranje testnih slučajeva navedenih karakteristika, čak i primenom osnovnih stilova tabela ne mora biti pod značajnim uticajem različitih podmodela domenskog modela. Međutim, evidentni su i brojni izazovi koji se moraju uzeti u obzir prilikom evaluiranja potencijala FitNesse okvira za konkretnu primenu.
KLJUČNE REČI: automatizovano testiranje prihvatljivosti, FitNesse, izvršiva specifikacija, agilni razvoj softvera

ABSTRACT: Automated acceptance testing is a new strategy whose successful implementation is possible even nowadays applying some of the available frameworks, tools and current practices. The main objective of this paper is to show the potential of automated acceptance testing from the viewpoint that automated tests should be user-oriented. The paper presents some of the most important concepts and results from this area and describes a case study carried out using one of the most popular frameworks and tools - FitNesse. The case study was aimed at verifying suitability of commonly used styles of FitNesse tables for creating simple analogous user-oriented tests considering various sub-models of the object-oriented domain model of a system under test. It is shown that FitNesse is a flexible framework, so creation of test cases that meet the mentioned characteristics, even with application of the basic table styles, may not be significantly influenced by the different sub-models of a domain model. Many challenges that must be taken into account when evaluating potential of FitNesse for specific application are evident though.
KEY WORDS: automated acceptance testing, FitNesse, executable specification, agile software development

1. UVOD

Automatizovano testiranje prihvatljivosti softvera je nova strategija za koju se tvrdi da može doneti brojne koristi projektima razvoja softvera. Za razliku od tradicionalnih pristupa u kojima se na testove prihvatljivosti obično gleda isključivo kao na artifakte testiranja, što oni i jesu, agilni pristupi ukazuju i na sledeći njihov aspekt; testovi prihvatljivosti su svojevrsna specifikacija zahteva. Testovima prihvatljivosti se specifikuju softverski zahtevi zahvaljujući tome što se njima izražavaju kriterijumi pomoću kojih naručilac treba da utvrdi da li sistem/deo sistema zadovoljava njihove potrebe, odnosno da li su zahtevi realizovani u potpunosti i korektno [1].

Primenom odgovarajućih alata za automatizaciju testova prihvatljivosti, ovakva specifikacija postaje izvršiva. Automatizovani izvršivi testovi ulivaju poverenje naručiocu da su zahtevi korektno shvaćeni. Izvršavanje tih testova može poslužiti kao indikator napretka projekta u realnom vremenu i „živa“ dokumentacija softverskog sistema. Da bi se testovi učinili izvršnim, predstavnici naručioca i članovi razvojnog tima su prinuđeni da budu konkretni i jasni u specifikaciji zahteva [2]. Posledično, a imajući u vidu da su tipične agilne prakse orijentisane na aktivno učešće korisnika/predstavnika naručioca tokom čitavog projekta, primena automatizovanih testova prihvatljivosti može poboljšti komunikaciju između učesnika na projektu [2].

Trenutno je dostupan veći broj alata za automatizovanje testova prihvatljivosti [3]. Od njih se očekuje, osim odgovarajuće podrške za realizaciju testova prihvatljivosti kao artifakata testiranja, odgovarajuća podrška realizaciji testova prihvatljivosti kao izvršive specifikacije zahteva. To znači, da ovi alati treba da omoguće efektivniji kolaborativni način za sticanje i deljenje znanja o sistemu koji se razvija, odnosno realizaciju testnih slučajeva koji će biti jasni i korisniku, pa čak i da obezbede da predstavnici naručioca (ljudi iz poslovne, a ne IT sfere) sami pišu testne slučajeve.

Kako je automatizacija testova prihvatljivosti još uvek u razvoju, oprečna su tvrđenja i malo je dokaza o njenoj delotvornosti i potencijalnim koristima koje može doneti, kao i o mogućnostima efektivne i efikasne primene dostupnih okvira i alata.

Glavni cilj ovog rada je da ukaže na mogućnosti automatizacije testova prihvatljivosti, imajući jasno u vidu da bi testovi prihvatljivosti morali biti korisnički (a ne tehnički) orijentisani i služiti kao medijum za kolaborativno sticanje znanja o zahtevima koje sistem treba da podrži, a zatim i kao osnova za automatsko validiranje tih zahteva. Rad izlaže rezultate studije kojom je ispitana pogodnost osnovnih stilova tebela, tj. tipova ‘fixture’ klasa, jednog od najpopularnijih okvira i alata, FitNesse, za automatizaciju (korisnički-orijentisanih) testova prihvatljivosti nad različitim podmodelima objektno-orijentisanog modela. Tačnije, proučavanjem dostupne literature i dokumentacije, kao i realizacijom studijskog primera, nastojalo se proveriti da li se često korišćeni stilovi FitNesse tabela (tj. tipovi *fixture* klasa) mogu koristiti za kreiranje korisnički-

orijentisanih testnih slučajeva koji bi obuhvatili iste/slične testne korake za različite podmodele objektno-orijentisanog modela prema kojim se realizuju domenski objekti sistema koji se testira.

Posle uvodne sekcije, u Sekciji 2, razmotreni su koncepti agilnog testiranja prihvatljivosti – agilni testovi prihvatljivosti, i automatizacija testova prihvatljivosti, i prezentirana praksa – razvoj vođen testovima prihvatljivosti. U Sekciji 3, iznete su značajene odrednice i karakteristike FitNesse-a. U Sekciji 5, izložen je postupak primenjen u realizaciji studijskog primera. U Sekciji 6 je dat sažet opis realizacije studijskog primera. Na kraju, u Sekciji 7 je dat zaključak.

2. AUTOMATIZOVANO TESTIRANJE PRIHVATLJIVOSTI SOFTVERA

2.1. *Testiranje prihvatljivosti softvera u agilnom razvoju softvera*

Agilni razvoj softvera, naročito Ekstremno programiranje (XP), doneo je radikalne promene u tradicionalnom načinu rada organizacija koje se bave razvojem softvera, pogotovo u pogledu testiranja softvera [4]. Aspekti od značaja za proces testiranja softvera po kojima se agilni razvoj znatno razlikuje od tradicionalnih pristupa su sledeći [5]:

- *Veća saradnja*. Agilni programeri blisko sarađuju i priznaju da je dokumentacija najmanje efektivan način saradnje.
- *Kraći ciklusi rada*. Vreme između specifikovanja zahteva i validiranja tog zahteva se sada meri minutima, a ne mesecima, zbog usvajanja praksi razvoja vođenog testovima (test-driven development – TDD), veće saradnje i manjeg oslanjanja na privremenu dokumentaciju.
- *Veća fleksibilnost se zahteva od testera*. Prošlo je vreme „kompletne specifikacije“ po kojoj se testira, a koju razvojni tim predaje testerima. Agilni timovi uvažavaju evoluiranje zahteva tokom projekta.
- *Zahteva se veća disciplina*. Sprovođenje praksi zasnovanih, između ostalog, na razvoju potencijalno isporučivog softvera na redovnoj osnovi, pisanju testova pre pisanja tek onoliko izvornog koda koliko je potrebno da bi test prošao itd., može lako krenuti u neželjenom smeru bez discipline.
- *Veća odgovornost se zahteva od interesnih strana.* Usvajanje praksi poput onih koje su zasnovane na aktivnom učešću interesnih strana, prioritetizaciji zahteva i realizaciji softvera na redovnoj osnovi traži od interesnih strana da budu odgovorne za odluke koje donose.
- *Zahteva se veći opseg veština*. Nije dovoljno da neko bude samo tester ili samo programer ili samo analitičar, itd. Visoko iterativni i kolaborativni pristupi ne negiraju potrebu za specijalistima, ali oni moraju imati mnogo širi opseg veština i sposobnosti.

Zahvaljujući svim novinama koje donosi, agilni razvoj softvera redefiniše tradicionalni proces – osiguranje kvaliteta počev od formalnih uloga do aktivnosti na dnevnoj osnovi čineći neke njegove tradicionalne odgovornosti i autpute irelevantnim. Agilne prakse se zasnivaju na punoj integraciji testiranja i razvoja što se odražava kako na organizacionu strukturu, tako i na poslovne politike visokog nivoa [4].

Prva asocijacija za testiranje na agilnom projektu, najčešće je jedinično testiranje. Razlog za to je što su prakse i alati za podršku ovoj vrsti testiranja dobro razvijene i dobro dokumentovane. Međutim, pored jediničnog testiranja, testiranje prihvatljivosti sve više dobija na značaju i primetni su nezanemarljivi napori koji se ulažu u razvoj praksi i alata za podršku ovoj vrsti testiranja. Dok jedinični testovi proveravaju ispravnost malog segmenta koda (obično određene metode u nekom specifičnom kontekstu), testovi prihvatljivosti su testovi softverske funkcionalnosti iz perspektive naručioca, odnosno korisnika. ‘Agilni testovi prihvatljivosti’ su poznati pod različitim terminima kao što su: korisnički testovi, izvršiva specifikacija, specifikacija na osnovu primera, testovi (korisničkih) priča. Različiti termini ukazuju na različite aspekte ovih testova. Iako, u nekim slučajevima termin ‘test prihvatljivosti’ može dovesti u zabludu, to je osnovni, široko prihvaćen termin.

Testiranje prihvatljivosti softvera je zastupljeno i u tradicionalnom razvoju softvera. Kako je definisano u IEEE standardu 1012-1986, to je formalno testiranje u cilju utvrđivanja da li sistem zadovoljava kriterijume prihvatljivosti i u cilju omogućavanja naručiocu da utvrdi da li da prihvati sistem. Ovaj oblik testiranja se obično sprovodi na čitavom softverskom sistemu ili većem delu sistema [6]. No, zbog svih specifičnih karakteristika agilnog razvoja softvera, jasno je da se razlike u odnosu na tradicionalne pristupe mogu očekivati i u pogledu ovog procesa. Pre svega, značajne implikacije na testiranje prihvatljivosti softvera imaju pristupi i prakse vezane za inženjerstvo zahteva, jer su pristupi validiranju zahteva nerazdvojivo vezani za pristupe prikupljanju i dokumentovanju tih zahteva. Agilni pristupi softverskim zahtevima se, između ostalog, odlikuju uvažavanjem dinamične prirode softverskih zahteva, iterativnošću i kontinuiranim interakcijama sa korisnicima. Prikupljanje zahteva, pregovaranje, dokumentovanje i validiranje tih zahteva se odvijaju zajedno u svakom kratkom razvojnom ciklusu [7]. Tako se testiranje prihvatljivosti javlja mnogo ranije i sprovodi se znatno češće [8]. Idealno, testovi prihvatljivosti se pišu pre produkcionog koda koji ih realizuje, i na taj način testovi postaju detaljna specifikacija zahteva [5].

2.2. *Testovi pruhvatljivosti*

U skladu sa definicijom testiranja prihvatljivosti softvera, test prihvatljivosti je formalni test koji se sprovodi kako bi se utvrdilo da li sistem zadovoljava kriterijume prihvatljivosti – iz perspektive korisnika, i da omogući korisnicima da utvrde da li da prihvate ili ne prihvate sistem. Inicijalno su nazivani funkcionalnim testovima zato što se svakim testom nastoji testirati funkcionalnost softverskog sistema. Za razliku od tradicionalnih pristupa u kojima se na testove prihvatljivosti obično gleda isključivo kao na artifakte testiranja, što oni i jesu, agilni pristupi ukazuju na to da se testovima prihvatlji-

vosti beleže softverski zahtevi. Prema [9], testovi prihvatljivosti su specifikacije za željeno ponašanje i funkcionalnosti sistema. Oni nose informacije kako, za datu korisničku priču (u pristupi zasnovanom na korisničkim pričama), odnosno zahtev, sistem obrađuje određene uslove i ulaze, i sa kojim vrstama ishoda. Uostalom, testovi prihvatljivosti predstavljaju način da se izraze mnogi detalji dobijeni u konverzacijama između korisnika i razvojnih inženjera, kao i da se dokumentuju pretpostavke koje eventualno nisu bile razmotrene [1].

Kako bi se ostvarila puna korist od testova prihvatljivosti, trebalo bi da oni poseduju određene atribute. Pre svega, testovi prihvatljivosti su [9]:

- U vlasništu naručioca;
- Pišu ih zajedno naručilac/korisnik, programer i/ili tester;
- Stavljaju naglasak na šta, ne na kako;
- Izraženi su jezikom domena problema;
- Sažeti, precizni i nedvosmisleni.

Prema tome, testovi prihvatljivosti se razlikuju od jediničnih testova prvenstveno prema nameni i tome šta se testira, ali i prema: vlasništvu, kreatorima, jeziku kojim su izraženi. Dok jedinične testove osmišljavaju i pišu programeri za određenu jedinicu koda, i u vlasništvu su razvojnog tima, testove prihvatljivosti makar osmišljavaju, a eventualno i pišu korisnici jezikom domena problema za određenu funkcinalnost sistema, i u vlasništvu su naručioca. Otuda još jedan naziv, korisnički test.

Specifikovanje testova prihvatljivosti se može realizovati na nekoliko načina. Ako se koristi popularna tehnika za prikupljanje zahteva zasnovana na korisničkim pričama, ono može podrazumevati zapisivanje testova na poleđini, ili posebno predviđenom delu indeksirane karte (papirne koje se tradicionalno koriste ili virtuelne koje podržava sve veći broj softverskih alata za agilni projektni menadžment) kojom se korisnička priča dokumentuje. Treba imati u vidu da korisničke priče nisu samo kratki pisani opisi, već podrazumevaju i konverzacije koje obezbeđuju detalje, kao i *testove* koji prenose i dokumentuju detalje i mogu biti od pomoći za utvrđivanje da li je korisnička priča realizovana [1]. Pod terminom „test“ se u ovom slučaju može podrazumevati i zabeležen kriterijum prihvatljivosti koji sistem treba da zadovolji, a koji će nekim budućim testom, manuelnim ili automatizovanim, biti proveren. Za specifikovanje navedenih testova, mogu se koristiti alati za automatizovano testiranje. No, tehnički kompleksnije alternative zahtevaju odgovarajuće tehničko znanje članova tima naručioca [1].

Testovi prihvatljivosti doprinose softverskom projektu na tri načina ([10] preko [3]). Prvo, oni „hvataju“ zahteve sistema na neposredno proverljiv način. Drugo, testovi prihvatljivosti izlažu probleme koje tehnički orijentisani testovi propuste (ciljaju „end-to-end“ funkcionalnost). Treće, oni pružaju informaciju u kojoj meri je sistem realizovan. Sistem je spreman za isporuku kad prođe sve testove prihvatljivosti, tako da je procenat takvih testova koji su prošli „jedina praktična definicija“ stvarnog napretka. Prema [3], nijedna druga mera, kao što su procenat napisanog koda, investirano vreme, korišćeni resursi itd., nisu dovoljno dobri pokazatelji kompletnosti kao što to testovi prihvatljivosti jesu.

2.3. *Automatizacija testova prihvatljivosti*

Automatizacija testova prihvatljivosti je nova obećavajuća inicijativa koja ima za cilj da olakša i poboljša ovaj proces.

Postoji više razloga zbog čega je automatizacija prihvatljivosti vrlo poželjna na softverskom projektu. Manuelno testiranje prihvatljivosti softvera je u većini slučajeva skupo, vremenski zahtevno i monotono [11], između ostalog, zbog potrebe da se u velikoj meri ponavlja testiranje mnogih funkcionalnosti kako bi se ispitali svi granični slučajevi [3]. Kod iterativnih razvojnih procesa zasnovanih na kratkim iteracijama kakvi su agilni procesi, manuelno testiranje prihvatljivosti zahteva alokaciju značajnih resursa na kraju svake iteracije kada treba bar jednom izvršiti testove [11]. Međutim, ovo važi za sve testove prihvatljivosti iz prethodnih iteracija kako bi se osiguralo da u delu softvera realizovanom u prethodnim iteracijama nisu proizvedene greške dodavanjem funkcionalnosti u sukcesivnim iteracijama [1]. Stoga, izvršavanje testova prihvatljivosti postaje sve više vremenski i resursno zahtevno sa svakom sledećom iteracijom.

Osnovna ideja na kojoj se automatizacija testova prihvatljivosti zasniva je da se zahtevi i željeni ishodi dokumentuju u formatu koji omogućava automatsko i ponovljivo testiranje [11]. Ideja je u velikoj meri bazirana na filozofiji jediničnog testiranja. Ipak, testovi prihvatljivosti uglavnom ciljaju poslovnu logiku [12], i treba da posluže kao izvršiva specifikacija zahteva.

Trenutno je dostupan veći broj alata, okvira, proizvoda i tehnika za automatizovanje testova prihvatljivosti. Većina njih je usmerena na pisanje testova na formalan, ali razumljiv način, izvršavanje testova na konfigurabilni način, i pružanje korisnih izveštaja i rezultata [3]. Neki od poznatijih su: Fit, FitNesse, Concordion, Cucumber, JBehave, TextTest.

Ipak, treba imati u vidu da automatizacija testova prihvatljivosti ima svoju cenu koja u najvećoj meri potiče od troškova održavanja testova. Potreba za izmenama testova prihvatljivosti potiče iz dva izvora promena: promene zahteva i promene u implementaciji sistema. Kako se navodi u [13], razlika između organizacija koje uspešno primenjuju automatizaciju testova prihvatljivosti od onih koje nisu uspele da postignu željene rezultate i odustale su od ove prakse, je upravo u tome što uspešne organizacije nisu iznenađene troškovima održavanja, već ih očekuju.

Da bi se automatizacija i izvršiv razvoj vođen testovima prihvatljivosti (razmotren u nastavku) dalje razvijao, potrebna je realizacija boljih alata, kao i veći broj empirijskih studija koje bi potvrdile ili odbacile dosadašnja tvrđenja. Prema [2], potrebno je da alati sledeće generacije za automatizovano testiranje prihvatljivosti budu prilagođeni širokom opsegu korisnika i multimodalni, odnosno da omogućavaju izražavanje testova u različitim formatima kako bi specifikacija zahteva bila u formi koja je lako razumljiva pojedinoj grupi korisnika. Pored toga, neophodno je da budući alati bolje odgovore na izazove vezane za organizaciju velikog broja testova (npr. efikasnim algoritmima pretrage, dobrim vizuelnim prikazom testova), kao i za njihovo lako održavanje (npr. dobrim mehanizmima za upravljanje testovima i refaktorisanje testova).

2.4. *Razvoj vođen testovima prihvatljivosti*

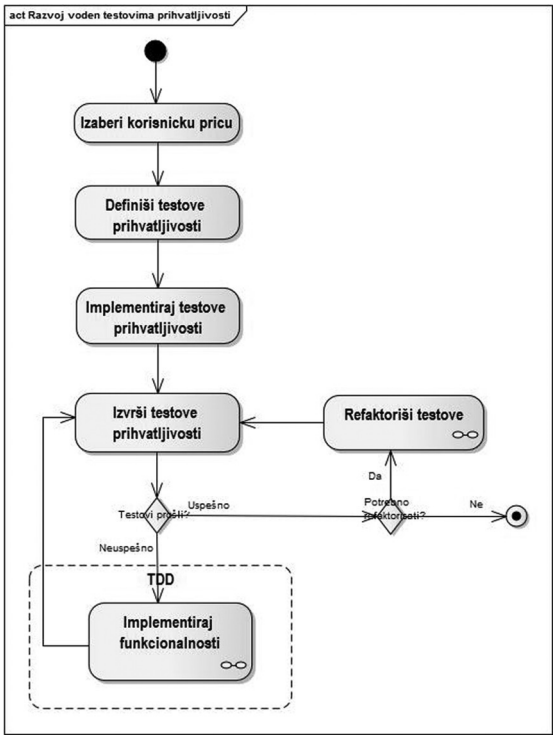
Tokom poslednje decenije razvoj vođen testovima prihvatljivosti (eng. Acceptance Test Driven Development – ATDD) je privukao dosta pažnje među pristalicama agilnog pravca razvoja softvera. Ovaj pristup je predložen kao način da se obezbedi zajedničko razumevanje zahteva od strane tima naručioca i razvojnog tima i da se na automatski i ponovljiv način testira softver na poslovnom nivou [14]. Pored ovog termina, koriste se i drugi kao što su razvoj vođen testovima (korisničkih) priča (eng. Story Test Driven Development – STDD) [15], specifikacija na osnovu primera (eng. Specification by Example) [16].

Prema [17], ideja ATDD-a (korišćen je termin STDD) je u prisutna u krugovima pristalica agilnog razvoja softvera počev od objavljivanja knjige Kent Beck-a [18] 1999. godine. Beck je, između ostalog, poznat po tome što je predstavio TDD [19] – razvojnu tehniku koja donosi mnogo više od promene redosleda kodiranja i testiranja po čemu je naročito poznata. Međutim, Beck je predložio pisanje dve vrste testova: testova iz perspektive programera i testova iz perspektive naručioca. Testovi koje pišu programeri pokazuju samo programersku perspektivu sistema zbog čega je potreban još jedan skup testova iz perspektive naručioca. Dupla provera, odnosno korišćenje dva tipa testova jednog naspram drugog, pomogla bi da se blagovremeno pronađu neuhvaćeni problemi. Međutim, dok je TDD već dobro poznat u agilnoj praksi, ATDD je znatno slabije prihvaćen i još uvek postoji dosta konfuzije oko samog koncepta i mogućnosti primene [17].

ATDD treba da pomogne realizaciji softvera visokog kvaliteta koji ispunjava poslovne potrebe pouzdano kao što TDD doprinosi tehničkom kvalitetu softvera. Takođe treba da doprinese koordiniranju softverskog projekta tako što pomaže da se isporuči tačno ono što naručilac želi onda kada želi, i tako što obeshrabruje polovičnu realizaciju zahtevane funkcionalnosti. Dok se u TDD-u prvo definiše očekivano specifično ponašanje koje kod treba da podrži, a zatim se definisano ponašanje implementira, u ATDD-u se prvo definiše karakteristična funkcionalnost (vredna za korisnika ili naručioca) koju sistem kao celina treba da podrži, a onda se definisana funkcionalnost implementira, često primenom TDD-a [9].

Razvoj vođen testovima prihvatljivosti, prikazan je UML dijagramom aktivnosti na slici 1. Ciklus, koji traje kroz iteraciju sve dok ima još priča koje treba implementirati, započinje odabirom korisničke priče. Zatim sledi definisanje testova prihvatljivosti za odabranu korisničku priču, potom i njihova implementacija, tj. prevođenje testova u automatizovane, izvršive testove, nakon čega sledi izvršavanje testova, i konačno, implementacija funkcionalnosti koja omogućava da se test uspešno izvrši. Izvršavanje testova se sprovodi čak i kada funkcionalnost koja se testira nije implementirana. Takvo inicijalno izvršavanje testova će biti neuspešno. Sa implementacijom funkcionalnosti se kreće, tek pošto se identifikuju neuspešni testovi. Implementacija funkcionalnosti je potproces koji se, kako je na dijagramu naznačeno, može realizovati primenom TDD-a. Pošto testovi prođu, treba preispitati mogućnost njihovog refaktorisanja, odnosno mogućnost poboljšanja njihovog dizajna kako bi se, na primer, eliminisali redundanti delovi testova, omogućila višestruka upotrebljivost i olakšalo održavanje, ili testovi učinili jasnijim. U pogledu refaktorisanja testova, uzori i dobre prakse su znatno razvijeniji i dokumentovaniji u TDD-u [20]. U praksi, naravno, proces nije uvek ovako jednostavan kao što je prikazano na dijagramu. Korisničke priče mogu biti dvosmislene ili čak kontradiktorne, mogu biti međusobno zavisne što utiče na redosled realizacije, itd.

Bitno svojstvo ATDD-a je da je to timski proces [9]. Kako se sugeriše u [21], pronalaženje odgovarajućeg ulaznog i očekivanog izlaznog skupa podataka zahteva domensko znanje, dok njihovo „pretvaranje“ u testove zahteva znanje iz oblasti testiranja. Za razliku od TDD-a koji zahteva od programera da prihvate nove prakse, ATDD proces zahteva da ljudi iz svih funkcionalnih sredina učestvuju u procesu [17].



Slika 1. – *UML dijagram aktivnosti – Razvoj vođen testovima prihvatljivosti*

Na kraju, ako se postavi pitanje aktuelne pozicije automatizacije testova prihvatljivosti i ATDD-a, evidentno je da se u literaturi sreće nemali broj izveštaja i tvrdnji o koristima automatizacije testova prihvatljivosti. Međutim, kako je primećeno u [11], one uglavnom potiču iz referata u kojima se iznose iskustva iz privrede. Empirijski dokazi su još uvek slabi i uglavnom potiču iz kontrolisanih eksperimenata sa studentima. Rezultati jednog širokog sistematskog pregleda literature iz 2010. godine radi identifikovanja praznina u aktuelnim istraživanjima, prezentovani su u [17]. Uopšteno gledano, rezultati su pokazali da postoji veoma ograničen broj radova u literaturi i da je empirijska osnova ATDD-a slaba. Postojeći radovi se fokusiraju na vreme, ljude, dizajn i alate, međutim još uvek ima mnogo pitanja na čije odgovore se čeka, kao i konfliktnih rezultata.

3. O FITNESSE OKVIRU/ALATU

Jedan od najpopularnijih okvira i alata za automatizovano testiranje prihvatljivosti softvera je FitNesse. Kako navode njegovi kreatori, FitNesse je [22]:

- alat za testiranje softvera;
- kolaborativni alat za razvoj softvera;
- wiki;
- web server.

FitNesse je lagan okvir otvorenog koda i alat koji omogućava: (1) definisanje testova prihvatljivosti – web strana koje sadrže jednostavne tabele ulaza i očekivanog izlaza; (2) izvršavanje tih testova i pregledanje rezultata. FitNesse je takođe kolaborativni alat zato što, prema rečima kreatora, pomaže članovima razvojnog tima i tima naručioca da zajedno uče šta softver „treba da radi“ i da to automatski uporede sa onim „šta on stvarno radi“. Drugim rečima, omogućava upoređivanje očekivanja sa stvarnim rezultatima. Time što je realizovan kao wiki, kreiranje i izmena web (testnih i drugih) strana su učinjeni jednostavnim. Iako funkcioniše kao web server, ne zahteva konfigurisanje ili podešavanje [22].

FitNesse je delimično baziran na Fit-u. Fit je najpoznatiji okvir (otvorenog koda) zasnovan na tabelarnom pristupu testiranju prihvatljivosti, kreiran sa namerom da omogući predstavnicima naručioca da pišu „izvršive“ testove prihvatljivosti korišćenjem jednostavnih HTML tabela. Iako je sam Fit omogućavao izvršavanje testnih tabela, nije obezbeđivao lak način za kreiranje tih tabela ili za prikazivanje rezultata testova. Iz tog razloga je kreiran FitNesse kao HTML i wiki „front-end“ Fit-a. FitNesse je realizovan sa korisnim „mark-up“ prečicama dizajniranim da pomognu u realizaciji testnih strana i olakšaju kreiranje, izvršavanje, organizovanje, opisivanje i deljenje testnih tabela. Kada se govori o primeni FitNesse-a sa Fit-om u pozadini kao mašinom (eng. engine) odgovornom za izvršavanje testova, često se koristi kovanica Fit/FitNesse.

Za rad sa Fit/FitNesse-om, potrebno je razumeti tri osnovna elementa: (1) *Fit/FitNesse tabela,* (2) *Fixture*, (3) *Test runner.* HTML tabele su način za izražavanje poslovne logike. *Fixture* je interfejs između testnih slučajeva (tabela), testnog okvira i sistema koji se testira. Ovaj tanak integracioni sloj obično čine klase (ili procedure/funkcije) koje pišu programeri. *Test runner* upoređuje očekivanja naručioca sa stvarnim rezultatima i izveštava o rezultatu bojenjem odgovarajućih ćelija tabela (zeleno, ako je test prošao; crveno, ako test nije prošao) i, eventualno, prikazivanjem poruke o greški, odnosno, stvarnog rezultata naspram očekivanog.

Mada je inicijalno bio baziran na Fit okviru, FitNesse je realizovan fleksibilno, tako da su se posle nekog vremena pojavile i alternative Fit-u. Naime, postoji više mogućnosti kako FitNesse povezati sa sistemom koji se testira (Tabela 1) [22].

Tabela 1: Pregled test sistema za FitNesse

Jezik	Fit	Slim	FitLibrary
Java	Da (built-in)	Da (built in)	Da
.NET	Da (FitSharp)	Da (FitSharp)	Da (FitSharp)
Python	Da	Da	Ne
Ruby	Da	Da	Ne
C	No	Da	Ne
PHP	Da	Da	Ne
JavaScript	Ne	Da	Ne

FitLibrary je prva veća alternativa Fit-u. FitSharp je .NET implementacija koja podržava Fit, Slim i FitLibrary. Slim (Simple List Invocation Method) je skoriji *test runner* baziran na novoj strategiji. Slim prepušta FitNesse-u HTML procesiranje, kao i upoređivanje i bojenje, tj. interpretiranje rezultata. SlimService je odgovoran da pronađe *fixture* klase i da izvrši komandu. Ova strategija ima više prednosti: (1) visoka portabilnost, (2) testne tabele ostaju konzistentne bez obzira na platformu (jer su funkcionalnosti na FitNesse strani), (3) moguće je istraživati nove testne sintakse (pošto HTML nije unutrašnji deo Slim-a) [22].

Neki od najčešće korišćenih stilova Fit tabela/*fixture* klasa: *ColumnFixture* (redovi predstavljaju ulaze i očekivane izlaze); *RowFixture* (za testiranje kolekcije elemenata); *ActionFixture* (za testne slučajeve koji imitiraju seriju događaja); *Comment* tabele (tabelarni komentar koji se ne izvršava kao test). Naravno, FitLibrary i Slim obezbeđuju svoje skupove *fixture* klasa.

Fit je predstavljen 2002. godine, tako da je Fit/FitNesse imao deceniju da sazri i bude proučavan/prihvaćen. Iako se to ne može reći za Fit u poslednjih nekoliko godina, FitNesse se i dalje aktivno razvija. No, stavovi praktičara su podeljeni. Sa jedne strane su protivnici koji tvrde da se automatizacija testova prihvatljivosti jednostavno ne isplati navodeći, kao razloge, pre svega, zahtevno održavanje testova ili to što predstavnici naručioca ne pišu testove. Sa druge strane, Fit/FitNesse je stekao i izvestan broj pristalica o čemu svedoči veći broj studija u kojima je istraživana primena, naročito, Fit okvira. Rezultati šireg pregleda literature i studije slučaja u vezi sa automatizovanim testiranjem prihvatljivosti softvera primenom Fit okvira, izneti su u [12]. Rezultati su pokazali da je donekle neopravdano očekivati od korisnika da aktivno izražavaju zahteve u formi automatizovanih testova, ali da to ne mora nužno da se odražava na izvodljivost ovog pristupa. Iako je nađeno da su u većini slučajeva programeri pisali testove prihvatljivosti, autori veruju da je automatizacija ove vrste testova vrlo obećavajuća za poboljšanje efikasnosti razvoja softvera. Sa druge strane, konstatovano je da pisanje i ništa manje održavanje automatizovanih testova mogu imati visoku cenu zbog čega je potrebno pažljivo unapred razmotriti potencijalne koristi u odnosu na troškove.

U studiji (2011) zasnovanoj na intervjuima sa razvojnim inženjerima koji svedoče o uspešnoj primeni FitNesse testova u praksi, zaključuje se da automatizovano testiranje prihvatljivosti može biti održiva strategija, ali da zahteva obazriv pristup. Među iznetim rezultatima, navodi se da je FitNesse dobar izbor kod razrešavanja zahteva koji podrazumevaju složenu poslovnu logiku i viši nivo neizvesnosti/nejasnoće, a pri tom su pogodni za tabelarne forme koje alat podržava [23]. Pored toga, potvrđeno je tvrđenje iz [24] da korišćenje

FitNesse tabela kao medijuma za komuniciranje sa naručiocem pospešuje razumevanje zahteva.

4. PRIMENJEN POSTUPAK

Studijski primer je realizovan prevashodno u cilju ispitivanja mogućnosti kreiranja korisnički-orijentisanih testnih slučajeva koji bi obuhvatili iste/slične testne korake primenom osnovnih stilova FitNesse tabela, a uzimajući u obzir različite podmodele domenskog modela sistema koji se testira. Drugim rečima, realizovanim studijskim primerom, nastojalo se proveriti da li je i na koji način kreiranje uniformnih korisnički-orijentisanih testnih slučajeva uslovljeno složenošću domenskih objekata sistema koji se testira, a prema kojima se realizuje interfejs primenom odabranih *fixture* klasa. Osnovna pretpostavka od koje se pošlo je da iz korisničke perspektive, testiranje poslovnih pravila nad određenim skupom podataka ne bi trebalo da se značajno konceptualno razlikuje zbog načina na koji su oni modelovani u sistemu.

Postupak primenjen u realizaciji studijskog primera je obuhvatio sledeće korake: odabir podmodela; definisanje zahteva u formi korisničkih priča (sa kriterijumima prihvatljivosti); definisanje i kreiranje testnih slučajeva (realizacija iz perspektive naručioca); implementacija integracionog sloja (realizacija iz perspektive programera); i na kraju izvršavanja testova i pregledanja rezultata uz razmatranje mogućih pravaca u refaktorisanju testova.

Postupak se naslanja na agilnu praksu u prikupljanju zahteva baziranu na korisničkim pričama i definisanju testnih slučajeva prihvatljivosti na osnovu kriterijuma prihvatljivosti pobrojanih u korisničkoj priči. Realizacija testnih slučajeva je zasnovana na praksi kreiranja FitNesse tabela (perspektiva korisnika), pre testnih (*fixture*) klasa – interfejsa ka sistemu koji se testira (perspektiva programera).

5. OPIS REALIZACIJE STUDIJSKOG PRIMERA

Studijski primer je realizovan u .NET okruženju i C# programskom jeziku. Izbor stila testnih tabela/*fixture* klasa je izvršen u skladu sa sledećim kriterijumima:

- pogodnost za konkretni testni slučaj;
- podrška realizaciji testova prihvatljivosti, odnosno korisničkih testova (neki stilovi testnih tabela/fixture klasa su pogodniji za realizaciju više tehnički orijentisanih testova);
- pripadaju Fit i FitLibrary bibliotekama (kao duže prisutnim i češće korišćenim u odnosu na Slim).

Sam sistem koji se testira je jednostavan studijski primer poslovne desktop aplikacije baziran na troslojnoj arhitekturi. Sistem za upravljanje bazom podataka je SQL Server Express. Pošto prevazilazi opseg ovog rada, njegova realizacija neće biti razmatrana.

5.1. *Odabir podmodela*

Za realizovan studijski primer, odabrani su podmodeli objektno-orijentisanog domenskog modela u skladu sa sledećim kriterijumima:

- podmodeli reprezentuju različite tipove UML (eng. Unified Modeling Language) asocijacija;
- podmodeli pripadaju istom modelu.

Tako su, u skladu sa zadatim kriterijumima, odabrani podmodeli u kojima se realizuje: 'klasična' asocijacija između klasa, 'self' asocijacija, asociajativna klasa, tj. asocijacija sa atributima i metodama, kompozicija.

5.2. *Korisničke priče*

Za definisanje zahteva u formi korisničkih priča, korišćen je templejt: *"Kao [korisnik], želim [funkciju], kako bi [vrednost]"*. Za svaku korisničku priču, utvrđeni su i kriterijumi prihvatljivosti, kao što se može videti na primeru datom ispod za priču: Evidentiraj angažovanje zaposlenog na projektu:

Kao menadžer ljudskih resursa, želim da u sistemu budu evidentirana sva angažovanja zaposlenih na projektima kako bi se pratila angažovanost zaposlenih.

Neki od definisanih kriterijuma prihvatljivosti:

- *Proveriti unos angažovanja zaposlenog na projektu;*
- *Proveriti ažuriranje angažovanja zaposlenog na projektu;*
- *Proveriti da procenat angažovanja zaposlenog na projektu ne sme da premaši 100%;*
- *Proveriti da ukupan procenat angažovanja zaposlenog na projektima ne sme da premaši 100% ;*

5.3. *Definisanje i kreiranje testnih slučajeva*

Za svaku korisničku priču, kreirano je više testnih slučajeva – jedan osnovni i više alternativnih. Kreiranje testnih slučajeva u FitNesse-u, obično znači kreiranje više tabela na jednoj strani. I kada je moguće da se sve što je potrebno nađe u okviru jedne (velike) tabele, prilično je jasno da bi takva testna strana bila teško čitljiva ili potpuno nečitljiva, čime se gubi svaki smisao ovakvih testova. Tabele na jednoj strani se izvršavaju u nizu. To omogućava kreiranje scenarija čiji su koraci predstavljeni različitim manjim tabelama.

Korišćeni su sledeći tipovi FitNesse tabela, odnosno *fixture* klasa: *ColumnFixture* (Fit), *DoFixture* (FitLibrary), *SetUpFixture* (FitLibrary), *RowFixture* (Fit).

ColumnFixture klasa se može koristiti za izvršavanje gotovo svakog testnog slučaja. Tabela za ColumnFixture ima najmanje tri reda. U prvom redu se daje ime fixture klase, u drugom nazivi ulaznih parametara, metode/propertija koji vraća izlaz, dok su naredni redovi namenjeni testnim podacima i očekivanim rezultatima. Takođe se može koristiti da postavi podatke za druge testne slučajeve i da pozove metode koje će da „počiste" za drugim. *ColumnFixture* klasa je laka za razumevanje i korišćenje i, pošto se može koristiti u tako mnogo situacija, prva je koja se uzima u razmatranje prilikom odlučivanja kako da se test realizuje. *ColumnFixture* klasa je naročito dobar izbor, ukoliko isti testovi treba da se ponove za odgovarajući broj različitih kombinacija ulaznih parametara. Ukoliko taj broj nije poznat, ili je potrebno izvršiti samo jednu

proveru, verovatno postoje bolja rešenja koja štede vreme i napor. Na slici 2, prikazana je *ColumnFixture* tabela za pojednostavljen korak jednog od realizovanih testnih slučajeva kojim se proverava višestruki unos (uključujući nedozvoljene vrednosti). Zelena boja ćelija kolone predviđene za izlaz označava da je test prošao.

ColumnFixture obezbeđuje da u slučaju nepodudaranja vraćenog izlaza sa očekivanim, u odgovarajućoj ćeliji označenoj crvenom bojom budu prikazane obe, stvarna i očekivana vrednost. Ako je bačen izuzetak, u odgovarajućoj ćeliji tabele će biti ispisana poruka o izuzetku, dok će ćelija biti označena žutom bojom.

Unesi Organizacionu Jedinicu		
naziv	pripada_id	proveri unos organizacione jedinice?
Proizvodnja	1	Unos uspešan
Management	2	Unos uspešan
Maloprodaja	2	Unos uspešan
IT	99	Greška
IT	null	Greška

Slika 2. – *ColumnFixture tabela za pojednostavljen testni korak provere višestrukog unosa (test je izvršen)*

Dakle, testni slučaj se može realizovati tako što se za svaki korak kreira zasebna tabela. Međutim, ukoliko broj testnih koraka raste i, pogotovo, ukoliko se svaki korak izvršava samo jednom, *ColumnFixture* klasa neće više biti prvi izbor. Njena primena bi značila dosta suvišnog rada, kako na kreiranju testnih tabela, tako i na kreiranju testnih klasa. Jedna od zanimljivih alternativa je *DoFixture*. *DoFixture* je deo FitLibrary. Ova klasa koristi mnogo manje striktan format tabele i omogućava da testni koraci liče na rečenice. Osim prvog reda u kojem se daje naziv testne klase, svaki naredni red izvršava jednu metodu te klase. Ćelije reda tabele zajedno formiraju naziv metode (npr. *UkupanProcenatAngazovanjaZaposlenogNaProjektimaJe*) koji čak može biti „isprekidan“ vrednostima parametara te metode. Na sledećoj slici (Slika 3) je prikazana FitNesse tabela kojom je definisan pojednostavljen osnovni testni slučaj za evidentiranje angažovanja zaposlenog (podmodel sa asocijativnom klasom). U prvom i trećem koraku se vrši provera unosa angažovanja zaposlenog na projektima, a u petom ažuriranja angažovanja zaposlenog na jednom od projekata. Posle svakog od ovih koraka, proverava se ukupan procenat angažovanja zaposlenog.

Posle izvršavanja testnog slučaja, ćelije sa nazivom metode su označene zelenom bojom (pošto su svi testni koraci prošli), tako da se jasno vide argumenti tih metoda. U opštem slučaju, ukoliko odgovarajuća metoda testne klase vraća *bool* vrednost, smatra se da je: test prošao, za vraćeno *true*, odnosno da nije prošao, za vraćeno *false*. To se može preokrenuti korišćenjem ključne reči *reject* kao prefiksa, što je korisno za alternativne testne slučajeve. *DoFixture* obezbeđuje još neke korisne ključne reči (npr. check – prikazuje, pored očekivane, vraćenu vrednost u slučaju kada ona odstupa od očekivane).

DoFixture omogućava da se drugi *fixture*-i ugrade u testni slučaj zahvaljujući funkcionalnosti nazvanoj *flow* režim. Kada je testna strana u *flow* režimu, testni koraci se prvo uparuju sa metodama *flow fixture* klase (*DoFixture* u *flow* režimu). U *flow* režimu, moguće je ugraditi i koristiti *fixture*-e više puta, pri čemu *DoFixture* klasa obezbeđuje kontekst. Ovo je FitNesse verzija *dependency injection* paterna koja olakšava pisanje i kombinovanje *fixture* klasa. Inače, ovakve testne slučajeve je lakše čitati, jer prvi red tabela ne mora biti posvećen nazivu klase (već nazivu *DoFixture* metode). Uz to, *flow* režim omogućava definisanje i čuvanje konteksta u privatnim varijablama *fixture* klase, a ne u statičkim globalnim varijablama.

U realizaciji testnih slučajeva za korisničku priču - *Evidentiraj projekat uključujući njegove projektne zadatke* (podmodel sa kompozicijom), korišćena je upravo *flow* funkcionalnost *DoFixture* klase. Tako su, na primer, testni koraci za pripremu podataka za projekat i pripremu podataka za pripadajuće projektne zadatke realizovani primenom SetUpFixture klase. Ovom klasom se ništa ne testira, već se samo podaci preuzimaju i za razliku od ColumnFixture klase, ona ne zahteva dodatnu kolonu u tabeli za poziv metode. Zatim, testni koraci za testiranje unosa/ažuriranja podataka, implementirani

EvidentiranjeAngazovanja													
Proveri Unos Angazovanja Zaposlenog	3	Koji Je	rukovodilac	Na Projektu	1	Od	01.01.2012	Do	01.12.2012	Sa Statusom	true	I Procentom Angazovanja	40
Ukupan procenat angazovanja zaposlenog	3	na projektima je	40										
Proveri Unos Angazovanja Zaposlenog	3	Koji Je	rukovodilac	Na Projektu	2	Od	01.10.2012	Do	01.06.2013	Sa Statusom	true	I Procentom Angazovanja	20
Ukupan procenat angazovanja zaposlenog	3	na projektima je	60										
Proveri Izmenu Angazovanja Zaposlenog	3	Koji Je	rukovodilac	Na Projektu	2	Od	01.10.2012	Do	01.06.2013	Sa Statusom	true	I Procentom Angazovanja	30
Ukupan procenat angazovanja zaposlenog	3	na projektima je	70										

Slika 3. – *DoFixture tabela za pojednostavljen osnovni testni slučaj – Evidentiraj angažovanje (test je izvršen)*

su preko metoda same *DoFixture* klase, a testni korak provere da li projektni zadaci poslednjeg unetog projekta odgovaraju podacima pripremljenim za unos (Slika 4), realizovan je primenom *RowFixture* klase. FitNesse obezbeđuje *ArrayFixture* (FitLibrary) i *RowFixture* (Fit) za testiranje redova i kolekcija objekata. *RowFixture* se koristi za veću preciznost, odnosno bolje izveštavanje o greškama

Pregledaj Projekat Ciji Je ID	<<id 5	
opis	datum_pocetka	datum_zavrsetka
Inicijalizacija	01/10/2012 09:00:00	02/10/2012 17:00:00
Planiranje	03/10/2012 09:00:00	03/11/2012 17:00:00
Realizacija	04/11/2012 09:00:00	04/03/2013 18:00:00 *expected* 03/04/2013 17:00:00 *actual*
Isporuka	05/03/2013 09:00:00	04/04/2013 17:00:00

Slika 4. – *RowFixture tabela za testiranje kolekcije projektnih zadataka (test je izvršen)*

Zahvaljujući flow režimu *DoFixture* klase, moguće je kombinovati i iskoristiti prednosti različitih stilova testnih tabela, tj. tipova *fixture* klasa u jednom testnom slučaju. *Flow režim*, takođe, omogućava pisanje testnih slučajeva u formi scenarija i za testiranje funkcionalnosti iza kojih se kriju složeni podmodeli,

Uostalom, FitNesse je vrlo zgodan za pisanje objašnjenja testova i pojedinačnih tabela. Moguće je uključiti slike, obezbediti linkove do dodatnih informacija ili modifikovati testne strane već na neki način za koji se smatra da poboljšava zajedničko (naručioca i razvojnog tima) razumevanje zahteva.

5.4. *Implementacija integracionog sloja*

Da bi kreirane testne tabele mogle da se izvrše, bilo je potrebno realizovati odgovarajuće testne klase kojim se obezbeđuje postavka testnog okruženja, odnosno čuvanje statičkog konteksta za testni slučaj, i testne tabele povezuju sa sistemom koji se testira.

U slučaju realizovanog studijskog primera, u klasi za postavku testnog okruženja, vrši se (najmanje) inicijalizacija kontolera aplikacione logike Testnom klasom/klasama koja nasleđuje odgovarajuću *fixture* klasu, definišu se odgovarajući ulazi i metode koje se pozivaju u kreiranim testnim tabelama. Metode samo pozivaju operacije sistema koji se testira, uglavnom vraćaju rezultat, i eventualno, u zavisnosti od tipa *fixture* klase, obrađuju izuzetake.

U slučaju *DoFixture* klase u *flow* režimu, ukoliko metoda vraća *Fixture* instancu, onda se testna tabela u kojoj se poziva ta metoda kreira i izvršava u skladu sa implementacijom konkretne *fixture* klase. Na isečku jedne testne klase koja je implementirana u flow režimu, datom u nastavku, može se videti postavka testnog okruženja, kao i metoda *PripremiProjekat()* koja vraća instancu klase *ProjekatFixture*. *ProjekatFixture* je implementiran kao *SetUpFixture* i koristi se za pripremu podataka.

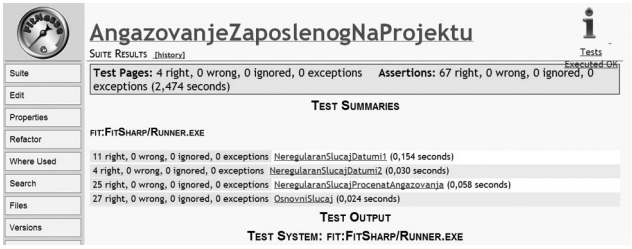
```
//START: Flow klasa
public class ProjekatFlowModeTest : fitlibrary.DoFixture
{
    internal static KALProjekat kontroler;
    private DomenskeKlase.Projekat projekat;
    private ArrayList zadaci;

    //START: DO - postavka testnog okruzenja
    public ProjekatFlowModeTest()
    {
        kontroler = new KALProjekat();
        projekat = new DomenskeKlase.Projekat();
        zadaci = new ArrayList();
    }//END: DO - postavka testnog okruzenja

    //START: DO - punjenje instance Projekat
    public Fixture PripremiProjekat()
    {
        return new ProjekatFixture(projekat);
    }
    //END: DO - punjenje instance Projekat
        …
}
//END: Flow klasa
```

5.5. *Izvršavanje testova i pregledanje rezultata*

Da bi testni slučajevi uopšte mogli da se izvrše, neophodno je u vrhu testne strane definisati varijable i postaviti odgovarajuće putanje do: *test runner-a* koji se koristi (FitSharp), FitNesse biblioteka koje se koriste, implementiranog integracionog sloja, kao i odgovarajućih komponenata sistema koji se testira. Odnosne varijable i putanje mogu da zauzmu dobar deo testne strane. Još gore, mogu zbunjivati naručioca, a kod kreiranja više testnih slučajeva za istu korisničku priču, navedeni sadržaj bi se ponavljao. Ponavljanje istog sadržaja na više testnih strana ukazuje na potrebu za refaktorisanjem.

Međutim, zahvaljujući funkcionalnosti grupisanja povezanih testnih slučajeva u test „suite“, korišćenoj i u realizovanom studijskom primeru, omogućeno je da opštenamenski sadržaj za određenu grupu testova bude izdvojen na jednom mestu. Osim toga, FitNesse obezbeđuje specijalne strane - SetUp i TearDown. Ako se definišu, one bivaju automatski uključene na početku, odnosno na kraju svih testnih strana u okviru jednog test „suite“-a. To znači da tabele koje se izvršavaju pre svake testne strane, treba izdvojiti u SetUp (npr. importovanje testnih klasa, postavka testnog okruženja itd.), dok tabele koje se izvršavaju posle svake testne strane, treba izdvojiti u TearDown.

Posle izvršavanja testnih slučajeva (koje se pokreće jednostavnim klikom na dugme), sumirani rezultati su vidljivi na vrhu strane, a konkretni u samim testnim tabelama (zelene ili crvene ćelije, stvarna vrednost naspram očekivane u slučaju nepodudaranja itd.) Takođe je moguće izvršiti sve testove iz jednog test „suite“-a u jednom potezu i pregledati sumirane rezultate (Slika 4).



Slika 5. – *Sumirani rezultati izvršavanja testova u „suite"-u*

Osim toga, svaki put kada se izvrše testovi, rezultati se snimaju, i omogućeno je njihovo kasnije pregledavanje i upoređivanje.

6. ZAKLJUČAK

Ispitivanje čak i manjeg podskupa funkcionalnosti koje pruža FitNesse, trenutno jedan od najpopularnijih okvira i alata za automatizovano testiranje prihvatljivosti, otkriva njegovu fleksibilnost i široke mogućnosti. Pisanje testnih scenarija čiji su koraci predstavljeni jednostavnim tabelarnim formama, zatim moguća primena alternativnih manje striktnih formata tabela (npr. testni koraci mogu ličiti na rečenice), kao i jednostavno izvršavanje testova i prikaz rezultata u prigodnoj formi, predstavljaju solidnu osnovu za efektivne korisnički orijentisane testove prihvatljivosti. Tome doprinosi i zasnovanost na wiki tehnologiji i, posledično, mogućnost da se jednostavno doda sadržaj (poput pojašnjenja, slika, linkova do dodatnih informacija) ili modifikuju testne strane na način za koji se smatra da poboljšava zajedničko razumevanje zahteva.

Studijski primer je pokazao da kreiranje korisnički-orijentisanih testnih slučajeva sa istim/sličnim testnim koracima, čak i primenom osnovnih stilova tabela, ne mora biti uslovljeno različitim podmodelima objektno-orijentisanog domenskog modela sistema koji se testira. To je obezbeđeno mogućnošću kombinovanja različitih klasa interfejsa (između testnih slučajeva, testnog okvira i sistema koji se testira - *fixture*). Na taj način, priprema podataka za test i testni koraci se mogu realizovati različitim klasama, odnosno stilovima tabela, prigodnim za određen podmodel podataka. Međutim, tabelarne forme nisu pogodne za sve vrste korisničkih zahteva. Uz to, evidentni su izazovi u pogledu izbora pristupa s obzirom na fleksibilnost, a sa povećanjem broja testova, i u pogledu njihovog efikasnog organizovanja i upravljanja. Osim poboljšanja alata (sa aspekta organizacije većeg broja testova, refaktorisanja, verzioniranja itd.), dokumentovanje dobrih praksi i preporuka u primeni okvira, kao i veći broj studija, bili bi od značajne pomoći njegovim aktuelnim i budućim korisnicima.

7. LITERATURA

[1] Cohn, M., *User Stories Applied for Agile Software Development*, Addison-Wesley, 2004
[2] Park, S.S., Maurer, F., *The Benefits and Challenges of Executable Acceptance Testing*, APOS `08 Proceedings of the 2008 international workshop on Scrutinizing agile practices or shoot-out at the agile corral, ACM, pp. 19-22, 2008
[3] Talby, D., Nakar, O., Shmuely, N., Margolin, E., Keren, A., *A Process-Complete Automatic Acceptance Testing Framework*, Proceedings of IEEE International Conference on Software - Science, Technology and Engineering, pp. 129-138, 2005
[4] Talby, D., Keren, A., *Agile Software Testing in a Large-Scale Project*, IEEE Software, Volume 23, Issue 4, pp. 30-37, 2006
[5] Scott, A., *Agile Testing and Quality Strategies: Discipline over Rhetoric*, http://www.ambysoft.com/essays/agileTesting.html, 2010
[6] Ricca F., Torchiano, M., Di Penta, M. C., Paolo, T., *Using acceptance tests as a support for clarifying requirements: A series of experiments*, Information and Software Technology, 51(2): pp. 270-283, 2009
[7] Ramesh, B., Cao, L., Baskerville, R., *Agile Requirements Engineering Practices and Challenges: an Empirical Study*, Information Systems Journal, Volume 20, Issue 5, pp. 449–480, 2010
[8] Huo, M., Verner, J., Zhu, L., Babar, M.A., *Software Quality and Agile Methods*, COMPSAC '04, Proceedings of the 28th Annual International Computer Software and Applications Conference , Volume 01, pp. 520-525, 2004
[9] Koskela, L., *Acceptance TDD Explained*, Methods and Tools, Issue Summer 2008
[10] Kaner, C., James, B. and Pettichord, B., *Lessons Learned in Software Testing*, John Wiley & Sons, 2001
[11] Noll, J., Pirotta, E., Solis, C., Wang, X., *Automated Acceptance Testing vs. Quality: A Case Study of an Open Source Project*, University of Limerick Institutional Repository**,** 2011
[12] Haugset, B., Hanssen, G.K., *Automated Acceptance Testing: A Literature Review and an Industrial Case Study*, Proceedings of Agile08, pp. 27-38, 2008
[13] Emery, H.D., *Writing Maintainable Automated Acceptance Tests*, http://dhemery.com/pdf/writing_maintainable_automated_acceptance_tests.pdf, 2009
[14] Haugset, B., Stalhane, T., *Automated Acceptance Testing as an Agile Requirements Engineering Practice*, Proceedings of 45th Hawaii International Conference on System Science (HICSS), pp. 5289 – 5298, 2012
[15] Mugridge, R., Cunningham, W., *Agile Test Composition*, from *Extreme Programming and Agile Processes in Software Engineering*, LNCS, 2005, Volume 3556/2005, pp. 137-144, Springer-Verlag, 2005
[16] Fowler, M., Specification by Example, http://www.martinfowler.com/bliki/SpecificationByExample.html
[17] Park, S., Maurer, F., *An Extended Review on Story Test Driven Development*, Technical report, Computer Science, 2010
[18] Beck, K., *Extreme Programming Explained: Embrace Change*, Addison-Wesley, 1999
[19] Beck, K., *Test Driven Development: By Example.* Addison-Wesley Professional, 2002
[20] Mirković, S., Lazarević, S.D., Test Driven Development uzori i refaktorisanje testnog koda, Info M, Volumen 45, str. 42-54, 2013.
[21] Kerievsky, J., Storytesting, http://industrialxp.org/storytesting.html
[22] http://fitnesse.org/
[23] Haugset, B., Hanssen, G.K., *The home ground of Autamated Acceptance Testing: Mature use of FitNesse*, Agile Conference (AGILE), pp. 97-106, 2011
[24] Ricca, F., Di Penta, M., Torchiano, M., *Guidlines on the use of Fit tables in software maintenance tasks: Lessons learned from 8 experiments*, IEEE International Conference on Software Maintenance, pp. 317-326, 2008

Sonja Dimitrijević, Institut „Mihajlo Pupin", Beograd
Kontakt: sonja.dimitrijevic@pupin.rs
Oblast interesovanja: softversko inženjerstvo, informacioni sistemi

Prof. dr. Saša D. Lazarević, Fakultet organizacionih nauka, Beograd
Kontakt: slazar@fon.rs
Oblast interesovanja: softversko inženjerstvo, informacioni sistemi, baze podataka, sistemi za upravljanje dokumentacijom, .NET platform