Zbornik radova Fakulteta tehničkih nauka, Novi Sad

UDK: 004.9
DOI: https://doi.org/10.24867/13BE04Kahriman

UTICAJ RAZVOJA VOĐENOG TESTOVIMA NA DIZAJN SOFTVERA

IMPACT OF TEST DRIVEN DEVELOPMENT ON SOFTWARE DESIGN

Aleksandar Kahriman, *Fakultet tehničkih nauka, Novi Sad*

Oblast – SOFTVERSKO INŽENJERSTVO

Kratak sadržaj - *U ovom radu su predloženi principi razvoja vođenog testovima čija primena vodi do fluidnijeg razvoja. Dodatno je ustanovljena veza između fluidnosti razvoja i potencijala za evoluiranje aplikativnog dizajna softvera. Proaktivnim pristupom evoluiranja modula se iterativnim postupkom dolazi do boljeg aplikativnog dizajna.*

Ključne reči: *razvoj vođen testovima, TDD, dizajn softvera, arhitektura softvera*

Abstract – *This paper offers principals of test driven development, applying which leads to more fluid development. Additionally, a link between fluid development and the potential to evolve application design is established. Proactive approach to evolving modules in iterations results in better application design*

Keywords: *test driven development, TDD, software architecture, software design*

1. UVOD

Verovatno većim ubrzanjem nego bilo koja druga, softverska industrija evoluira metodologije i alate pomoću kojih teži da zadovolji apetite čovečanstva za revolucionarnom tehnologijom. Softverska infrastruktura, koja se postepeno razvijala, je danas najuticajniji katalizator rasta industrije. Ona omogućava razvoj izuzetno složenih rešenja za frakciju vremena koje je bilo potrebno u prošlosti. Mnoge kompanije postaju lideri na tržištima u kojima posluju upravo zahvaljujući ekonomičnoj optimizaciji i automatizaciji procesa koju softverska infrastruktura omogućava. Upravo ova duboka integrisanost u svakodnevni rad mnogih industrija i živote individua stvara veliki pritisak na softversku industriju da isporučuje kvalitetan softver u što kraćim mogućim vremenskim rokovima.

Značajan doprinos softverskoj infrastrukturi čine metodologije čije praktikovanje dovodi do većeg stepena optimizacije razvoja softvera. Jedna takva metodologija je razvoj vođen testovima (engl. *Test Driven Development, TDD*), koja se pojavila u industriji početkom 21. veka. TDD je ciklična metodologija razvoja softvera u kojoj svaki ciklus započinje pisanjem automatskog testa. Dodatno, tokom svakog ciklusa vrši se neophodna implementacija kako bi se test uspešno izvršio. Ciklus se završava restrukturiranjem koda u cilju povećanja kvaliteta. Proaktivnim pristupom, TDD metodologija smanjuje broj grešaka u razvoju i pruža sigurnost tokom evoluiranja softvera [1-3]. Uspešno praktikovanje TDD metodologije je veoma izazovno iz razloga što zahteva pragmatičnost, kao i dugogodišnje iskustvo inženjera u aplikativnom dizajnu. Stoga, neophodno je osmisliti dizajn strategije kako bi razvoj softvera bio efektivniji. U radu su predloženi principi, koji su oformljeni unapređivanjem ovih strategija. Ovi principi omogućavaju fluidnije TDD cikluse, koji dovode do boljeg aplikativnog dizajna i umanjuju negativne posledice praktikovanja metodologije.

NAPOMENA:
Ovaj rad proistekao je iz master rada čiji mentor je bio dr Goran Savić.

2. TEORIJSKI OKVIR

Softverska industrija je davno prepoznala implikacije grešaka u razvoju i cenu ispravki istih u različitim periodima životnog ciklusa softvera. U cilju amortizacije ovog problema uvođene su mere verifikacije ispravnog ponašanja softvera. Inicijalna strategija se oslanjala isključivo na manuelnu verifikaciju, ispoljavajući negativne karakteristike pri skaliranju. Naglim razvojem softverske infrastrukture automatsko testiranje doprinosi skalabilnosti verifikacije u razvoju softvera. Izvršavanje velikog broja automatskih testova koji verodostojno analiziraju integritet softvera za relativno kratak vremenski period uliva samopouzdanje u proces razvoja koje se direktno preslikava u rast kvaliteta.

2.1. Razvoj vođen testovima

Postepenim uvođenjem preporučenih praksi za pisanje kvalitetnih automatskih testova ispraćeno evolucijom tehnologije i pratećih alata postalo je moguće izvršavati veliki broj testova za relativno kratko vreme. Ovaj napredak u automatskom testiranju je doveo do novih metodologija razvoja softvera, među kojima je TDD. Najvažnija novina koju TDD potencira je pisanje automatskih testova neposredno uoči aplikativnog koda čije ponašanje verifikuju. Automatski test postaje forma specifikacije za mali deo funkcionalnosti koji je potrebno implementirati. Pošto aplikativni kod u trenutku pisanja testa još ne postoji, test se inicijalno neuspešno izvršava, označavajući crvenu fazu TDD ciklusa. Dalje se prelazi u zelenu fazu ciklusa, u kojoj se piše minimalni aplikativni kod kako bi se test uspešno izvršio. Posle zelene faze nastupa opciona faza restrukturiranja u kojoj se aplikativni kod evoluira pod sigurnosnom mrežom automatskih testova. Kada inženjer iskustveno odluči da dalje poboljšanje aplikativnog koda nije pragmatično, i kada se svi testovi uspešno izvršavaju, prelazi se u novi TDD ciklus, tj. u crvenu fazu.

Prilikom TDD ciklusa konstantnim pokretanjem automatskih testova se dobijaju blagovremene povratne informacije o integritetu softvera. Na ovaj način, održavanje kvaliteta aplikativnog koda i smanjenje tehničkog duga prelaze iz reaktivnih u proaktivne radnje rezultujući

predvidivim razvojem, manjim brojem grešaka, usporavanjem degradacije kvaliteta softvera vremenom i smanjenjem stresa razvojnog tima [1-6].

2.2. Dizajn softvera

Dizajn (arhitektura) softvera predstavlja skup značajnih odluka koje oblikuju softverski sistem, gde se značaj meri cenom promene. Ove odluke se konstantno donose na raznim nivoima granularnosti u opsegu od jedne funkcionalnosti do celog sistema. Cilj softverskog dizajna je optimizacija količine resursa koji su potrebni da se razvije i održava softverski sistem [7].

Rastom kompleksnosti sistema, do te mere da je standardno da se prosečan softver sastoji iz stotina hiljada linija aplikativnog koda, došlo je do ozbiljnih izazova, posebno u segmentima ispravnosti i proširivosti. Naravno, industrija je vremenom učila iz svojih poteškoća, uočavajući apstraktne šablone koji vode do smanjenja entropije i povećanja kvaliteta u razvoju softvera. Prethodno spomenuti šabloni su diskutovani aktivno u okviru zajednice softverskih inženjera tokom poslednjih par decenija. Dobra sinteza ovih napora je skup principa poznat kao SOLID akronim [7]. Ovi principi postavljaju apstraktne ciljeve ka kojima treba težiti kako bi se došlo do boljeg aplikativnog dizajna. Jedini način za poboljšanje dizajna je analiza svake odluke u kontekstu SOLID i drugih principa dizajniranja softvera. TDD navodi na rasparčavanje problema na manje delove. Prolazeći kroz TDD cikluse na većem nivou granularnosti inženjer dobija blagovremene detaljne povratne informacije o kompleksnosti dizajna i ispravnosti rešenja. Dakle, usporavajući tempo razvoja i povećavajući granularnost analize problema TDD na proaktivan način povećava verovatnoću donošenja boljih dizajn odluka.

3. PREDLOG PRINCIPA EFEKTIVNOG PRAKTIKOVANJA TDD METODOLOGIJE

Aplikativni dizajn i TDD imaju kontraintuitivan odnos. Naime, dobar dizajn vodi do fluidnijih TDD ciklusa koji omogućavaju bolji uviđaj u celokupnu sliku tehničkih i domenskih suptilnosti problema. Blagovremeno prepoznavanje ovih suptilnosti vodi do boljeg dizajna. Aktivan rad na poboljšanju aplikativnog dizajna i praktikovanje TDD metodologije čine sinergiju koja ima potencijal povećanja kvaliteta softvera. Ukoliko TDD ciklusi nisu fluidni, gube svoju potentnost u produkovanju dobrog dizajna. Značajna činjenica je da TDD ne garantuje dobar dizajn, već forsira blagovremenu evaluaciju dizajna, koja vodi do fluidnosti pri razvoju. U daljem radu su identifikovani i predloženi principi koji omogućavajući fluidnije TDD cikluse vode do boljeg aplikativnog dizajna i obrnuto.

3.1. Princip konstrukcije grafa zavisnosti

Upotrebom klasične metode instanciranja objekata u testovima dolazi do konflikta između težnje ka evoluciji modula i zadržavanja osećaja sigurnosti ispravnog ponašanja postojećih funkcionalnosti. Uvođenjem novih klasa konstruktori koji se koriste za instanciranje objekata bivaju narušeni pa samim time testovi koji ih koriste prijavljuju greške tokom kompajliranja. Da bi testovi mogli da se kompajliraju potrebno je napraviti izmene u njima. Ako se nakon tih izmena neki testovi neuspešno izvršavaju, postavlja se pitanje da li se neuspešno izvršavaju zato što je napravljena greška prilikom faze restrukturiranja ili prilikom izmena pri instanciranju objekata u testovima. Svaki put prilikom ovakve evolucije inženjer mora da prilagodi instanciranje objekata u potencijalno velikom broju testova. Neophodan napor i mogućnost nastanka dileme obeshrabruje pokušaje evoluiranja modula. Ukoliko modul ne evoluira, dolazi do gubitka jasnoće, jednostavnosti i narušavanja principa dobrog dizajna (SOLID i drugih). Moguće je umanjiti ovaj problem delegiranjem instanciranja objekata specijalizovanom modulu koji se naziva kontejner za injekciju zavisnosti. DI kontejner (engl. *dependency injection (DI) container*) rekurzivno konstruiše objekte zajedno sa objektima od kojih zavise i omogućava konfigurabilnost implementacionih detalja.

Delegiranjem kreiranja objekta klase preko čijeg javnog interfejsa se vrši testiranje modula se drastično smanjuje mogućnost neuspešnog izvršavanja testova usled izmena u grafu zavisnosti. Ukoliko se pri inicijalizaciji testa iskoriste iste konfiguracije DI kontejnera kao u aplikaciji verodostojnost istog raste jer bolje opisuje realnu upotrebu modula. TDD je izuzetno tesno uvezan sa osećajem samopouzdanja do mere da je ono apsolutno neophodno za fluidan razvoj. Iz potrebe smanjenja pojava lažno neuspešno izvršavajućih testova koji degradiraju samopouzdanje tokom razvoja proističe sledeći princip:

Korišćenje isključivo DI kontejnera za konstruisanje objekata pri testiranju vodi do smanjenja lažno neuspešno izvršenih testova, pojačavajući osećaj sigurnosti koji testovi omogućavaju tokom TDD ciklusa.

3.2. Princip kontinualne evolucije kolekcije testova

Inicijalni testovi koji se napišu tokom implementacije funkcionalnosti ne moraju biti konačni. Oni često nisu ni ispravni dugoročno posmatrano, a i nije nužno da budu. Njihov zadatak je da obezbede referentne tačke sigurnosti u toku iterativne evolucije aplikativnog koda. Samim time, postoji momenat kada oni bivaju uklonjeni ili evoluirani u testove većeg opsega ili striktinijih kriterijuma verifikacije.

Striktno praćenje pravila TDD metodologije dovodi praktikante do konstatnih zastoja, jer impliciraju da neki deo ponašanja mora biti unapred poznat u potpunosti pre pisanja testa u crvenoj fazi. Ovo često za posledicu ima pojavu koja je poznata pod nazivom paraliza analizom (engl. *analysis paralisys*). Inženjer troši dragoceno vreme da osmisli sve detalje, a za uzvrat često ne dolazi do dugoročno tačnih informacija. Kako ne bi došlo do ovoga, prvo treba prihvatiti da nijedan kvalitetno napisan test nije beznačajan, bez obzira na dugoročnu vrednost uslova verifikacije. Ideja je početi od onoga što je trenutno poznato. Provera da rezultat neke metode koja enkriptuje ulazni tekst nije prazan tekst je dobar primer. Nakon par TDD ciklusa ovaj i još par sličnih testova će se uspešno izvršiti pri čemu ih treba prepraviti uvođenjem striktinijih uslova verifikacije ili ih ukloniti. Nakon nekog broja ciklusa deo funkcionalnosti biva implementiran do te mere da pouzdano radi. Ovo i dalje nije poželjno stanje kolekcije testova. Neminovno postoji dosta testova nad manjim opsegom koji potvrđuju ista ponašanja kao i testovi nad većim opsegom, samo sa verodostojnijim kriterijumom verifikacije. Stoga, deo testova treba evoluirati u testove većeg opsega, a deo ukloniti, kako bi se

omogućilo samopouzdanje i manervabilnost za evoluiranje dizajna bez ograničenja koje uvode testovi nad manjim opsegom. Odnosno, pojedinačne testove treba tretirati kao večno privremene u cilju evolucije aplikativnog dizajna i kvaliteta cele kolekcije testova. U razvijanju softvera je jedina konstanta promena, pa odatle proizilazi sledeći princip:

Kontinualnom evolucijom kolekcije automatskih testova prilikom TDD ciklusa, vodeći se kriterijumom težnje ka verifikaciji nad većim opsegom, se omogućava evolucija aplikativnog dizajna nad većim opsegom.

3.3. Princip poreza na nasleđivanje

Objektno orijentisana paradigma definiše četiri glavna principa koji služe kao gradivni elementi kvalitetnog aplikativnog dizajna: apstrakcija, enkapsulacija, nasleđivanje i polimorfizam. Od ovih principa nasleđivanje je verovatno u najširoj upotrebi i ujedno najmanje shvaćeno [8]. U velikom broju slučajeva inženjeri koriste nasleđivanje u svrhu ponovne upotrebe koda. Pri tome podklase vremenom moraju da obezbede funkcionalnosti koje zahtevaju promene u ponašanju implementiranom u nadklasi. Promene u nadklasi se veoma često reflektuju na ostale podklase uzrokujući nepredvidive posledice. Pored toga nadklasa postaje kompleksnija vremenom, i samim time, skuplja za održavanje. Zloupotreba nasleđivanja na ovaj način narušava princip otvorenosti-zatvorenosti, kao i princip Liskov substitucije [7]. Na manje apstraktnom nivou važi da od 11 dizajn šablona ponašanja opšte poznatih u softverskoj industriji samo jedan (*template method*) koristi nasleđivanje kao glavni gradivni element postizanja željenog cilja [9]. Posmatrajući sve tipove veza u aplikativnom kodu nasleđivanje je ujedno najjača i najrigidnija.

Ovu grešku TDD adresira na proaktivan način. Naime, za svaku podklasu je potrebno napisati testove koji prolaze i kroz ponašanje nadklase. Više koda u nadklasi vodi do veće replikacije testova, što vodi do veće potrošnje resursa radi postizanja cilja visoke pokrivenosti. TDD na ovaj način efektivno, po svom dizajnu, uvodi porez na zloupotrebu nasleđivanja koji se oseti odmah. Ovo je velika prednost nad kasnijim pisanjem testova, jer podstiče tim da evaluira dizajn odluke pre nego što šteta postane velika. Posledice grešaka u dizajnu softvera se obično otkrivaju kada prođe znatno vreme od momenta donošenja istih, pa je ovaj blagovremeni ishod izuzetno poželjan i iz njega proizilazi sledeći princip:

Visok ciljani stepen pokrivenosti aplikativnog koda automatskim testovima u kontekstu TDD-a vodi do smanjenja štetne upotrebe nasleđivanja uvođenjem pragmatične potrebe za blagovremenom evaluacijom kvaliteta odluke upotrebe istog.

3.4. Princip određivanja opsega verifikacije jediničnih testova

Jedinični testovi verifikuju male segmente koda. Ne postoji opšte pravilo koje može da dovede do izbora opsega verifikacije koji neće ispoljiti nijednu negativnu posledicu. Odabir opsega verifikacije jediničnog testa je proces određivanja najoptimalnijeg za dati aplikativni kod u kontekstu potreba zainteresovanih strana. Intuitivno je da bi grupa klasa povezana kompozicijom predstavljala dobar opseg verifikacije jediničnog testa, dok bi klase povezane agregacijom bile dobri kandidati za imitiranje pri verifikaciji. Klase povezane kompozicijom trebaju po dizajnu da rešavaju konkretan problem na istom nivou apstrakcije, da nemaju veliku kompleksnost, da se sastoje iz relativno malo koda i da se ne oslanjaju na nestabilne konkretizacije. U TDD-u testovi prednjače dizajnu i implementaciji na ovom nivou granularnosti. Stoga, problem se svodi na identifikaciju vrste asocijacije u toku TDD ciklusa. Ukoliko međusobnom asocijacijom dve klase narušavaju princip jedne odgovornosti onda je ta asocijacija agregacija. Primenjujući princip jedne odgovornosti kao kriterijum za blagovremenu eliminaciju klasa povezanih agregacijom oblikuje se opseg verifikacije. Primenom ove strategije aktivno tokom TDD ciklusa smanjuje se vreme donošenja odluka kao što su gde smestiti i kako uvezati novo ponašanje sa ishodom poboljšanog dizajna, iz čega proističe sledeći princip:

Primenom principa jedne odgovornosti aktivno tokom TDD razvoja moguće je blagovremeno identifikovati segmente koda koji ne pripradaju opsegu verifikacije jediničnog testa, vodeći do jasno definisanog opsega verifikacije i boljeg dizajna.

3.5. Princip balansa verodostojnosti i uloženog napora

Dosadašnja istraživanja su pokazala da je razvoj praktikovanjem TDD-a čak do 35% sporiji [10, 11]. Pitanje se postavlja šta praktikant TDD metodologije može da uradi kako bi smanjio dodatan napor pri razvoju softvera. Strategija za optimizaciju uloženog napora sledi iz razumevanja doktrina dveju TDD škola. Starija je Detroit škola, čija je doktrina poznata po pristupu iz srži problema ka spoljašnosti. Primenom ove doktrine bi praktikant krenuo da razvija softver od internih do integracionih modula, od domenskih do ulazno izlaznih. Novija je London škola, čija je doktrina poznata po pristupu sa periferija problema ka srži. Primenom ove doktrine bi praktikant krenuo da razvija softver od integracionih do internih modula, od ulazno izlaznih do domenskih.

Dobra početna strategija je krenuti sa periferija ukoliko je u pitanju razvoj složenog softvera. Na ovaj način je inicijalna razlika napora pri praktikovanju TDD-a relativno mala. Ovo važi zato što se inicijalno pišu testovi koji verifikuju funkcionalnost dosta oskudno i čija je primarna svrha da otkriju da li je ideja za rešavanje problema obećavajuća ili ne. Doktrina London škole uvođenjem imitacija na više nivoa modulaziracije omogućava upravo to jer se korišćenjem savremenih alata imitiacije mogu brzo implementirati.

Ukoliko je ideja obećavajuća, može se preći na sledeći nivo (sledeći modul) gde opet treba evaluirati opštu ideju bez trošenja previše vremena. Ovaj proces se ponavlja sve do inicijalne implementacije funkcionalnosti sa imitacijama na određenim mestima. Potom treba implementirati detalje koji su ostali imitirani. Značajno je napomenuti da u ovom momentu većina slučajeva izršavanja toka funkcionalnosti nisu pokriveni ovim testovima, tj. da je verodostojnost testova veoma niska.

U daljoj implementaciji je potrebno pooštriti kriterijume verifikacije i evoluirati kolekciju testova nastalu u inicijalnoj fazi. Ukoliko su ti detalji deo integracionih modula, preferabilno je nastaviti implementaciju primenom doktrine London škole. Ukoliko su pak deo internih modula koji su sami po sebi složeni i u idealnom

slučaju ne zavise od drugih modula, preferabilno je primeniti doktrinu Detroit škole.

Primena TDD-a sa visokim ciljanim stepenom verodostojnosti je preferabilna ukoliko je poznato da je problem koji se rešava složen. Naravno, nisu svi problemi složeni i nekada je teško unapred proceniti koji jesu. Iz potrebe da se optimizuje napor uložen pri praktikovanju TDD metodologija proističe sledeći princip:

Pri razvoju vođenom testovima treba dovoljno često evaluirati složenost problema koji se rešava kako bi se donela odluka koja doktrina TDD-a je preferabilnija za rešavanje istog.

4. ZAKLJUČAK

Uticaj razvoja vođenog testovima na dizajn softvera je neosporiv samom činjenicom da podstiče inženjera da blagovremeno evaluira dizajn odluke. Dizajn softvera se idejno osmišljava uoči razvoja i iterativno poboljšava tokom životnog ciklusa softvera. Dakle, realizacija boljeg dizajna je moguća isključivo kroz mogućnost i ekonomičnost konstantne evolucije istog.

Principi predloženi u ovom radu predstavljaju dobre smernice koje teže da omoguće i olakšaju ekonomičnu evoluciju dizajna softvera, nadograđujući se na proces i beneficije TDD metodologije.

5. LITERATURA

[1] K. Beck, *Test-driven development: by example*. Boston: Addison-Wesley, 2003.

[2] L. Koskela, *Test driven: TDD and Acceptance TDD for Java* developers. Greenwich, CT: Manning, 2008.

[3] R. C. Martin, *The clean coder: a code of conduct for professional programmers*. Upper Saddle River, N.J.: Prentice Hall, 2011.

[4] B. George and L. Williams, "An initial investigation of test driven development in industry," in *Proceedings of the 2003 ACM symposium on Applied computing - SAC '03*, Melbourne, Florida, 2003, p. 1135, doi: 10.1145/952532.952753.

[5] N. Nagappan, E. M. Maximilien, T. Bhat, and L. Williams, "Realizing quality improvement through test driven development: results and experiences of four industrial teams," *Empir. Softw. Eng.*, vol. 13, no. 3, pp. 289–302, Jun. 2008, doi: 10.1007/s10664-008-9062-z.

[6] E. M. Maximilien and L. Williams, "Assessing test-driven development at IBM," in *25th International Conference on Software Engineering, 2003. Proceedings.*, Portland, OR, USA, 2003, pp. 564–569, doi: 10.1109/ICSE.2003.1201238.

[7] R. C. Martin, *Clean architecture: a craftsman's guide to software structure and design*. 2018.

[8] E. Tempero, J. Noble, and H. Melton, "How Do Java Programs Use Inheritance? An Empirical Study of Inheritance in Java Software," in *ECOOP 2008 – Object-Oriented Programming*, vol. 5142, J. Vitek, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 667–691.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: Elements of resuable object-oriented software*. England: Pearson education Limited, 1995.

[10] T. Bhat and N. Nagappan, "Evaluating the efficacy of test-driven development: industrial case studies," in *Proceedings of the 2006 ACM/IEEE international symposium on International symposium on empirical software engineering - ISESE '06*, Rio de Janeiro, Brazil, 2006, p. 356, doi: 10.1145/1159733.1159787.

[11] L. Huang and M. Holcombe, "Empirical investigation towards the effectiveness of Test First programming," *Inf. Softw. Technol.*, vol. 51, no. 1, pp. 182–194, Jan. 2009, doi: 10.1016/j.infsof.2008.03.007.

Kratka biografija:

Aleksandar Kahriman rođen je u Novom Sadu 1994. god. Diplomski rad na Fakultetu tehničkih nauka iz oblasti Računarstvo i automatika – Softversko inženjerstvo odbranio je 2017. god. Interesuju ga pragmatične strategije postizanja većeg nivoa kvaliteta softverskih sistema. Kontakt: aleksandar.kahriman@gmail.com