Zašto (ne) odabrati Feature Driven Development?

Pavlović, Marko Fabijan

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Marko Fabijan Pavlović

ZAŠTO (NE) ODABRATI FEATURE DRIVEN DEVELOPMENT?

DIPLOMSKI RAD

Varaždin, 2021.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Marko Fabijan Pavlović

Matični broj: 0016104578

Studij: Organizacija poslovnih sustava

ZAŠTO (NE) ODABRATI FEATURE DRIVEN DEVELOPMENT?

DIPLOMSKI RAD

Mentor:

Izv. prof. dr. sc. Zlatko Stapić

Varaždin, rujan 2021.

Marko Fabijan Pavlović

Izjava o izvornosti

Izjavljujem da je moj diplomski rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

_____

Sažetak

Feature-driven development (FDD), odnosno razvoj softvera navođen funkcionalnostima, metoda je koja se temelji na dvotjednim iteracijama koje omogućavaju kontrolu napretka. Za provođenje metode potrebna je podjela na uloge, a sačinjava je pet koraka od kojih posljednja dva predstavljaju jednu iteraciju. Glavne uloge u provođenju ove domene su projektni menadžer, glavni arhitekt, razvojni menadžer, glavni programeri, vlasnici klasa i eksperti za domene, dok se preostale uloge dijele na potporne i dodatne. Metoda je brzo stekla popularnost te se počela kombinirati s ostalim metodama i tehnikama, na temelju čega su nastale nove razvojne metodologije. Unatoč tome nedostaje kvalitetna softverska podrška za provođenje FDD-a. U radu su iznešene određene prednosti i nedostatci metode te odgovor na pitanje kada razvoj prema funkcionalnostima dolazi do izražaja. Primjer provođenja metode prikazan je u praktičnom dijelu rada, kroz razvoj aplikacije za vođenje evidencije zaposlenih studenata.

Ključne riječi: Feature-Driven Development, funkcionalnost, detaljan dizajn, izvještavanje napretka, vlasništvo nad klasom

Sadržaj

1. Uvod

Zadnju četvrtinu prošlog stoljeća obilježio je pojam digitalne revolucije, popularne i kao treća industrijska revolucija. Digitalna revolucija označava prijelaz iz analognih u digitalne tehnologije, konstantan porast korištenja stolnih i prijenosnih računala i mobitela, te povećanje ljudi koji koriste Internet.

Novo tisućljeće karakterizira četvrta industrijska revolucija, odnosno informatizacija u svim sferama društvenog života. Informacijsko-komunikacijske tehnologije više se ne odnose samo na računala i mobitele, već i na brojne suvremene tehnologije kao što su umjetna inteligencija, proširena stvarnost, virtualna stvarnost, Internet stvari, 3D printer, robotika i slično. Računala više nisu primarno sredstva komunikacije, nego i sredstva obavljanja poslova. Autopiloti u automobilima i udaljene asistencije prilikom medicinskih operacija samo su neki primjeri od prednosti četvrte industrijske revolucije, koju karakterizira brza izmjena u tehnologijama.

Sve što omogućava suvremena tehnologija, treba i proizvesti, gdje je pak lakši dio proizvodnja fizičkih komponenti, dok je onaj teži, zahtjevniji, i često skuplji prilikom pogreške, proizvodnja softvera. Razlozi koji dovedu do propasti projekta razvoja softvera mogu biti brojni: loša komunikacija ili nedostatak iste, neredovita kontrola napretka, nejasni i nedovoljno definirani zahtjevi, nedovoljno stručno osoblje, često i prevelika očekivanja od strane klijenta. Među navedenim uzrocima, najčešća su prva dva spomenuta, odnosno problemi u komunikaciji i praćenju napretka.

Jedna od metoda za razvoj softvera koja naglašava redovitu kontrolu napretka, te tako omogućava i pravovremenu reakciju ako projekt krene u krivom smjeru, je razvoj softvera pokretan funkcionalnostima, odnosno feature-driven development. Ova metoda provodi se kroz dvotjedne faze, gdje nakon svake faze slijedi recenzija dotad napravljenog dijela, čime se kontrolira uspješnost procesa razvoja projekta. U nastavku rada objašnjen je postupak provođenja metode, nakon čega su navedene glavne snage i slabosti metode, koje će čitateljima olakšati donošenje odluke o odabiru ove metode. Potom je u praktičnom dijelu rada prikazan primjer razvoja aplikacije ovom metodom korak po korak, odnosno funkcionalnost po funkcionalnost.

Tema rada odabrana je nakon sudjelovanja u timskom razvoju web aplikacije, gdje se zbog loše komunikacije i neredovite kontrole dogodilo da kôd jednog člana tima nije kompatibilan s kodom drugog člana, te je na usklađivanje potrošeno podosta vremena. Stoga je u ovoj metodi viđena prilika za rješavanje loših programerskih navika i stjecanje vještina koje

bi omogućile kvalitetniji pristup u planskom i programerskom dijelu razvoja softvera po završetku fakulteta. Također, kao nekome tko je pretežno koristio tradicionalni, vodopadni model razvoja softvera, ova metoda je novost te predstavlja izazov za provedbu u praktičnom dijelu.

2. Metode i tehnike rada

Rad je podijeljen na dva dijela: teorijski i praktični dio. Teorijski dio rada sastoji se od četiri poglavlja. U prvom poglavlju opisana je metoda feature-driven development (FDD), pri čemu su kao osnovna literatura korišteni prva knjiga u kojoj se spominje pojam FDD i glavni priručnik za FDD u kojemu je detaljno opisan postupak provođenja. Nad osnovnom literaturom provedena je metoda analize, kojom se feature-driven development kao cjelovit proces raščlanjuje na podprocese i korake aktivnosti, te je osnovnoj literaturi pridružen i priručnik za agilne metode razvoja softvera, kako bi se stekao kontekst FDD-a unutar agilnih metoda. Nakon analize, svaki korak je detaljno opisan metodom deskripcije.

Drugo poglavlje opisuje softversku podršku FDD-u, napisano je sintezom studentskog rada u kojemu se osmišljava plan za razvoj softvera za FDD i literature o prethodno izrađenim aplikacijama. Treće teorijsko poglavlje temeljeno je na proučavanju studentskih radova na temu nadogradnje FDD-a drugim programerskim konceptima i razvojnim metodama, dok četvrto, posljednje teorijsko poglavlje, iznosi dobre i loše strane FDD-a te tako predstavlja odgovor na naslovno pitanje. Napravljeno je sintezom prva tri poglavlja, uz komparaciju s nekim drugim metodama razvoja softvera.

Drugi, praktični dio rada, predstavlja izradu aplikacije provedenu po metodi feature-driven development, gdje se u praksi primjenjuje gradivo opisano u teorijskom dijelu. Koraci FDD procesa potkrijepljeni su slikama i isječcima programskog koda te opisom što je u aplikaciji napravljeno u tom koraku. U izradi aplikacije primijenjena su znanja prikupljena tokom studija, pretežno na kolegijima Programsko inženjerstvo (izrada aplikacije) i Teorija baza podataka (izrada baze podataka). Za izradu programskog rješenja odabran je programski jezik C# u razvojnom okviru MS Visual Studio, dok je za bazu podataka korišten MS SQL uz pripadajući alat Microsoft SQL Server Management Studio. Za verzioniranje projekta korišten je GitHub.

3. Feature-driven development

Feature-driven development (u nastavku FDD) agilna je metoda za razvoj softvera navođena funkcionalnostima. U ovom poglavlju najprije je opisana povijest nastanka metode, a potom i temelj ove metode, tj. pojam funkcionalnosti. Nakon što je naveden optimalan skup traženih vještina te podjela na tri tipa uloga u provođenju FDD-a, detaljno je opisan skup koraka i važnost pojedinih uloga u njima. Radi stjecanja uvida u područje kojim se FDD bavi, sljedeća slika prikazuje aktivnosti u razvoju softvera koje ulaze u domenu FDD-a:

Slika 1. Domena FDD-a (prema Palmer i Felsing, 2002.)

Iz slike se iščitava da se FDD ne bavi toliko financijskim i tehničkim stvarima i pripremom projekta, već je naglasak na samim funkcionalnostima sustava.

John Hunt (2006.) smatra FDD rješenjem za kontrolu kompleksnosti u agilnim projektima. Tvrdi da snaga FDD-a leži u tome što kao iterativni proces ostavlja prostor za promjene u planiranju, o čemu je više rečeno u poglavlju 3.6. Kako na početku svake iteracije postoje aktivnosti planiranja, trajanje određenih faza projekta nije unaprijed definirano kao kod tradicionalnih stilova. Hunt kao ključne ciljeve FDD-a navodi:

1. zadovoljiti klijenta,
2. isporučiti radni softver koji klijentu stvara vrijednost,
3. postaviti radni softver kao primarnu mjeru napretka,
4. promicati održivi razvoj,
5. svaki proces održati najjednostavnijim mogućim.

Općenito, ovaj pristup provodi se korištenjem metoda jednostavnih za razumijevanje i implementaciju te se kontinuirano prikazuju mjerljivi rezultati. Naglasak je na produktivnosti, ne u praćenju dokumentacije.

3.1. Povijest FDD-a

FDD nije unaprijed osmišljen kao metoda i planski testiran na projektima, već je razvijen kroz praksu, nastao je spontano. Početak razvoja softvera pokretanog funkcionalnostima seže u 1997. godiinu, kada je Jeff De Luca vodio projekt za singapursku banku United Overseas Bank (Chowdhury i Nazmul Huda, 2011.). Na projektu je kao glavnog modelera zaposlio Petera Coada, kod kojega je primjetio da funkcionalnosti detaljizira kroz sitniju, profinjeniju zrnatost, te je shvatio da funkcionalnostima treba posvetiti veću pozornost.

Kao glavni razlog zbog kojeg nije slijedio upute neke već opisane metode, De Luca je naveo kako niti jedna metoda ne daje odgovore na dva najvažnija pitanja u razvoju softvera: koliko daleko smo stigli i koliko je preostalo. Kako bi se mogao pratiti napredak, potrebno je u startu postaviti određenu informatičko-analitičku aktivnost koja uključuje postavljanje skale za mjerenje napretka.

Tokom projekta, De Luca i Coad shvatili su da je metoda koju provode učinkovita te su je odlučili detaljno opisati po koracima. Tako je sljedeće godine prvi put službeno spomenut izraz *feature-driven development*. De Luca je izjavio kako su za stvaranje FDD-a zaslužni njegovo iskustvo u IBM laboratorijima i Coadova sitna zrnatost (Roock, 2007.).

FDD je bio dobro prihvaćen među softverskim inženjerima i brzo je postao popularan. Steve Palmer, koji je na tom projektu radio kao voditelj razvoja, izjavio je da FDD vole klijenti, menadžeri i programeri:

- Klijenti ga vole jer im dostavlja stvarne rezultate i izvješća o napretku napisane načinom koji razumiju.
- Menadžeri ga vole jer im dostavlja cjelovitu i točnu sliku napretka i statusa projekta, što im predstavlja informacije potrebne za pravilno upravljanje projektom.
- Programeri ga vole jer rade u malim iteracijama, često koriste riječ „dovršeno“ i često dobivaju nove zadatke, što doprinosi dinamičnosti posla. Još jedan razlog je što im olakšava dostavu informacija menadžerima (Palmer i Felsing, 2002.).

Vidljivo je da je glavni razlog što se FDD svidio osobama uključenima u razvoj softvera dostava traženih podataka, odnosno izvješća o napretku za klijente i menadžere, što je u uvodu napomenuto kao ključna karakteristika metode.

3.2. Funkcionalnost i skupovi funkcionalnosti

Funkcionalnost se definira kao funkcija koja klijentu predstavlja vrijednost i čija implementacija traje najviše dva tjedna. Funkcionalnosti se prilikom modeliranja objekta domene iskazuju u sljedećem obliku:

<akcija> <rezultat> <objekt>

primjeri:

Izračunati	prosjek golova	po utakmici.
Obrisati	neaktivne korisnike	iz baze.
Očitati	potrošnju	memorije.

Srodne funkcionalnosti grupiraju se u setove funkcionalnosti, koji se iskazuju u identičnom obliku, osim što se akcija iskazuje kao glagolska imenica. Naprimjer, funkcionalnosti „učitati uplatu osiguranja“, „provjeriti tehničku ispravnost vozila“ i „naplatiti tehnički pregled“ za auto mogu se grupirati u set funkcionalnosti „polaganje tehničkog pregleda“.

Funkcionalnosti se neformalno popisuju prilikom razvoja modela cjelokupnog sustava, dok se detaljan popis funkcionalnosti sastavlja nakon što je model razvijen. Dodatne funkcionalnosti mogu se u model dodavati naknadno, tokom faza „dizajniraj po funkcionalnostima“ i „gradi po funkcionalnostima“ (Coad, Lefebvre i De Luca, 1999.).

Napomenuto je da implementacija funkcionalnosti traje maksimalno dva tjedna. Donje granice nema, što znači da se implementacija pojedinih funkcionalnosti mogu može dovršiti u nekoliko dana, ili čak nekoliko sati. Ukoliko se za neku funkcionalnost procijeni da dva tjedna nisu dovoljna za implementaciju, glavni problem razlaže se na potprobleme prema kojima se

određuju nove funkcionalnosti smanjenog opsega. Važnost poštivanja granice implementacije je u tome da se očuva jedna od glavnih odlika FDD-a, a to je kontinuirano mjerenje napretka, što omogućava klijentima učestalo davanje povratnih informacija.

3.2.1. Odabir funkcionalnosti za iteraciju

Odabir funkcionalnosti obično zajedno rade projektni menadžer i glavni programeri, uloge o kojima će biti više rečeno u nastavku. Prilikom odabira funkcionalnosti, potrebno je obratiti pažnju na sljedeće:

- Odabrane funkcionalnosti ne bi trebale ovisiti o funkcionalnostima koje još nisu implementirane.
- Odabrane funkcionalnosti smiju ovisiti samo o funkcionalnostima koje će se implementirati u istoj iteraciji.
- Kako bi se postupno mogle izdavati verzije programa, potrebno je odabrati funkcionalnosti koje čine cjelovit skup, što može zahtijevati koordinaciju svih glavnih programera u timu.
- Svaki glavni programer odabire funkcionalnosti relevantne za njegov podtim.

Klijenti također mogu utjecati na odabir funkcionalnosti za itereaciju ako imaju svoj isplanirani redoslijed implementacije funkcionalnosti (Khramtchenko, 2006.).

3.3. Potrebne vještine

Svaka metoda razvoja softvera ima svoju najbolju praksu, odnosno skup vještina potreban da bi se metoda izvela na optimalan način. Te vještine potrebne su i za druge metode, a ono što ih razlikuje je njihova kombinacija. FDD je relativno fleksibilan u tom pogledu, što znači da omogućava kombinaciju ljudi s različitom količinom iskustva u svakoj vještini, zbog čega je lako prilagodljiv u organizacijama (Palmer i Felsing, 2002.). Ipak, postoje određene vještine koje moraju biti obuhvaćene, stoga su navedene u nastavku.

3.3.1. Modeliranje objekta domene

Model (objekta) domene vizualni je prikaz konceptualnih klasa ili objekata iz stvarnog svijeta u domeni interesa. Služi kao opći okvir za dodavanje novih funkcija, te tako i za izradu funkcionalnosti. Usmjerava projektne timove u izradi početnog dizajna za određeni skup funkcionalnosti, čime se smanjuje broj potrebnih refaktoriranja klasa prilikom dodavanja funkcionalnosti, te pomaže u održavanju cjelokupnog integriteta sustava (Chursin, 2017.).

Modeliranje objekta domene aktivnost je koja uključuje izgradnju dijagrama klasa koji prikazuju tipove objekata domene i odnose među njima. Ustaljena praksa je dijagrame klasa potkrijepiti dijagramima stanja koji prikazuju međudjelovanje objekata.

3.3.2. Razvoj po funkcionalnostima

Pojam „usmjeren na funkcionalnosti“ označava razvojni proces koji u svrhu planiranja kombinira izražavanje zahtjeva i jedinice aktivnosti. Funkcionalnosti su usko povezane s dijagramima slučajeva korištenja (eng. *use-case*). (Hunt, 2006.)

Ključni element svakog projekta je izjava o svrsi, odnosno o razlogu postojanja sustava, koja sadrži funkcionalne zahtjeve koje novozigrađeni sustav mora zadovoljiti da bi se projekt smatrao uspješnim. Kako funkcionalni zahtjevi imaju tendenciju miješanja korisničkog sučelja, pohrane podataka i mrežnih komunikacijskih funkcija s poslovnim funkcijama, programeri često veliku količinu vremena posvećuju tehničkim značajkama na štetu poslovnih značajki. Stoga je dobra praksa funkcionalne zahtjeve ograničiti na one vrijedne za korisnika ili klijenta, i koji se, kako je spomenuto u poglavlju 3.2, nazivaju funkcionalnosti. (Palmer, Felsing, 2002.)

3.3.3. Vlasništvo nad klasom/kodom

Palmer i Felsing (2002.) su vlasništvo nad klasom/kodom definirali kao pojam u razvojnom procesu koji označava koja je osoba ili uloga odgovorna za sadržaj klase ili dijela koda. Vlasništvo nad klasom ili kodom moguće je ostvariti na dva načina: kao individualno vlasništvo ili kao kolektivno vlasništvo.

Individualno vlasništvo, kako i sam naziv govori, način je u kojem svaki dio koda ima samo jednog vlasnika. Ovaj način prihvaćen je u objektno-orijentiranom programiranju, gdje se na klasu gleda kao najmanju jedinicu koda te jedna klasa predstavlja jedan entitet. Nedostatak ovoga pristupa očitava se prilikom naknadne promjene izvornog koda; u slučaju da jedan programer mora izmijeniti svoj kôd koji ovisi o klasi koja je u vlaništvu drugog programera, stvara se nepotrebno čekanje programera i nemogućnost da nastavi s radom dok drugi programer ne obavi svoje izmjene, što utječe na kašnjenje cjelokupnog projekta. Također postoji rizik da vlasnik klase napusti projekt, te ostatku tima bude potrebno dogledno vrijeme da shvati funkcioniranje klase, što također produljuje projekt.

U kolektivnom vlasništvu odgovornost za kôd, pa tako i za klasu, snosi cjelokupan tim. Svi članovi tima su ravnopravni, niti jedan pojedinac nije iznad ostatka tima u polaganju prava na kôd, te svi zajedno rade kako bi se potigao kvalitetan rezultat (Karanth, 2016.).

Kolektivno vlasništvo ima brojne prednosti, kao što su dijeljenje znanja između članova tima te ujednačen stil kodiranja, što pridonosi boljoj čitljivosti koda, kao i činjenica da dovršenost koda ne ovisi o jednom programeru (Karanth, 2016.). Nedostatak ovog pristupa je moguća situacija bez vlasništva, odnosno nitko ne želi preuzeti odgovornost za kôd te tako osigurati kvalitetu napisanog koda. Također je moguće da se nekoliko dominantnih pojedinaca u timu izdvoji i preuzme vlasništvo, što dovodi do njihovog preopterećenja u projektu, a ujedno i demotivira ostale članove tima (Palmer i Felsing, 2002.).

Dok se u ekstremnom programiranju preferira kolektivno vlasništvo, za FDD je uobičajeno individualno.

3.3.4. Timovi za funkcionalnosti

Kako su vlasnici dodijeljeni klasama, istu stvar potrebno je napraviti i funkcionalnostima. Vlasnici funkcionalnosti odgovorni su za njihov dovršetak te formiraju tim programera koji će raditi na funkcionalnosti. Kako bi se postiglo da voditelj tima uvijek u svojem timu ima vlasnike potrebnih klasa, optimalno je mijenjati članove tima za svaku novu funkcionalnost (Palmer i Felsing, 2002.).

U timovima za funkcionalnost članovima nisu izravno dodijeljene uloge, npr. programer, tester ili analitičar, već su svi titulirani jednostavno kao članovi tima. Za uspješan tim potrebno je postići sklad između osnaživanja, odgovornosti, identiteta, konsenzusa i ravnoteže:

- osnaživanje – stručnjacima iz određenog područja dopustiti kontrolu nad svojim područjem,
- odgovornost – međusobno dijeljenje kritičkih zapažanja, prenošenje percepcije na ostatak tima,
- identitet – članovi tima identiiciraju se s dijelom proizvoda, ne s usko specijaliziranom vještinom,
- konsenzus – neophodan je određeni stupanj otvorenosti, uključujući reorganizaciju, preraspodjelu resursa i izmjenu rasporeda prihvaćenu od članova tima,
- ravnoteža – različiti skupovi vještina, različiti zadaci, različita gledišta.

Smisao takvog univerzalnog članstva u timu je da se specijalizirani članovi ne bi striktno držali svojeg primarnog područja, nego bi mogli pomagati i u manje dominantnim područjima kako bi se posao prije dovršio, ali i radi dijeljenja vještina među članovima (Larman i Vodde, 2008.).

3.3.5. Inspekcije

Kako bi se osigurala kvaliteta koda, FDD se u velikoj mjeri oslanja na inspekcije. FDD ima dvije vrste inspekcija. Prva vrsta, pregled dizajna, služi kako bi se vidjelo uklapa li se odabrani dizajn u zamisli članova tima. Ako je odgovor negativan, dizajn se briše ili mijenja dok tim ne bude zadovoljan odabranim rješenjem. Druga vrsta pregleda je inspekcija koda. Kôd je odabran za inspekciju i provjerava se pridržava li se standarda kodiranja, potencijalnih grešaka ili logičkih pogrešaka. Inspekcije su također velike mogućnosti učenja, osobito ako se vaši timovi sastoje od mješavine starijih i mlađih programera. Oni mogu biti izvrsna prilika za razmjenu znanja u cijelom timu, kao i pomoć u identificiranju potencijalnih problema prije nego što se dogode (Cause, 2004.).

Primarna svrha inspekcija druge vrste je otkrivanje pogreške u kodu, dok se prednost očitava kroz još dvije činjenice. Prijenos znanja putem inspekcija omogućava širenje razvojne kulture i stjecanje iskustava; pregledom koda iskusnih programera te njihovim uvidom u vlastiti kôd, manje iskusni programeri stječu bolje prakse kodiranja. Inspekcijama se također postiže usklađenost sa standardom; ukoliko su programeri svjesni da njihov kôd neće proći provjeru ako ne odgovara standardima dizajna i kodiranja, za pretpostaviti je da će uskladiti kôd sa standardima.

Inspekcije se obavljaju na način da programerima ne stvaraju strah od neugodnosti ili poniženja. Treba ih smatrati izvrsnim alatom za otkrivanje pogrešaka te prilikom za međusobno učenje, a ne osobnim pregledom uspješnosti (Palmer i Felsing, 2002.).

3.3.6. Redoviti raspored izgradnje

U redovitim intervalima uzima se dovršeni kôd s pripadajućim bibliotekama i komponentama te se gradi sustav koji se demonstrira klijentu. Funkcionalnosti koje sustav obavlja moraju imati za klijenta uočljivu vrijednost. Redovita izgradnja omogućava rano isticanje integracijskih grešaka te osigurava postojanost ažuriranog sustava za demonstraciju klijentu (Palmer i Felsing, 2002.).

3.3.7. Upravljanje konfiguracijom

Kako bi se sačuvala povijest promjena, potrebno je, uz izvorni kôd, verzionirati i dokumente o zahtjevima, kao i testne slučajeve i skripte. Zahtjevi prema sustavu za upravljanje konfiguracijom ovise o složenosti projekta (Palmer i Felsing, 2002.).

3.3.8. Izvještavanje o vidljivosti rezultata

Menadžerima i voditeljima tima za ispravno usmjeravanje projekta nužno je biti precizno upućen u trenutni status projekta te imati sliku o tome koliko razvojnom timu treba za dodavanje nove funkcionalnosti. FDD omogućava prikupljanje točnih i pouzdanih informacija o statusu i predlaže niz jednostavnih, intuitivnih formata za izvještavanje o napretku svih uloga uključenih u projekt (Palmer i Felsing, 2002.).

3.4. Uloge u FDD-u

Kako bi se ušlo u dubinu samog FDD-a, potrebno je najprije objasniti uloge koje sudjeluju u procesu razvoja. Uloge su podijeljene u tri skupine: glavne, sporedne ili potporne te dodatne. U sljedećoj tablici prikazana je njihova podjela:

Tablica 1. Uloge u FDD

Glavne uloge	Potporne uloge	Dodatne uloge
projektni menadžer	menadžer domene	testeri
glavni arhitekt	menadžer distribucije	deployeri
razvojni menadžer	jezični guru	tehnički pisci
glavni programeri	konstrukcijski inženjer	
vlasnici klasa	alatničar	
eksperti za domene	administrator sustava	

(Izvor: Palmer i Felsing, 2002.)

U nastavku su za svaka ulogu iz tablice objašnjeni zadaci koje ta uloga obavlja u okviru FDD procesa, pri čemu su najdetaljnije objašnjene glavne uloge. Također je za svaku ulogu naveden i originalni naziv na engleskom jeziku.

3.4.1. Glavne uloge

Projektni menadžer (*Project Manager*) administrativni je voditelj projekta, čiji zadaci obuhvaćaju upravljanje projektnim budžetom, određivanje potrebnog broja zaposlenika na projektu te upravljanje prostorom, opremom i resursima. Dužnost mu je stvoriti okolinu u kojoj će radni tim raditi najbolje, i održavati je takvom; članove tima ne treba prisiliti na djelovanje, već im omogućiti djelovanje.

Glavni arhitekt (*Chief Architect*) odgovoran je za sveukupni dizajn sustava. Vodi sjednice na kojima tim surađuje u dizajniranju sustava te ima završnu riječ na tim sjednicama. Nužne su mu dobre tehničke i modelerske vještine.

Razvojni menadžer (*Development Manager*) odgovoran je za vođenje svakodnevnih razvojnih aktivnosti. Također se bavi rješavanjem svakodnevnih sukoba za resurse kada ih glavni programeri ne mogu riješiti međusobno. U nekim se projektima ta uloga kombinira s ulogom glavnog arhitekta ili voditelja projekta. Voditelj razvoja ima glavnu riječ o sukobima s izvorima koji razvijaju resurse unutar projekta i usmjerava projekt kroz potencijalne zastoje u resursima.

Glavni programeri (*Chief Programmers*) su developeri s iskustvom razvoja softvera, prošli su kroz cijeli životni ciklus razvoja softvera. Vode male timove od tri do šest programera kroz analizu, dizajn i razvoj novih značajki softverska. Glavni programeri također rade s drugim glavnim programerima za rješavanje svakodnevnih tehničkih problema i resursa. Samoinicijativni za postizanje kvalitetnih rezultata na vrijeme, kombiniraju veliku tehničku sposobnost za postizanje rezultata svakih nekoliko dana. Glavni programeri obično žive posao koji obavljaju, odlučni su u tome da tim uspije te ih suradnici i programeri uvažavaju.

Vlasnici klasa (*Class Owners*) su developeri koji rade kao članovi malih razvojnih timova pod vodstvom Chief Programmera i koji osmišljavaju, kodiraju, testiraju i dokumentiraju značajke koje zahtijeva novi softverski sustav. Vlasnici klasa su ili talentirani developeri/programeri s nedovoljno iskustva da postanu Chief Programmers, ili pak vrhunski programeri koji se žele posvetiti isključivo programiranju, odnosno ne žele rukovodeće uloge.

Stručnjaci ili eksperti za domene (*Domain Experts*) koriste duboko poslovno znanje kako bi detaljno objasnili programerima zadatke koje sustav mora obavljati, predstavljaju bazu znanja na koju se programeri oslanjaju. Njihovo znanje i sudjelovanje apsolutno su presudni za uspjeh izgrađenog sustava. Trebaju dobre verbalne, pisane i prezentacijske vještine, moraju biti beskonačno strpljivi s programerima te osjećati entuzijazam dok govore o novom sustavu. Eksperti za domene mogu biti korisnici, klijenti, sponzori, poslovni analitičari ili bilo koja njihova kombinacija (Palmer i Felsing, 2002.).

3.4.2. Potporne uloge

Menadžer domene (*Domain Manager*) vodi stručnjake za domene, kako bi eksperti usuglasili mišljenje prilikom izrade dizajna sustava. U malom projektu ova se uloga često kombinira s onom uloge voditelja projekta.

Menadžer distribucije (*Distribution Manager*) osigurava da glavni programeri na tjednoj bazi izvještavaju o napretku. Oni su temeljiti i osiguravaju da su svi planirani i stvarni datumi ispravno uneseni te da se grafikoni ispravno ispisuju i distribuiraju. Upravitelj izdanja izravno izvještava voditelja projekata. Podnose izvještaje direktno projektnom menadžeru, te se ova uloga može kombinirati s ulogom administrativnog pomoćnika projektnog menadžera.

Jezični guru (*Language Lawyer/Language Guru*) osoba je odgovorna za poznavanje programskog jezika ili određene tehnologije. Ova uloga posebno se ističe kod prvog korištenja programskog jezika ili tehnologije. Često ga igra konzultant doveden u tu svrhu, i uloga se može postupno smanjivati dok u potpunosti ne nestane potreba za konzultantom.

Konstrukcijski inženjer (*Build Engineer*) odgovoran je za postavljanje, održavanje i pokretanje redovnog postupka *build*-a, što uključuje upravljanje sustavom za verzioniranje, generiranje izvještaja ili dokumentacije i pisanje skripti za izgradnju ili implementaciju.

Alatničar (*Toolsmith*) stvara male razvojne alate za razvojni tim, testni tim i tim za upravljanje podacima. Ukoliko je potrebno, to može uključivati postavljanje i upravljanje bazom podataka i web mjestom koje djeluje kao spremište znanja tima. Ovu ulogu može igrati svježe diplomirani ili mlađi programer.

Administrator sustava (*System Administrator*) postavlja poslužitelj na kojem radi projektni tim, konfigurira ga upravlja njime te rješava probleme s povezivanjem. To uključuje razvojno okruženje i sva specijalizirana testna okruženja. Administrator sustava je također uključen u početnu implementaciju sustava u proizvodnju. Kod složenijih sustava ova uloga može se podijeliti na serverskog administratora, mrežnog administratora i administratora baze podataka (Palmer i Felsing, 2002.).

3.4.3. Dodatne uloge

Testeri (*Testers*) provjeravaju ispunjava li sustav zahtjeve korisnika i obavlja li pravilno te zahtjeve. Mogu biti dio projektnog tima ili dio neovisnog odjela za osiguranje kvalitete.

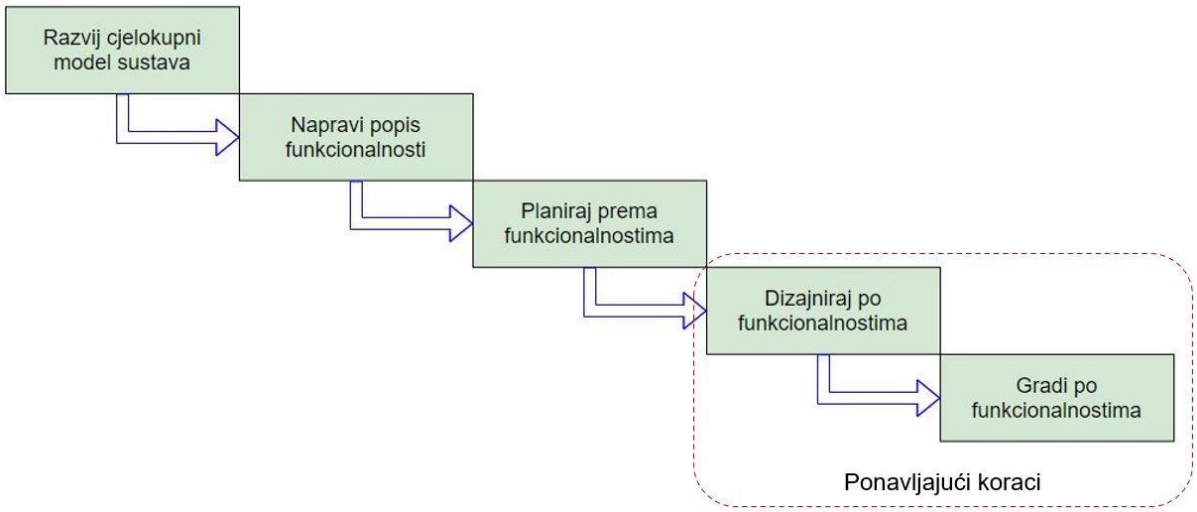
Deployeri (*Deployers*) pretvaraju postojeće podatke u nove formate koje zahtijeva sustav i rade na fizičkoj implementaciji novih izdanja sustava. Tim za implementaciju može biti dio projektnog tima ili dio neke vrste odjela za upravljanje i upravljanje sustavom.

Tehnički pisci (*Technical Writers*) pišu korisničku dokumentaciju i pripremaju je za *online* čitanje ili ispis. U nekim organizacijama imaju vlastiti odjel koji će služiti svim projektima (Palmer i Felsing, 2002.).

3.5. Koraci u FDD

Hunt (2005.) napominje da je u većini formalnih razvojnih procesa naglasak na fazama kroz koje projekt prolazi, a zatim na koracima unutar faza koji se mogu ili ne moraju iterativno provoditi. To se odnosi i na vodopadne i na iterativne procese. FDD započinje vodopadno, nakon čega se krajnji koraci odvijaju u iteracijama.

FDD započinje modeliranjem cjelokupnog sustava, nakon čega se nastavlja u dvotjednim iteracijama koje uključuju dizajn i izradu funkcionalnosti (Coad i sur., 1999.). U prvom koraku, članovi domene i razvojnog tima rade zajedno pod vodstvom glavnog arhitekta. Članovi domene prolaze kroz opseg i kontekst cjelokupnog sustava. Nakon toga se ponovno prolazi kroz kontekst, ali s više detalja za svako područje problemske domene, te se po svakom tom prolasku izrađuju objektni modeli. Potom se prema područjima domene funkcionalnosti grupiraju u skupove, koji obično predstavljaju određenu poslovnu aktivnost. Svaki proces na kraju prolazi verifikaciju (Palmer i Felsing, 2002.). Detaljan redoslijed koraka prikazan je na sljedećoj slici:



Slika 2. Koraci u FDD (prema Coad i sur., 1999.)

3.5.1. Razvij cjelokupni model sustava

Prvi korak u FDD-u započinje početnom verzijom popisa zahtjeva dostavljenom od strane naručitelja projekta. Članovi domene prolaze kroz opseg cjelokupnog sustava, naglašavajući samo važne stavke. Članovi domene se pod vodstvom vrhovnog arhitekta dijele u manje podtimove, u kojima razvijaju kostur projekta kao temelj budućeg rada, te se na kraju predstavljaju rezultati podtimova i spajaju se u zajednički model, prilagođavajući oblik modela.

Postupak:

1. Projektni menadžment uspostavlja tim za modeliranje.
2. Tim za modeliranje prolazi kroz domenu projekta pomoću tutoriala koji obuhvaća područje koje je potrebno modelirati, i koji obuhvaća područje teme u ponešto širem opsegu nego je potrebno za sustav. Ukoliko su dostupni, proučavaju se popratni dokumenti: komponente modela, funkcionalnosti zahtjeva, podaci modela i korisničke upute.
3. Vrhovni arhitekt i glavni programeri izrađuju neformalni popis funkcionalnosti.
4. Tim za modeliranje dijeli se u podtimove, gdje svaki podtim izrađuje dijagram klasa za razmatranu domenu, također skicira jedan ili više neformalnih dijagrama slijeda.
5. Svaki podtim predlaže svoj model, nakon čega se odabire jedan od modela kao osnovica te mu se pridružuje sadržaj iz ostalih modela. Taj model predstavlja model tima za modeliranje u cjelini.
6. Zapisuju se alternative modela.

Verifikacija ovoga koraka provodi se na način da članovi domene pružaju unutarnju samoprocjenu, dok se vanjska samoprocjena provodi po potrebi radi boljeg razumijevanja domene, potreba funkcionalnosti i opsega. Korak se završava se predajom dijagrama klasa vrhovnom arhitektu. Dijagramom klasa uspostavlja se oblik modela, metode u klasama predstavljaju osnovu za izradu popisa funkcionalnosti. Također se dostavljaju neformalni dijagrami slijeda, neformalni popis funkcionalnosti i bilješke o alternativnim modelima (Coad i sur., 1999.).

3.5.2. Napravi popis funkcionalnosti

Ovaj korak nastupa nakon što je razvijen početni model cjelokupnog sustava. Glavni programeri iz prethodnog koraka formiraju se u tim koji domenu dijeli na brojne aktivnosti, od kojih svaka predstavlja skup funkcionalnosti, a svaki korak unutar aktivnosti je funkcionalnost. Tim identificira funkcionalnosti te ih grupira po prioritetu i hijerarhiji te je rezultat kategorizirani popis funkcionalnosti (Palmer i Felsing, 2002.).

Postupak:

1. Projekni menadžer i razvojni menadžer formiraju tim za izradu popisa funkcionalnosti.
2. Taj tim identificira funkcionalnosti i formira skupove funkcionalnosti na temelju neformalnog popisa funkcionalnosti iz prethodnog koraka.
3. Funkcionalostima, ali i skupovima funkcionalnosti, dodijeljuje se prioritet u 4 razine:
   A – sustav obavezno mora imati funkcionalnost/skup,
   B – poželjno je da funkcionalnost/skup postoji u sustavu,
   C – napraviti ako bude dovoljno vremena,

D – ostaviti za budućnost (ažuriranja ili nadogradnje).

4. Kompleksne funkcionalnosti, za koje je utvrđeno da se ne mogu dovršiti unutar dva tjedna, dijele se na više manjih funkcionalnosti za koje je to izvedivo.

Internu verifikaciju koraka provode članovi tima, a korak završava predajom detaljnog popisa funkcionalnosti grupiranih u skupove funkcionalnosti razvojnom menadžeru i vrhovnom arhitektu na pregled (Palmer i Felsing, 2002.).

3.5.3. Planiraj prema funkcionalnostima

Projektni i razvojni menadžer te glavni programeri uspostavljaju ključne točke za iteracije u koracima „dizajniraj po funkcionalnostima“ i „gradi po funkcionalnostima“ na temelju hijerarhijskih, prioritetnih i težinskim popisa funkcionalnosti (Coad i sur., 1999.).

Izrađuje se redoslijed implementacije funkcionalnosti. Tipičan scenarij je najprije uzeti u obzir razvojni slijed, potom dodjelu skupova funkcionalnosti glavnim programerima i potom odrediti koje ključne klase dodijeliti kojim programerima. Vlasništvo klasa kompletirano je kada se postigne ravnoteža u razvojnom slijedu i dodjeli skupova funkcionalnosti glavnim programerima (Palmer i Felsing, 2002.).

Postupak:

1. Formira se tim za planiranje u kojemu su projektni i razvojni menadžer te glavni programeri.
2. Tim za planiranje određuje redoslijed razvoja, postavlja inicijalne datume završetka za skupove funkcionalnosti.
3. Sukladno redoslijedu razvoja i težinskoj vrijednosti funkcionalnosti, klasama se dodjeljuju vlasnici, a funkcionalnosti i skupovi funkcionalnosti raspodjeljuju se po glavnim programerima.

Verifikaciju provodi tim za planiranje preko unutarnje samoprocjene, dok se po potrebi vanjska procjena odvija s višim menadžmentom na način da se programerima pruži prilika da ocijene plan i postigne ravnoteža. Prije dizaniranja funkcionalnosti, tim za planiranje mora izraditi plan razvoja kojeg pregledavaju i moraju odobriti razvojni menadžer i vrhovni arhitekt (Coad i sur., 1999.).

3.5.4. Dizajniraj po funkcionalnostima

Često je slučaj da se male grupe funkcionalnosti planiraju istovremeno, pri čemu je uobičajeno da glavni programer odabere one funkcionalnosti koje koriste zajedničke klase, i te funkcionalnosti tvore radni paket glavnog programera. Glavni programer potom formira jedan ili više timova za izradu funkcionalnosti, u koje uključuje vlasnike potrebnih klasa. Svaki

tim izrađuje dijagram slijeda za svoju funkcionalnost, te glavni programer na temelju tih dijagrama izrađuje objektni model (Palmer i Felsing, 2002.).

Postupak:

1. Glavni programer identificira klase koje potrebne za izradu odabrane funkcionalnosti te skupa s vlasnicima klasa pokreće dizajn funkcionalnosti. Ukoliko je potrebno, kontaktira člana domene za pomoć u dizajniranju.
2. Tim za izradu funkcionalnosti izrađuje formalan, detaljan dijagram slijeda za funkcionalnost, pri čemu zapisuje alternative, odluke i bilješke vezane za dizajn. Glavni programer dodaje dijagram slijeda projektnom modelu.
3. Svaki vlasnik klase ažurira svoje prologe klase i metode za svoje metode u dijagramu slijeda. Uključuje tipove parametara, tipove povratnih vrijednosti, iznimke i poruke koje se šalju.
4. Tim za izradu funkcionalnosti provodi inspekciju dizajna.

Verifikacija ovog koraka je inspekcija dizajna; po završetku postupka, tim za izradu funkcionalnosti obavlja unutarnju samoprocjenu prolaskom kroz vlastite dijagrame slijeda, dok se vanjska procjena odvija prema potrebi radi pojašnjavanja potreba i opsega funkcionalnosti. Tim mora vrhovnom arhitektu na odobrenje dostaviti funkcionalnost i popratne dokumente, detaljan dijagram slijeda, ažuriranja dijagrama klasa, ažuriranja prologa klasa i metoda te bilješke o timskom razmatranju alternativa u dizajnu (Coad i sur., 1999.).

3.5.5. Gradi po funkcionalnostima

Prilikom svake iteracije, za ovaj korak potrebno je prethodno dovršiti dizajn funkcionalnosti. Svaki vlasnik klase kreira svoje metode za funkcionalnost koja se trenutno implementira te provodi testiranje na razini klase. Nakon uspješnog testiranja, kôd se promovira u izgradnju (eng. *build*).

Postupak:

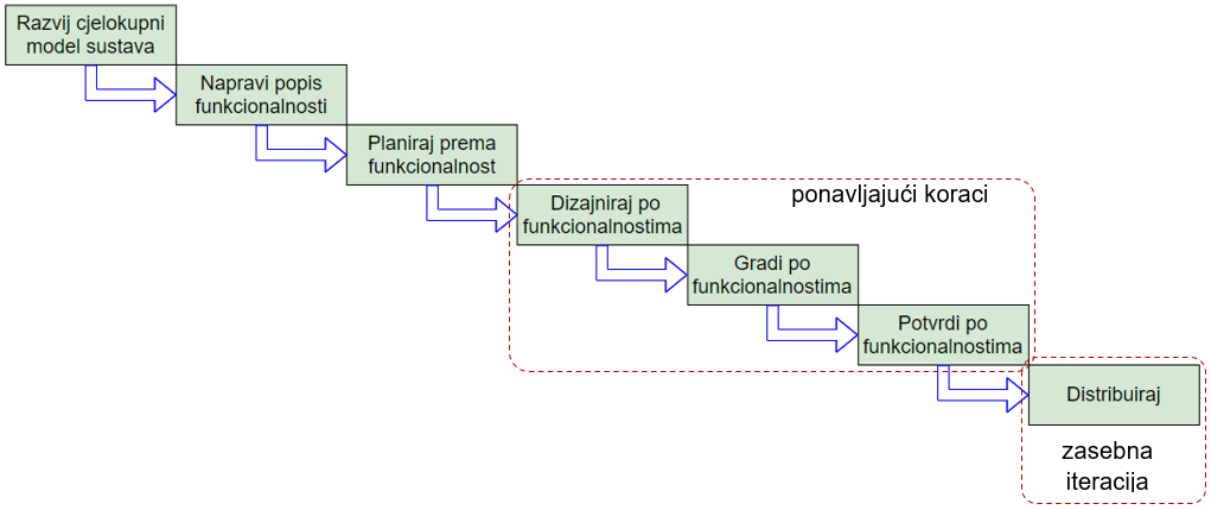
1. Sukladno dijagramu slijeda iz prethodnog koraka, implementiraju se klase i metode.
2. Glavni programer određuje inspekciju kôda koju provodi tim za izradu funkcionalnosti. U inspekciju se po potrebi mogu uključiti i vanjski sudionici.
3. Zapisničar bilježi stavke inspekcije kôda kako bi vlasnici klasa mogli pratiti inspekciju.
4. Svaki vlasnik klase testira svoj kôd i njegovu podršku za funkcionalnost. Glavni programer integrira cijelu funkcionalnost i provodi testiranje funkcionalnosti od početka do kraja.
5. Nakon uspješne implementacije, inspekcije i testiranja, klase se prijavljuju u sustav upravljanja konfiguracijom. Nakon što su prijavljene sve klase koje se nalaze na popisu klasa za funkcionalnost, glavni programer označava funcionalnost kao dovršenu.

Verifikaciju provodi tim za izradu funkcionalnosti, koji obavlja inspekciju koda i provodi jedinične testove, a zapisničar bilježi akcijske stavke za svakog vlasnika klasa. Tim glavnom programeru mora dostaviti implementirane i provjerene metode i testne metode, rezultate jediničnog testiranja za svaku metodu i za cjelokupni slijed, klase koje je prijavio vlasnik klase te gotovu funkcionalnost (Palmer i Felsing, 2002.).

Ovaj set koraka predstavlja osnovnu FDD metodologiju, a svaki od opisanih koraka bit će dodatno objašnjen u praktičnom dijelu rada, gdje je prikazana izrada projekta prema ovim koracima. U nastavku slijedi proširenje osnovnog seta koraka napravljeno od strane kompanije Cognizant.

3.6. Cognizantova verzija FDD-a

Američka tehnološka kompanija Cognizant razvila je svoju verziju FDD-a, proširenu s još dva procesa. Verzija je bazirana na temelju prethodnih iskustava kompanije u implementaciji kompleksnih poslovnih rješenja (Microsoft, 2021.). Standardnih pet procesa FDD-a prošireno je s dodatna dva procesa: potvrdu funkcionalnosti i distribuciju, koja predstavlja završetak razvojnog procesa. Sljedeća slika prikazuje dodane korake u odnosu na standardne korake prikazane na slici 2:



Slika 3. Procesi u Cognizantovoj verziji FDD-a (autorska izrada)

Vidljivo je da je potvrđivanje funkcionalnosti dio iterativnog ciklusa, odnosno provodi se svaki put po dovršetku funkcionalnosti. Tester zadužen za osiguranje kvalitete (eng. *Quality Assurance*) potvrđuje uspješnost funkcionalnosti, nefunkcionalne zahtjeve i usklađenost s prethodno dovršenim funkcionalnostima. Korak distribucije, iako nije dio iterativnog ciklusa,

može se izvesti više puta, ovisno o projektnom planu. Nekoliko dovršenih funkcionalnosti grupira se i postaje spremno za distribuciju.

Osim glavnih koraka, odnosno procesa, postoje i potporni procesi koje pokreće klijent: ispravljanje grešaka (eng. *bugs*) i promjena zahtjeva. Prilikom potrebe za tim koracima, vraća se na korak dizajna funkcionalnosti kako bi se utvrdilo zbog čega dolazi do greške, odnosno izmijenio dizajn radi prilagođavanja novim zahtjevima (Shindhe, 2007.).

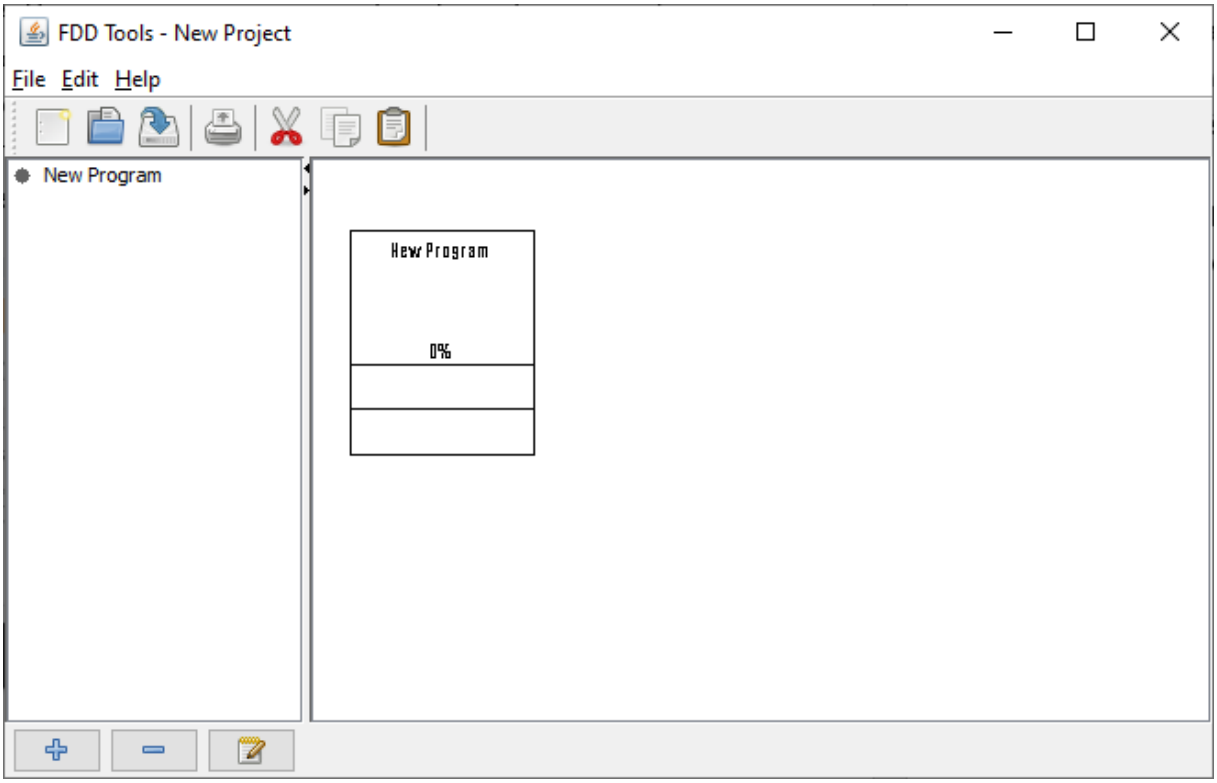
Cognizant je svoju verziju u suradnji s Microsoftom implementirao u alat za MS Visual Studio, o čemu će više riječi biti kasnije, u poglavlju o softverskoj podršci FDD-u.

4. Softverska podrška za FDD

Nedugo nakon pojave FDD-a, sredinom 2000-ih, počeli su se javljati programi koji podržavaju planiranje projekta prema funkcionalnostima, odnosno koji provode FDD metodologiju. Studenti na Fakultetu informacijskih tehnologija u Brnu, Marek Rychlý i Pavlína Tichá, napravili su usporedbu dotad postojećih aplikacija te razvili plan vlastitog softvera za FDD metodologiju. Istraživanje su proveli 2007. godine, stoga neke aplikacije više nije moguće preuzeti. Konkretno, od četiri aplikacije, samo je jedna dostupna za preuzimanje.

4.1. FDD Tools Project – jedini dostupan alat

FDD Tools Project aplikacija je otvorenog koda (eng. *open-source*) bazirana na Javi i namijenjena za stolna računala. Aplikacija je namijenjena za jednog korisnika i omogućuje grupiranje funkcionalnosti u skupove funkcionalnosti i projektna područja, a napredak se prikazuje na interaktivnom dijagramu projektnog parka. Prema podacima na službenoj stranici http://fddtools.sourceforge.net/, posljednje ažuriranje programa izašlo je 2009. godine. Sljedeća slika prikazuje početni ekran aplikacije FDD Tools s praznim projektom:



Slika 4. Početni zaslon programa FDD Tools (autorska izrada)

Na prethodnoj slici prikazana je aplikacija pokrenuta na operacijskom sustavu Windows 10, no nije konkretno vezana za operacijski sustav Windows, može se pokrenuti na svim operacijskim sustavima koji podržavaju Javu. Na sljedećoj slici vidi se aplikacija pokrenuta na macOS-u koja prikazuje projekt blizu završetka. Za svaku funkcionalnost prikazan je postotak dovršenosti:

Slika 5. Projekt s nekoliko funkcionalnosti u FDD Toolsu (Izvor: http://fddtools.sourceforge.net/)

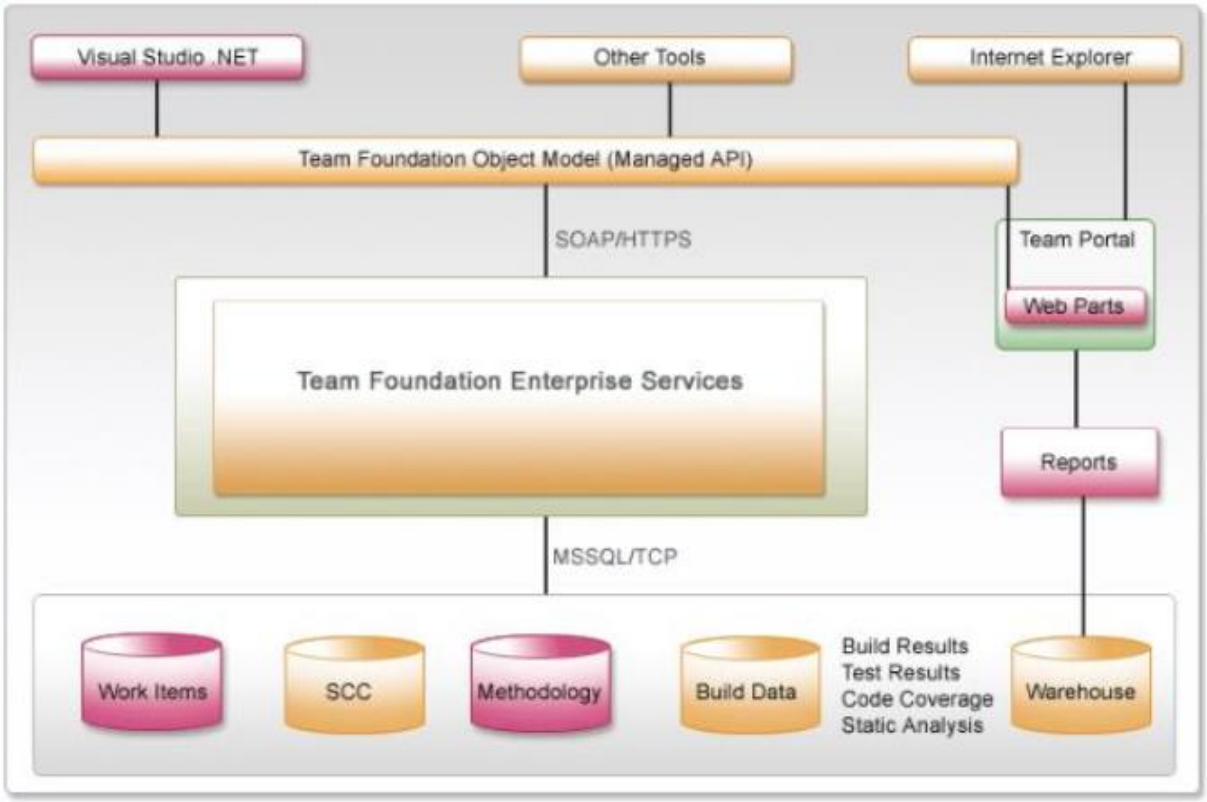
Nedostatak ove aplikacije je nemogućnost dekompozicije funkcionalnosti na projektne zadatke (Rychly i Ticha, 2007.). Prilikom isprobavanja alata, program se rušio kod spremanja projekta, odnosno nije moguće spremiti započeti projekt.

4.2. Ostali pokušaji razvoja alata

Kako su svi drugi alati osim FDD Tools Projecta ugašeni, nije ih moguće isprobati, stoga je napravljen kratak pregled prema pronađenim izvorima.

FDD Tracker bila je stolna aplikacija namijenjena operacijskim sustavima MS Windows, omogućavala je više uloga i dijeljenu bazu podataka te dekompoziciju funkcionalnosti na radne pakete (Rychly i Ticha, 2007.). Aplikaciju više nije moguće preuzeti, budući da je domena www.fddtracker.com/ napuštena.

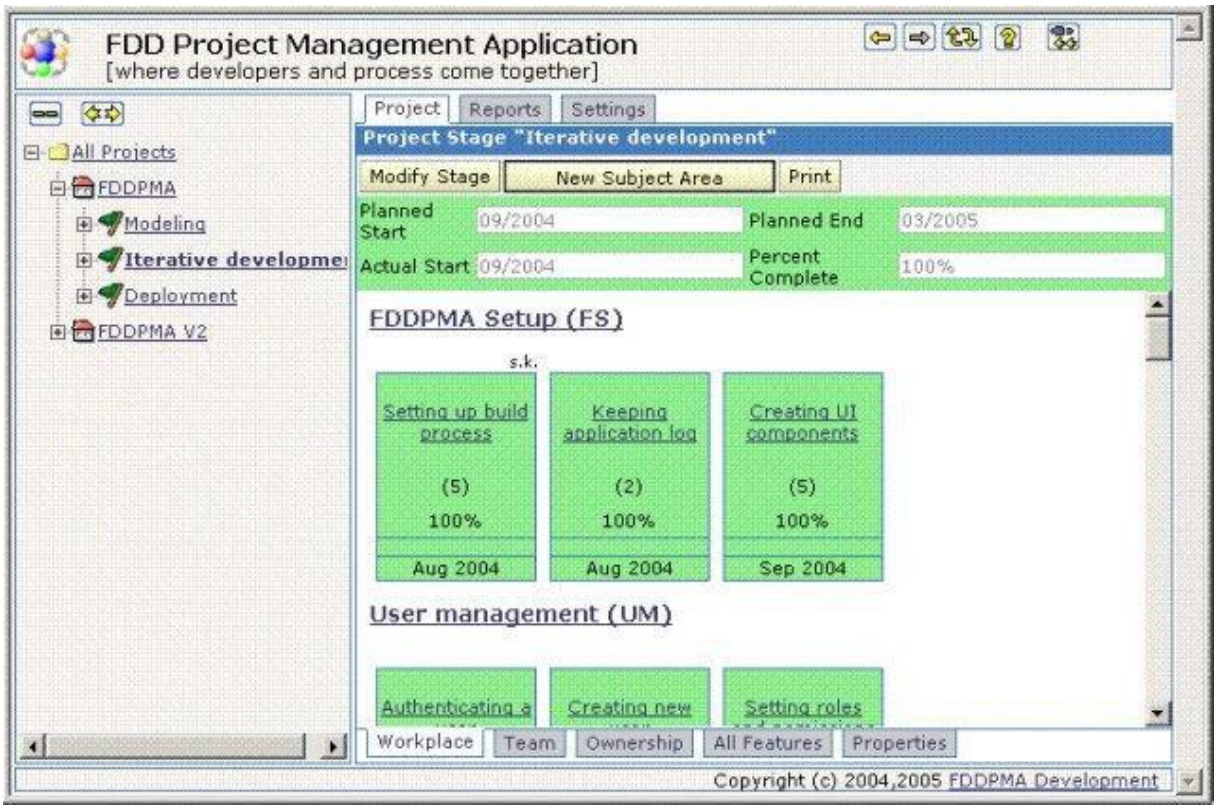
Cognizantova verzija FDD-a spomenuta je u poglavlju 3.6. Cognizant Feature Driven Development bio je alat za Microsoft Visual Studio, napravljen kao ekstenzija za MVS Team Foundation Server, današnji Azure DevOps Server (Microsoft, 2021.). Sljedeća slika prikazuje način implementacije Cognizant FDD-a u Team Foundation Server.



Slika 6. Arhitektura Cognizant FDD-a (Izvor: Shindhe, 2007.)

Mogući razlog neuspjeha alata je ovisnost o okruženju MS Visual Studio, odnosno nemogućnost korištenja u drugim razvojnim okolinama. (Rychly, Ticha, 2007.) Iako se alat još uvijek nalazi na Microsoftovim stranicama, poveznica za preuzimanje preusmjerava na nepostojeću stranicu.

FDD Project Management Application, ili skraćeno FDDPMA, bila je web aplikacija otvorenog koda bazirana na Javi. Aplikaciju je razvio Serguei Khramtchenko 2005. godine tokom rada na projektu za upravljanje cjeloživotnim ciklusom aplikacije (Khramtchenko, 2005.). Sljedeća slika prikazuje korisničko sučelje aplikacije FDDPMA:

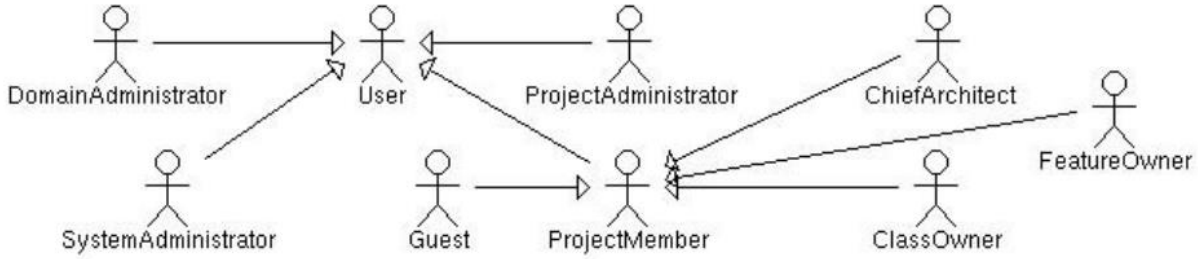


Slika 7. Korisničko sučelje FDDPMA (Izvor: SourceForge)

Budući da niti ova aplikacija više nije dostupna za preuzimanje, odnosno domena za registraciju je neaktivna, za prikaz sučelja odabrana je slika sa službene stranice za preuzimanje.

4.3. Plan vlastitog softvera

Na temelju proučavanja ostalih aplikacija, Ticha i Rychly razvili su plan za vlastiti softver za FDD. Odabirom najboljih karakteristika iz proučavanih aplikacija, ključni zahtjevi novog sustava su da bi softver trebao biti web-orijentiran te da omogućava grupiranje funkcionalnosti u skupove funkcionalnosti, ali i njihovo razlaganje na radne pakete, odnosno da omogućava i *bottom-up* i *top-down* pristup. Korištenje web sučelja omogućava jednostavan pristup podacima osobama koje rade na projektu, posebno stručnjacima za domenu i predstavnicima korisnika. Radni paketi pak predstavljaju zadatke potrebne za izvršavanje funkcionalnosti, povezivanje aktivnosti i relevantnih klasa ili njihovih dijelova koji moraju implementirati programeri kako bi dovršili aktivnost, te detaljno hijerarhijsko praćenje stanja i napretka funkcionalnosti od programera do menadžera. Sljedeća slika prikazuje korisničke uloge u osmišljenom sustavu Tiche i Rychlyja:



Slika 8. Dijagram uloga planiranog sustava (Izvor: Rychly i Ticha, 2007.)

Projekt je vođen od strane tri administratora, od kojih svaki ima zasebno područje kojim se bavi. Administrator sustava rješava tehnička pitanja u cjelokupnom sustavu, kao npr. kontrola nad pravima korinika i domenom, dok administrator domene održava dio sustava specifičnog za domenu, kao što su korisnici u domeni. Administrator projekta kreira i briše projekt te može obavljati promjene u njemu, dodaje i uklanja korisnike te im dodjeljuje uloge. U ovom sustavu gost je vanjski supervizor sustava, koji je tu u ulozi predstavnika kupca i posjeduje mogućnost pregledavanja izvješća o napretku projekta (Rychly i Ticha, 2007.).

Kao nedostatak njihovog sustava istakli su izoliranost od drugih razvojnih alata i nedostatak naprednih mogućnosti kao što je praćenje procesa verifikacije funkcionalnosti, praćenje i bilježenje nedostataka te nemogućnost kolaboracije timova.

Budući da Rychly i Ticha nisu objavljivali više radova, za pretpostaviti je da njihov osmišljeni softver nije realiziran. Kako je aktivan samo alat FDD Tools Project, nije bilo moguće napraviti usporedbu s drugim alatima, tek je iz slike 7 vidljivo da FDDPMA ima slično korisničko sučelje. FDD Tools Project alat je ograničenih funkcionalnosti, koji uz to ne radi ispravno, stoga se može zaključiti da feature-driven development nema odgovarajuću programsku potporu.

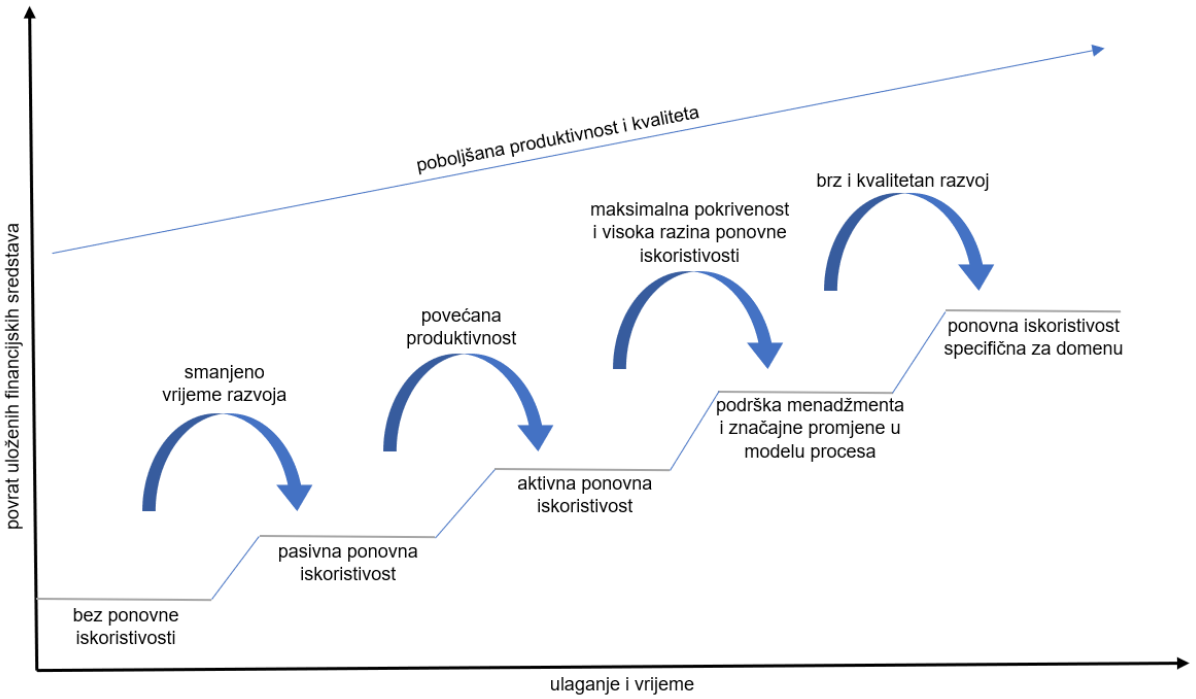
5. Nadogradnje na FDD

Kako je FDD brzo postao popularan i široko rasprostranjen, počele su se javljati različite modifikacije i nadogradnje na osnovni FDD proces, koje uključuju neke druge metode ili tehnike razvoja, te primjena FDD-a koja nije striktno vezana za razvoj softera. Većinu tih metoda začeli su studenti tokom projekata za potrebe fakulteta.

5.1. FDD s ponovnom iskoristivosti (FDRD)

S pojmom ponovne iskoristivosti koda su dobro upoznati front-end programeri, pogotovo React developeri, budući da se React temelji na izradi komponenata koje je moguće koristiti koliko god je puta potrebno. Riječ je o praksi gdje se prilikom izrade programa koriste prethodno napravljeni dijelovi koda, kako bi se uštedjelo vrijeme pisanja koda, ali i u projekt uključio prethodno provjeren i ispitan kod.

Ponovna iskoristivost softvera proces je stvaranja novog softvera od postojećih softverskih artefakata, odnosno softver se ne izrađuje od nule u cjelini. Ne uključuje nužno samo čisti kôd, već i gotove uzorke dizajna, testne slučajeve, razvojni okvir, pa čak i dokumentaciju i znanje. Učinkovita ponovna uporaba softverskih artefakata skraćuje vrijeme potrebno za razvoj i plasiranje na tržište te tako povećava produktivnost i smanjuje troškove razvoja. Ponovna iskoristivost, odnosno ponovna uporaba, dijeli se na aktivnu i pasivnu. Aktivnu uporabu planiraju softverski inženjeri, organizacija je može prilagoditi na najbolji mogući način. Pasivna uporaba nastaje kad programer koristi svoj stari kôd što nije u nadležnosti organizacije ili voditelja projekta. Takva pasivna iskoristivost smanjuje potrebno vrijeme razvoja, ali ne pridonosi ni rastu organizacije ni osobnom razvoju programera. Graf na sljedećoj slici prikazuje važnost ponovne iskoristivosti (Thakur i Singh, 2014.).



Slika 9. Važnost ponovne iskoristivosti (prema Thakur i Singh, 2014.)

Indijski studenti Sonia Thakur i Harshpreet Singh odlučili su modificirati FDD na način da u proces izrade softvera ubace ponovnu iskoristivost i tako je nastao razvoj temeljen na funkcionalnostima uz ponovnu iskoristivost, odnosno Feature-Driven Reuse Development (FDRD).

Način na koji je u FDD implementirana ponovna iskoristivost počinje još u procesu analize domene. Inženjer domene u procesu identificiranja modela uočava da nije riječ o potpuno novom sustavu, već o varijanti drugih sustava. Integracija ponovne iskoristivosti odvija se kroz sastavljanje funkcionalnosti po *bottom-up* pristupu, koje uključuje kodiranje, testiranje i implementaciju, gdje dolaze do izražaja funkcionalnosti za višekratnu iskoristivost. Potom se u fazi dizajna po funkcionalnosti prolazi kroz domenu, dizajnira i pregledava dizajn, čime nastaje funkcionalnost za ponovnu upotrebu s potpuno informativnom dokumentacijom. Tako nastaje skup funkcionalnosti spreman za ponovnu upotrebu u bilo kojem projektu. Kako bi se komponenta mogla ponovno koristiti u budućnosti, važna je uloga tehničkih pisaca, koji prave odgovarajuću dokumentaciju, koja sadrži arhitekturu, uml. modele, rječnik podataka, opisni dokument itd.

Tri ključne uloge u primjeni ponovne iskoristivosti u softverskom inženjerstvu su kreator (*creator*), pobornici (*supporters*) i korisnici (*reusers*). Korisnici pristigle zahtjeve za projekte šalju kreatoru, koji pronalazi postojeće komponente za koje smatra da se mogu iskoristiti te ih

prosljeđuje poborniku. Pobornik potvrđuje te komponente i izvješće šalje korisniku, koji ih ujedinjuje s novoizgrađenim komponentama te izbacuje verziju projekta.

Thakur i Singh svoje su istraživanje proveli u tvrtci Microchip Pvt Ltd., organizaciji srednje veličine koja nudi web usluge. Problemi koji su prisutni u tvrtci su odgoda isporuke, djelomično testiranje, loše navike kodiranja i sukladno tome kasniji problem u održavanju, dok je fleksibilnost najbolji dio radnog okruženja tvrtke. Prva primjena FDRD-a nije bila posebno uspješna u pogledu kvalitete koda i vremena izrade projekta, budući da je organizaciji to bio potpuno novi pojam te je trebalo vremena da se zaposleni priviknu. Druga primjena već je pokazala povećanu produktivnost, ali i kvalitetu proizvoda. Osim ponovne iskoristivosti koda, ovakav model pozitivno utječe i na znanje i iskustvo zaposlenika. Različitim članovima tima dodjeljuje se nova odgovornost, čime se povećava njihovo znanje i sposobnost za učinkovit rad. Kako osoba nije ograničena na isključivo svoju ulogu, stječe iskustva iz ostalih uloga te se tako poboljšava predvidljivost i sposobnost upravljanja rizicima u drugim projektima (Thakur i Singh, 2014.).

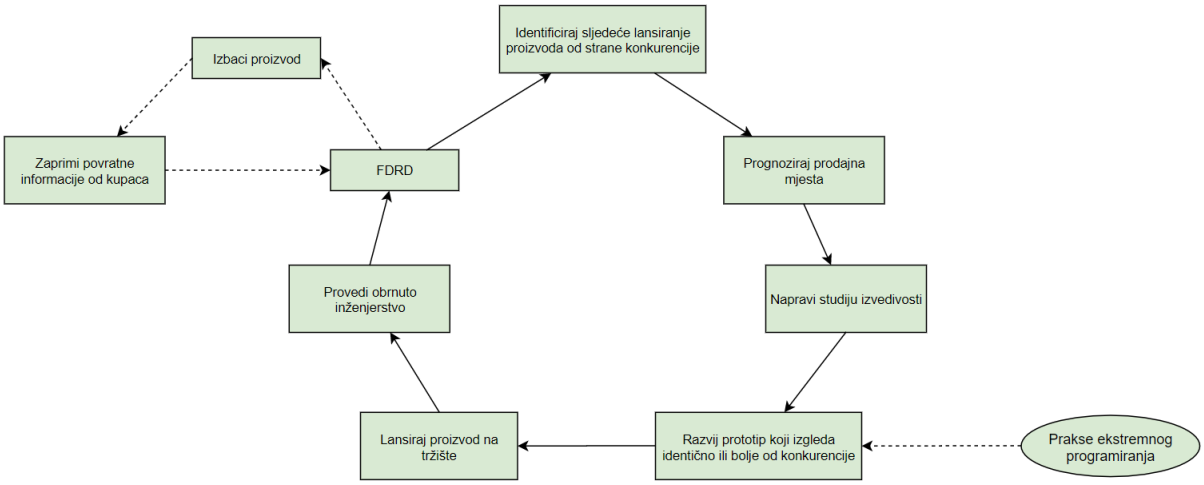
5.2. Razvoj softvera orijentiran na konkurenciju (CDD)

Istraživanje tržišta pokazalo je da kupac radije kupuje proizvod koji ispunjava njegove potrebe po najnižoj cijeni, nego dodatno plaća proizvod koji ima neke dodatne nepotrebne funkcionalnosti. Stoga se u informatičkoj industriji programeri ne bi trebali usredotočiti samo na vlastiti proizvod, već i na ono što nudi konkurencija.

Ekstremno programiranje još jedna je agilna metoda razvoja softvera. Suprotno FDD-u, velika pažnja posvećuje se testiranju. Zahtjevi se izražavaju kao scenariji, odnosno korisničke priče, koje se provode izravno kao niz zadataka. Za svaki zadatak, programeri u paru razvijaju testove prije pisanja koda, a prilikom integracije koda, svi testovi moraju biti uspješno izvedeni. Kao i kod FDD-a, često se izbacuju nove verzije programa (Sommerville, 2011.).

Student Vrajesh Pankaj Doshi i asistentica Vandana Patil sa Sveučilišta u Mumbaiju razvili su metodu za razvoj softvera koja u obzir uzima konkurenciju, odnosno tržišno natjecanje. Razvoj softvera orijentiran na konkurenciju (*Competitive Driven Development*, skraćeno CDD) kombinacija je feature-driven developmenta i ekstremnog programiranja (XP), namijenjena zadovoljenju tržišnih zahtjeva tamo gdje konkurentnost ima jednaku važnost kao i inovacije. Ovaj model namijenjen je organizacijama koje razvijaju IT proizvode za masovno ciljane kupce, a ne pojedincima ili organizacijama koje rade za individualnog klijenta ili nekoliko njih.

Sljedeći grafikon prikazuje korake koji sačinjavaju ovakav kompetitivni tip razvoja softvera i provode se kroz kružni ciklus. Grafikon prikazuje gdje do izražaja dolaze FDRD i ekstremno programiranje, a svi koraci objašnjeni su u nastavku.



Slika 10. Ciklus razvoja softvera orijentiranog na konkurenciju (prema Doshi i Patil, 2016.)

Koraci:

1. **Identificiraj sljedeće lansiranje proizvoda od strane konkurencije.**
   Tim za istraživanje i razvoj prikuplja uvjete za razvoj proizvoda promatrajući proizvod koji konkurent planira lansirati na tržište, a koji se pojavljuje vani putem oglasa. Proizvod se još nije počeo prodavati, moguće je skoro beta izdanje.
2. **Prognoziraj prodajna mjesta.**
   Tim za istraživanje i razvoj pokušava identificirati prodajnu tehniku konkurenta, tj. strategiju koju koristi kako bi kupce privukao da kupuju proizvod. Iz ovoga se može predvidjeti hoće li proizvod biti uspješan ili ne.
3. **Napravi studiju izvedivosti.**
   Ako se procijeni da je proizvod zanimljiv i profitabilan, provode se različite vrste studija izvedivosti za razvoj sličnog proizvoda. Ovisno o rezultatima, organizacija donosi odluku hoće li nastaviti razvoj.
4. **Razvij prototip koji izgleda identično ili bolje od konkurencije.**
   Ako je rezultat studije izvedivosti pozitivan, donosi se odluka o razvoju proizvoda i raspoređuju svi potrebni resursi za njegov razvoj. Na temelju izgleda, tj. korisničkog sučelja prikazanog u oglasima ili video promocijama, organizacija razvija identičan prototip proizvoda uz smanjen broj funkcionalnosti, ali uz određenu jedinstvenost i neke dodatne funkcionalnosti koje se razlikuju od konkurenata ako je moguće. Ovdje do izražaja dolazi ekstremno programiranje: budući da zahtjev koji tim razumije može biti nedovoljno jasan, ili se pak zahtjevi promijene, ili se nakon početka razvoja pojave novi zahtjevi za koje ne

postoji dovoljno razrađen algoritam ili postupak za provedbu nisu dovoljno jasni. Praksa ekstremnog programiranja „prvo test“ implicitno generira dizajn, odnosno promjenom zahtjeva ne troše se dodatni napori za izmjenu dizajna.

5. **Lansiraj proizvod na tržište.**
   Proizvod sličan konkurentskom treba izbaciti na tržište po vrlo niskoj cijeni, po mogućnosti otprilike u isto vrijeme kada se konkurentski proizvod počne prodavati na tržištu.
6. **Provedi obrnuto inženjerstvo.**
   Odmah po početku prodaje, potrebno je kupiti konkurentski proizvod, detaljno analizirati sve njegove karakteristike i funkcionalnosti te na taj način korak po korak provesti obrnuti inženjering konkurentskog proizvoda.
7. **FDRD.**
   Nakon provedenog obrnutog inženjeringa konkurentskog proizvoda, potrebno je navesti sve bolje i atraktivnije funkcionalnosti koje sadrži, kako bi se ugradile u vlastiti proizvod. Uz dovoljno jasne zahtjeve u pogledu funkcionalnosti, implementiraju se koristeći FDRD model procesa. Koncept ponovne uporabe ovdje omogućava postizanje visoke produktivnosti u manje vremena, što predstavlja ispunjavanje osnovnog zahtjeva tržišnog natjecanja.
8. **Izbaci proizvod.**
   Nakon što se su bolje i atraktivnije funkcionalnosti iz konkurentskog proizvoda ugrađene u vlastiti, proizvod se lansira na tržište kao novo izdanje, po konkurentnoj cijeni.
9. **Zaprimi povratne informacije od kupaca.**
   Provodi se istraživanje tržišta koje uključuje povratne informacije kupaca o vlastitim i konkurentskim proizvodima. Povratne informacije uključuju se u upravljanje zahtjevima, na temelju njih nastaju novi zahtjevi dizajna koji uključuju poboljšanje karakteristika vlastitog proizvoda koje su kupci ocijenili nezadovoljavajućima ili inferiornima u odnosu na konkurenciju te uključivanje funkcionalnosti i karakteristika iz konkurentskih proizvoda koje su kupci ocijenili kao bitne i ključne pri odabiru proizvoda. Promjene u zahtjevima šalju se u odgovarajuću fazu FDRD-a i tvore ciklus.

Kružni model sa slike 10 omogućava početak i završetak u bilo kojoj fazi, iako postoje određene preporuke. Ako je konkurencija izbacila potpuno novi proizvod, koji nije prethodno postojao na tržištu, preporuka je započeti s identifikacijom sljedećeg lansiranja konkurencije. S druge strane, ako je riječ o razvijenom proizvodu, koji predstavlja poboljšanje već postojećih proizvoda na tržištu, optimalno je započeti s obrnutim inženjerstvom.

Za ovakav pristup optimalna je kombinacija FDRD-a i ekstremnog programiranja. Da je korišten samo FDRD, koji zahtijeva detaljno projektiranje, mnogo vremena potrošile bi

promjene u dizajnu uzrokovane čestim promjenama zahtjeva, dok bi isključivo ekstremno programiranje stvorilo poteškoće kod obrnutog inženjeringa, gdje upravo detaljno projektiranje omogućava bolje razumijevanje funkcionalnosti za implementaciju (Doshi i Patil, 2016.).

5.3. Razvoj metodologije prema funkcionalnostima (FDMD)

Inženjering situacijskih metoda (SME) grana je softverskog inženjerstva koja se bavi razvojem metodologija prilagođenih specifičnim karakteristikama softverskog projekta. Razvoj metodologije prema funkcionalnostima, odnosno Feature-Driven Methodology Development (FDMD), namijenjen je malim i srednjim poduzećima. Korištenje funkcionalnosti u malim i srednjim poduzećima ima sljedeće potencijalne koristi: održivost, ponovna upotreba i podrška za izradu alata, budući da bi se objektno orijentirani modeli metodologije mogli ponovno koristiti za implementaciju alata.

FDMD obuhvaća puni životni ciklus metodologije koja uključuje objektnu orijentiranost u svim svojim aktivnostima. Sastoji se od tri faze: pokretanje, izrada metodologije i prekid. U fazi pokretanja sastavlja se popis funkcionalnosti te se rješavaju tehnički detalji projekta, potom se u drugoj fazi konstruira metodologija, a treća faza predstavlja aktivnosti prije isporuke metodologije klijentima, kao i samu isporuku. Svaka od tih faza sastoji se od ugniježđenih etapa, koje se pak sastoje od sitno zrnatih koraka.

5.3.1. Faza I – pokretanje (inicijacija)

U ovoj, početnoj fazi, određuju se funkcionalnosti te razvojni okvir, odnosno arhitektura metodologije. Ovu fazu čine sljedeće etape:

1. **Odaberi odgovarajući razvojni okvir za ciljnu metodologiju.**
   Okvir pruža opći životni ciklus metodologije te će se koristiti za identifikaciju klasa. Moguće je ponovno koristiti okvire generičkih procesa. U donošenje odluke uključeni su eksperti za organizacijsku domenu, softverski inženjeri, koji predstavljaju korisnike krajnje metodologije, inženjeri metoda, koji provode FDMD metodu, te projektni menadžer.
2. **Odredi klase.**
   Ovu etapu čini nekoliko koraka kroz koje se identificiraju klase:
   - Formiraj timove za ekstrakciju klasa. – Projektni menadžer uspostavlja timove inženjera metoda, softverskih inženjera i stručnjaka za domenu.
   - Ekstrahiraj klase. – Timovi paralelno izrađuju dijagrame klasa za ciljnu metodologiju. Ključne klase su radna jedinica, proizvod i proizvođač.
   - Odredi konačan skup klasa. – Konačan dijagram klasa dobiven je integracijom dijagrama klasa od svakog tima.

3. **Opiši zahtjeve po funkcionalnostima.**

   U ovoj etapi spoznaju se funkcionalnosti ključne metodologije.

   - Ekstrahiraj situacijske čimbenike. – Projektni menadžer ekstrahira karakteristike projekta kao situacijske faktore, koji dobivaju vrijednosti na temelju projektne situacije. Također se definiraju nefunkcionalni zahtjevi, koji mogu biti izvedeni iz situacijskih faktora ili ih specificira klijent.
   - Kategoriziraj funkcionalnosti.
   - Formiraj timove za funkcionalnosti. – Formiraju se timovi kao u drugoj etapi, te se timovima dodjeljuju funkcionalnosti.
   - Navedi funkcionalnosti na temelju unaprijed definiranih skupova uzoraka.
   - Integriraj sve razvijene funkcionalnosti u jedinstven dokument.
   - Provjeri dosljednost među funkcionalnostima. – Provjeravaju se nedosljednost i konflikti te se rješavaju potencijalni uočeni problemi.

Po dovršenju arhitekture, prelazi se na konstruiranje metodologije.

5.3.2. Faza II – konstrukcija metodologije

Kroz etape u fazi konstrukcije dolazi se do metodologije koja predstavlja cilj razvojnog projekta:

1. **Osmisli „motor“ konstrukcije.**

   Projektni menadžer uspostavlja razvojne timove inženjera metoda, softverskih inženjera i stručnjaka za domenu. Skupovi funkcionalnosti već su proiritizirani i dodjeljeni razvojnim timovima.

2. **Motor konstrukcije.**

   Ova etapa provodi se kao iterativni ciklus dok se ne dovrše sve funkcionalnosti. Svaki razvojni tim dovršava svoje funkcionalnosti.

   - Odredi prioritet i odaberi grupu funkcionalnosti visokog prioriteta.
   - Modeliraj ponašanje. – Svaka funkcionalnost iz odabrane skupine dobija svoj dijagram slijeda koji prikazuje interakcije objekata potrebne za njezinu realizaciju.
   - Ažuriraj dijagram klasa dodavanjem objekata za koje je utvrđena uloga u prethodnom koraku.
   - Provodi razvojni proces. – Razvojem odabrane skupine funkcionalnosti prvi put, kreira se model metodologije, dok se svakom sljedećom iteracijom ažurira.
   - Navedi ponovno iskoristive komponente, što mogu biti funkcionalnosti, dijagrami i klase pohranjene u skladištu podataka.
   - Upravljaj promjenom zahtjeva, tj. ažuriraj popis značajki i bilježi promjene.
   - Održi recenziju na sastanku.

- Opcionalno implementiraj alatnu podršku. – Nakon što se odabere razvojno okruženje, razvijaju se alati na temelju dijagrama klasa i dijagrama slijeda proširenim objektima dizajna.

3. **Završi konstrukciju.**
   Projektni menadžer objedinjuje dijagrame procesa koje su razvili timovi u jedan konačan dijagram, na kojem je potrebno obratiti pozornost na (ne)dosljednost.

Kako je metodologija konstruirana, započinju pripreme za njezinu isporuku krajnjim korisnicima. Prije isporuke potrebno je testirati ispravnost metodologije, ali i održavati je nakon isporuke.

5.3.3. Faza III – završetak (terminacija)

U ovoj fazi prozivod se testira, isporučuje i održava. Kao i prethodne dvije faze, sastoji se od tri etape:

1. **Testiraj.**
   Prije isporuke, stručnjaci metodologiju provjeravaju i validiraju prema zahtjevima.
2. **Provedi popratne aktivnosti.**
   Naučeno bitno iskustvo pohranuje se u dokument, a razvojni proces ispituje se hoće li biti pogodan za buduće projekte.
3. **Isporuči i održavaj.**
   Dokument metodologije sadrži detaljne informacije o proizvedenoj metodologiji, a korisnici, tj. softverski inženjeri kojima je namijenjena, obučeni su o njezinom pravilnom provođenju te započinju s primjenom. Problemi koji se pojave identificiraju se i rješavaju.

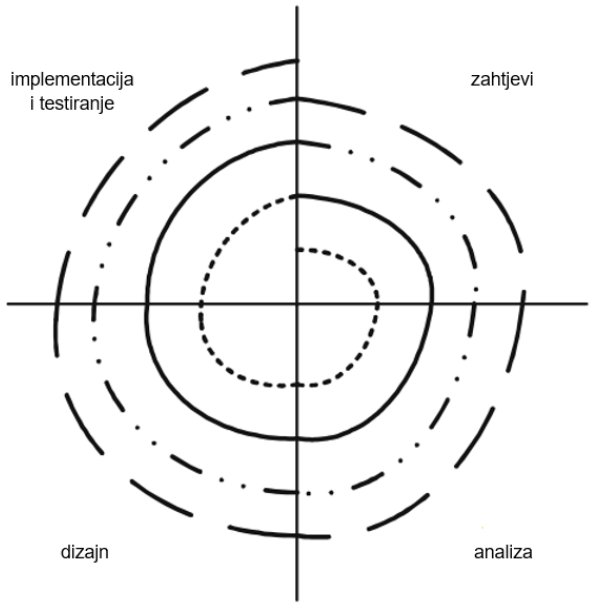
Ovakav način razvoja metodologije namijenjen je malim i srednjim poduzećima, koje karakterizira rigorozno inženjerstvo zahtjeva, a u procesu razvoja zahtjevi ciljne metodologije obuhvaćeni su kao funkcionalnosti (Mahdavi-Hezave i Ramsin, 2015.).

6. Zašto (ne) koristiti FDD?

Ovo pitanje predstavlja bit cjelokupnog rada, no konkretan odgovor na njega nije lako dati. Kao i svaka druga metoda, tako i razvoj softvera prema funkcionalnostima ima svojih prednosti i nedostataka. Određene karakteristike će u nekim situacijama doći do izražaja kao prednost, dok će se u nekim drugima ogledati kao nedostatak.

FDD metodologija relativno je jednostavna za naučiti. Sastoji se od pet koraka koji su lako prilagodljivi i odgovaraju većini projekata, a unatoč jednostavnosti, postupak prilično temeljito pokriva životni ciklus razvoja softvera. Također, ono što može korisnicima biti prekretnica za odabir FDD-a je činjenica da nije riječ o pukoj teorijskoj metodologiji, najprije napisanoj u knjizi, već dolazi iz stvarnog životnog iskustva i kroz godine je u praksi dokazana mnogo puta i tijekom mnogih projekata (Cause, 2004.).

Sama činjenica da se FDD ubraja u agilne metode govori da je riječ o metodi koja u obzir uzima promjenjive zahtjeve korisnika, omogućuje često objavljivanje verzija programa, a s time i često primanje povratnih informacija od korisnika. S druge strane, agilni i inkrementalni projekti mogu biti teški za planiranje, upravljanje i praćenje. U tradicionalnom, linearnom modelu razvoja softvera, projektiranje ne započinje prije nego se završe sve analize, niti implementacija kreće prije dovršetka dizajna, tako da je u svakom trenutku jasno tko radi što, kako i zašto. Također su konačni zahtjevi ugovoreni odmah na početku projekta, što olakšava posao programerima. (Hunt, 2006.) Stoga FDD za mlade i neiskusne programere može predstavljati izazov. Hunt također upozorava da je zbog agilnosti moguće izgubiti kontrolu nad projektom.



Slika 11. Spiralni model razvoja softvera (Izvor: Hunt, 2006.)

FDD neće nužno samo neiskusnim programerima stvoriti poteškoće, moguće su i kod iskusnih projektnih menadžera i glavnih programera koji u određenoj aplikacijskoj domeni rade prvi put. Kako se FDD provodi po spiralnom modelu sa slike, pretpostavka projekta je da će velika većina jedne faze biti dovršena prije nego sljedeća faza dođe u razmatranje. To je izvedivo kod istovrsnih projekata kakvi su slični već rađeni u prošlosti, dok je kod projekata potpuno

novih u pogledu domene vjerojatnost za to mala. Zbog takvog spiralnog pristupa, vrijeme potrebno za izvođenje iteracije može postati jasno tek na početku te iteracije, što se može negativno odraziti na planiranje proračuna, planiranje termina izbacivanja verzije ili općenito na upravljanje projektom.

Jedna od najvećih prednosti FDD-a ogleda se već u prvom koraku, odnosno modeliranju domene: fleksibilnost. FDD nije primjenjiv samo na projekte koji kreću od nule; prethodno obavljeni posao može se iskoristiti kao polazna točka za poboljšanje i dodavanje detalja. Fleksibilnost se, osim u polasku projekta, iskazuje i prilikom definiranja tehničke arhitekture. Moguće je detaljno definirati proces za svaki scenarij i pokriti sve moguće varijacije, što stvara veliku količinu procesne dokumentacije za razvojne programere, ili zadatak ostaviti uglavnom nedefiniranim da razvojni tim može prilagoditi proces vlastitoj specifičnoj situaciji. Budući da moguće varijacije onemogućuju davanje sažetog, jednoznačnog odgovora, uobičajeno je koristiti drugi pristup.

Osim toga, u procesu modeliranja domene otkrivaju se i nejasnoće i neodređenosti zahtjeva. Budući da zahtjevi kod razvoja softvera često mogu biti nejasni i neodređeni, njihovo rano otkrivanje sprječava da se vuku sve do slučajeva korištenja, funkcionalnih specifikacija ili čak korištenja od strane klijenata. Ukoliko se nejasnoće ne razriješe tokom modeliranja domene, većina ih se otkrije i riješi u koracima „dizajniraj po funkcionalnostima“ i „gradi po funkcionalnostima“ (Palmer i Felsing, 2002.).

FDD može pomoći u rješavanju komunikacijskih barijera na poslovnim sastancima. Mnoge konzultantske tvrtke često razgovaraju o tehnologiji s poslovnim ljudima kojima je to izvan područja sturčnosti, što dovodi do poteškoća u komunikaciji, koje pak mogu uzrokovati postizanje za obje strane nezadovoljavajućih rezultata. FDD korištenjem ugrađenih predložaka imenovanja izbjegava prekid komunikacije iz ovog razloga te omogućuje objema stranama korištenje zajedničkog jezika (Cause, 2004.).

Snaga FDD-a je i u tome što se čestim recenzijama i povratnim informacijama kontrolira tijek projekta. Ako se projekt ne bude razvijao ispravnim tokom, te ako se poštuje pravilo o dvotjednim funkcionalnostima, nepravilnost će biti otkrivena relativno brzo (Palmer i Felsing, 2002.). Uz recenzije i povratne informacije, prisutne su i ranije spomenute inspekcije. Može se reći da FDD inspekcijama pokušava spriječiti probleme prije nego što se dogode. Jedan od načina na koji to čini je korištenje inspekcija. S vremenom su se inspekcije pokazale kao jedna od najučinkovitijih, ali još uvijek nedovoljno korištenih metoda za sprječavanje programskih pogrešaka.

Ovakav koncept instant izvještavanja rezultata naziva se visoka vidljivost, i još je jedan od razloga popularnosti FDD-a. Kako FDD pruža mehanizam izvještavanja rezultata kao dio procesa, nije nužno voditi brigu o tome na koji način će se prikupiti podaci o (ne)uspješnom tijeku projekta, što značajno smanjuje opterećenje članova tima (Cause, 2004.).

Jedna karakteristika FDD-a istovremeno može biti i prednost i nedostatak. Naime, u FDD-u je prisutno tzv. „pravilo 10%“, koje govori da projekt u kojem se primjenjuje FDD može podnijeti povećanje broja funkcionalnosti za do 10% bez utjecaja na razvoj projekta. Razlog tome je što je trajanje jedne funkcionalnosti relativno kratko, do dva tjedna. Za povećanje broja funkcionalnosti za više od 10%, projektni menadžer mora obavijestiti sponzore projekta i vrhovni menadžment te se takve promjene rješava na jedan od tri načina:

1. smanjiti opseg projekta uklanjanjem manje bitnih funkcionalnosti,
2. produljiti vremenski plan razvoja kako bi se našlo mjesta za nove funkcionalnosti,
3. povećati kapacitet tima dodavanjem novog glavnog programera te još nekoliko programera kako bi se istovremeno moglo razvijati više funkcionalnosti.

Stoga, uz pretpostavku da se naknadno neće dodavati mnogo novih funkcionalnosti, FDD će biti prikladan za projekte. Štoviše, bit će i koristan, budući da se eventualnim dodavanjem manjeg broja funkcionalnosti neće poremetiti vremenski plan projekta. S druge strane, kod značajnih promjena u projektu, odnosno naknadnih zahtjeva, tražit će velike promjene u planu.

Ono što također može predstavljati ključan faktor u izboru metode, ovisno o osobnim preferencijama programera, je testiranje. Testiranju u FDD nije posvećeno toliko pažnje kao npr. u ekstremnom programiranju. Jedan od razloga je što u većini slučajeva procesi korišteni za testiranje nisu glavni problemi organizacije, već je potrebno osigurati da se stvori sustav vrijedan testiranja prije nego što se previše vremena potroši na detalje procesa koje treba slijediti. Stvaranjem radnog koda visoke kvalitete, testiranje postaje formalnije (Palmer i Felsing, 2002.).

Još jedna značajna odlika FDD-a je prilagodljivost. Cijelo peto poglavlje posvećeno je kombinacijama FDD-a i drugih metoda razvoja softvera, gdje bi se stvorio kombinirani proces koji se pokazao uspješnim i profitabilnim, a da se nije izgubio smisao osnovnog FDD-a. Tako da se FDD ne mora primjenjivati isključivo samostalno, već i u okviru nekih drugih metoda, a s tim i predstavljati temelj za razvoj novih metoda.

Nažalost, četvrto poglavlje istaknulo je jednu značajnu manu FDD-a: nedostatak softverske podrške. Iako je kroz povijest pokušaja bilo, projekti su napušteni. Jedina aplikacija koja je o(p)stala, FDD Tools Project, nije potpuno dovršena da bi bila primjenjiva kod velikih projekata, s većim brojem uloga, timova i funkcionalnosti. Upravo to bi za menadžere i eksperte

domene moglo biti ključno za davanje prednosti nekim drugim metodama, za koje postoji prikladna softverska podrška.

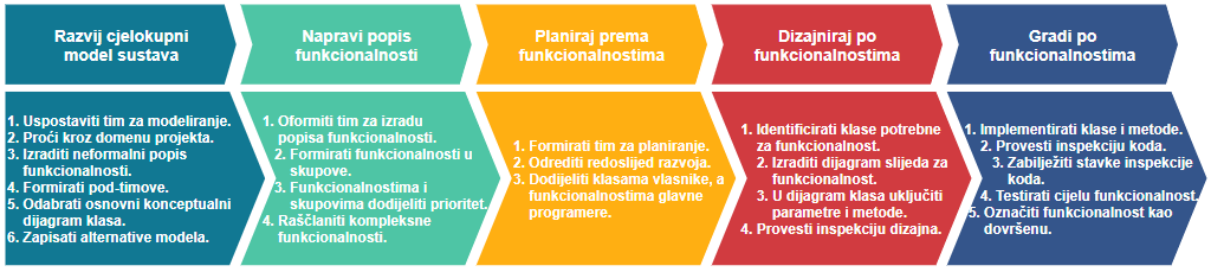
U usporedbi s adaptivnim razvojem softvera, FDD zahtijeva unaprijed definiranu tehniku za proces zahtjeva, odnosno u fazi implementacije zahtjeva potrebno je pridržavati se standarda za kodiranje. S druge strane, adaptivni razvoj softvera više se fokusira na rezultate nego na radne zadatke i ne zahtijeva tehniku ili pridržavanje određene prakse, osim za rješavanje problema. Zbog toga je FDD zatvoreniji za usvajanje novih tehnologija nego adaptivni razvoj (Chowdhury i Nazmul Huda, 2011.).

Na osnovu iznešenih pozitivnih i negativnih obilježja, može se reći da je FDD izuzetno atraktivna i fleksibilna metoda za provođenje, čemu doprinose spiralni model razvoja i pravilo 10%. Također rješava probleme komunikacije i kontrole napretka spomenute u uvodu, razlog odabira ove teme. Ipak, detaljan zaključak bit će donesen nakon provedbe programskog dijela.

7. Praktični dio

U ovom dijelu rada prikazana je provedba razvoja softvera prema funkcionalnostima u praksi. Aplikacija koja se gradi nazvana je TrackYourCrew, i riječ je o desktop aplikaciji izrađenoj u programskom jeziku C# koja služi za evidenciju zaposlenih studenata. Od naručitelja su zaprimljeni zahtjevi za unošenjem studenata i radnih mjesta u sustav te da svaki student može imati samo jedan ugovor u mjesecu, pri čemu može odraditi najviše 160 sati mjesečno, zbog pravila student servisa. Isplaćene ugovore moguće je ispisati kao izvješće, te je opcija grafički prikazati broj zaposlenih studenata po mjesecima.

Razvoj aplikacije provodi se po standardnoj FDD metodologiji kakva je opisana u poglavlju 3.5. i prikazana na sljedećoj slici:



Slika 12. FDD metodologija (autorska izrada)

Budući da je riječ o novom projektu, odnosno ne postoje prethodni projekti slične strukture koji bi mogli sadržavati komponente iskoristive za ovaj softver, nije izvedivo primijeniti ponovnu iskoristivost, odnosno provesti FDRD metodologiju. Provedba postupka opisana je na način da su prva tri koraka prikazana kao zasebna poglavlja, dok su ponavljajući koraci 4 i 5 sadržani u poglavljima koja predstavljaju skupove funkcionalnosti.

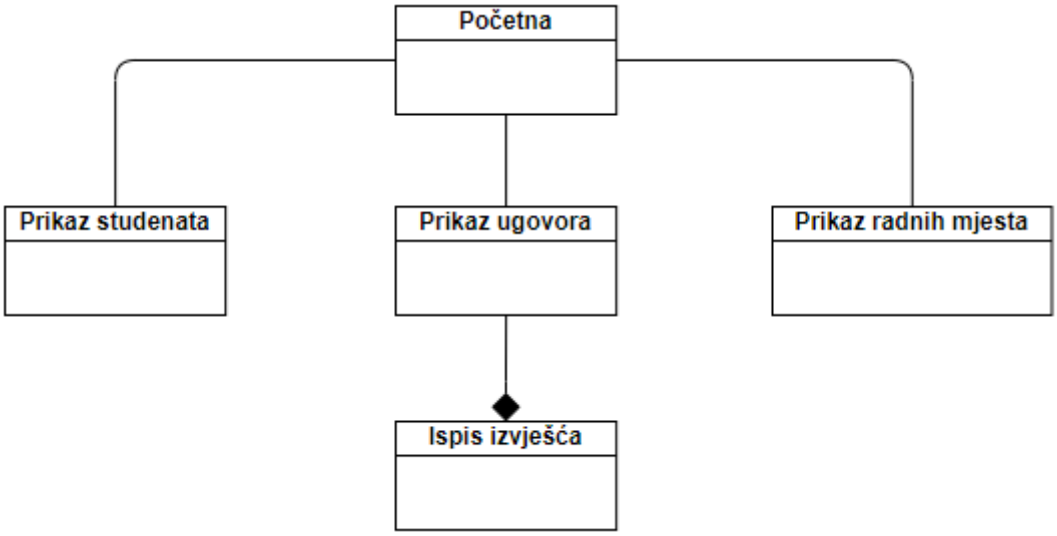
7.1. Razvij konceptualni model sustava

Prvi korak u postupku razvoja započinje formiranjem tima za modeliranje. Novo uspostavljeni tim izrađuje neformalni popis funkcionalnosti:

- pokrenuti aplikaciju s navigacijom među formama,
- omogućiti operacije upravljanja bazom podataka nad studentima,
- omogućiti operacije upravljanja bazom podataka nad radnim mjestima,
- napraviti podnošenje ugovora,
- dodati grafički alat.

Tim za modeliranje podijeljen je u dva pod-tima. Prvi pod-tim bavi se formama aplikacije, što uključuje prvu i četvrtu funkcionalnost s popisa. Zadatak drugog pod-tima je konstruirati bazu podataka, što označava povezivanje studenata, radnih mjesta i ugovora.

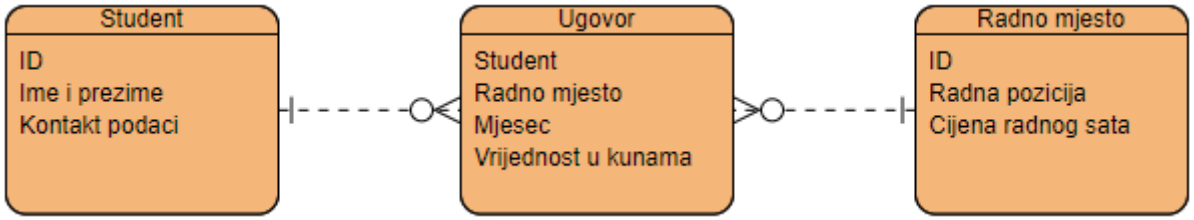
Kako je aplikacija izrađena u predlošku Windows Forms, forme su konstruirane kao klase, stoga je za njih potrebno napraviti dijagram klasa. Na slici 13 prikazan je konceptualni dijagram klasa za forme napravljen od strane prvog pod-tima, koji predstavlja planirani raspored otvaranja formi kroz aplikaciju:



Slika 13. Konceptualni dijagram klasa za forme u aplikaciji (autorska izrada)

Početna forma je s ostalim formama povezana asocijacijom, budući da služi samo za pokretanje ostalih formi, bez međudjelovanja objekata. Izvješće je s ugovorom povezano kompozicijom jer je riječ o klasi koje se pokreće na temelju odabrane instance ugovora, odnosno uvjetovana je instancom. Na formi Prikaz ugovora prikazuju se isključivo aktivni, tj. trenutno otvoreni ugovori, ne i zatvoreni, koji su već isplaćeni.

Drugi pod-tim, koji se bavi osmišljavanjem baze podataka, umjesto dijagrama klasa napravio je ERA model, u ovoj fazi također konceptualni:



Slika 14. Konceptualni ERA model (autorska izrada)

Klase u ERA modelu predstavljaju tablice u bazi podataka, gdje je su atributi Student i Radno mjesto u tablici Ugovor vanjski ključevi prema istoimenim tablicama, dok bi primarni ključ tablice Ugovor bio kombinacija atributa Student i Mjesec.

Alternativa modela osmišljena je tako da se popisu ugovora ne pristupa direktno s početne forme, nego da na formi Prikaz studenata budu dvije tablice u kojima bi bili sadržani aktivni i isplaćeni ugovori.

Vrhovni arhitekt odobrio je konceptualni dijagram slijeda i ERA model, kao i alternativni model, tako da se može nastaviti s izradom popisa funkcionalnosti.

7.2. Napravi popis funkcionalnosti

U ovom koraku napravljen je detaljan popis funkcionalnosti koje su grupirane u sljedeće skupove:

1. Pokretanje aplikacije:
   - Izraditi bazu podataka. (A)
   - Spojiti aplikaciju na bazu podataka. (A)
   - Napraviti navigaciju među formama. (A)
2. Dodavanje studenata i radnih mjesta u sustav.
   - Unijeti novog ili izmijeniti podatke o postojećem studentu. (A)
   - Unijeti novo ili izmijeniti podatke o postojećem radnom mjestu. (A)
3. Potpisivanje ugovora.
   - Učitati popis svih studenata i ugovora iz baze podataka. (A)
   - Staviti ograničenje o jednom ugovoru mjesečno po studentu. (A)
   - Zatvoriti ugovor. (A)
   - Ispisati izvješće. (C)
4. Statistički prikaz studentskog rada.
   - Prikazati broj radnih sati mjesečno kroz godinu. (D)

Kako se sve funkcionalnosti mogu izvršiti kroz manje od dva tjedna, nije potrebno vršiti njihovu dekompoziciju. Slovne oznake A-D objašnjene su u poglavlju 3.5.2. Većina funkcionalnosti označena je slovom A i riječ je o funkcionalnostima koje predstavljaju osnovicu sustava, dok će se ispisivanje izvješća (oznaka C) implementirati ako bude dovoljno vremena. Funkcionalnost statističkog prikaza osmišljena je kao grafička, te je ostavljena za buduće ažuriranje programa (oznaka D).

7.3. Planiraj prema funkcionalnostima

Upravo uspostavljeni tim za planiranje izrađuje vremenski plan projekta kroz rok od devet radnih dana, pri čemu se radi na pola radnog vremena, odnosno četiri sata dnevno, što maksimalno trajanje projekta ograničava na 36 sati. Izrada aplikacije provedena je u veljači 2021. godine s početkom na ponedjeljak 7.2.2021. i zadanim završetkom na četvrtak 17.2.2021. prema planu projekta prikazanom u sljedećoj tablici:

Tablica 2. Vremenski plan projekta

Funkcionalnost	Datum početka	Datum završetka	Trajanje u danima
Izraditi bazu podataka.	7.2.2021.	8.2.2021.	2
Spojiti aplikaciju na bazu podataka.	9.2.2021.	9.2.2021.	0,5
Napraviti navigaciju među formama.	9.2.2021.	10.2.2021.	1,5
Unijeti novog ili izmijeniti podatke o postojećem studentu.	11.2.2021.	11.2.2021.	0,5
Unijeti novo ili izmijeniti podatke o postojećem radnom mjestu.	11.2.2021.	11.2.2021.	0,5
Učitati popis svih studenata i ugovora iz baze podataka.	14.2.2021.	14.2.2021.	1
Staviti ograničenje o jednom ugovoru mjesečno po studentu.	15.2.2021.	15.2.2021.	1
Izračunati iznos prilikom sklapanja ugovora.	16.2.2021.	16.2.2021.	0,5
Ispisati izvješće.	16.2.2021.	16.2.2021.	0,5
Ukupno	**7.2.2021.**	**16.2.2021.**	**8**

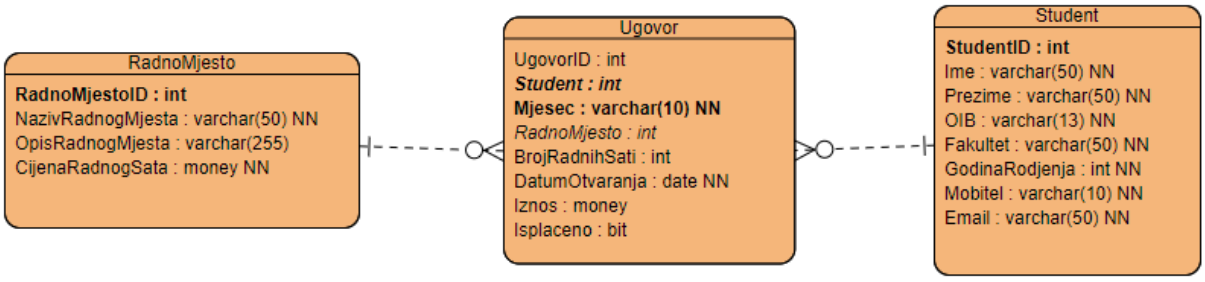
(autorska izrada)

Iako je funkcionalnost „ispisati izvješće“ označena slovom C, odnosno napraviti ako ostane vremena, plan projekta pokazao je da se funkcionalnost može implementirati unutar zadanog roka, stoga je uvrštena u raspored. Spomenuta funkcionalnost vezana za grafički prikaz i ostavljena za buduće ažuriranje projekta nije uvrštena u plan projekta, budući da se planira isključivo za prvu sljedeću verziju za izbacivanje. Prema planu iz tablice, izračunato je da bi projekt trebao biti gotov dan prije isteka roka.

7.4. Skup funkcionalnosti „pokretanje aplikacije“

Ovaj skup uključuje izradu baze podataka, spajanje aplikacije na bazu potadaka te kreiranje početne forme i formi koje se s nje otvaraju, odnosno uređivanje navigacije kroz aplikaciju.

Baza podataka predstavlja osnovu projekta. Za njezinu izradu odabrana je inačica SQL-a MS SQL te pripadajući alat Microsoft SQL Server Management Studio. Model tablica u bazi identičan je konceptualnom ERA modelu sa slike 14, dok je atribute u tablicama potrebno detaljizirati, odnosno popisati sve potrebne atribute dodijeliti im tip podataka. Detaljni ERA model prikazan je na sljedećoj slici:



Slika 15. ERA model baze podataka (autorska izrada)

Iako je u tablici Ugovor primarni ključ skup atributa Student i Mjesec, dodan je atribut UgovorID, koji se dodaje automatski kao serijski broj, radi kasnijeg lakšeg pronalaska ugovora u bazi podataka i pojednostavljenja pisanja upita u bazi. Atribute Iznos i Isplaćeno ne unosi korisnik, već se unose automatski. Prilikom otvaranja ugovora, Iznos poprima vrijednost NULL, a kasnije se automatski izračunava prilikom podnošenja ugovora na temelju cijene radnog sata i broja odrađenih sati. Bitovni atribut Isplaćeno prilikom otvaranja ugovora poprima vrijednost 0, što simbolizira ne, odnosno činjenicu da je ugovor aktivan, dok se prilikom zatvaranja ugovora vrijednost mijenja u 1, čime se označava da je ugovor isplaćen.

Automatasko unošenje vrijednosti u tablicu Ugovor napravljeno je korištenjem dva okidača (eng. *trigger*) nazvanih OpenContract i CloseContract koji se aktiviraju na dodavanje novih redaka (*on insert*) ili ažuriranje (*on update*) u tablici Ugovor. U nastavku je prikazan SQL kôd za kreiranje okidača:

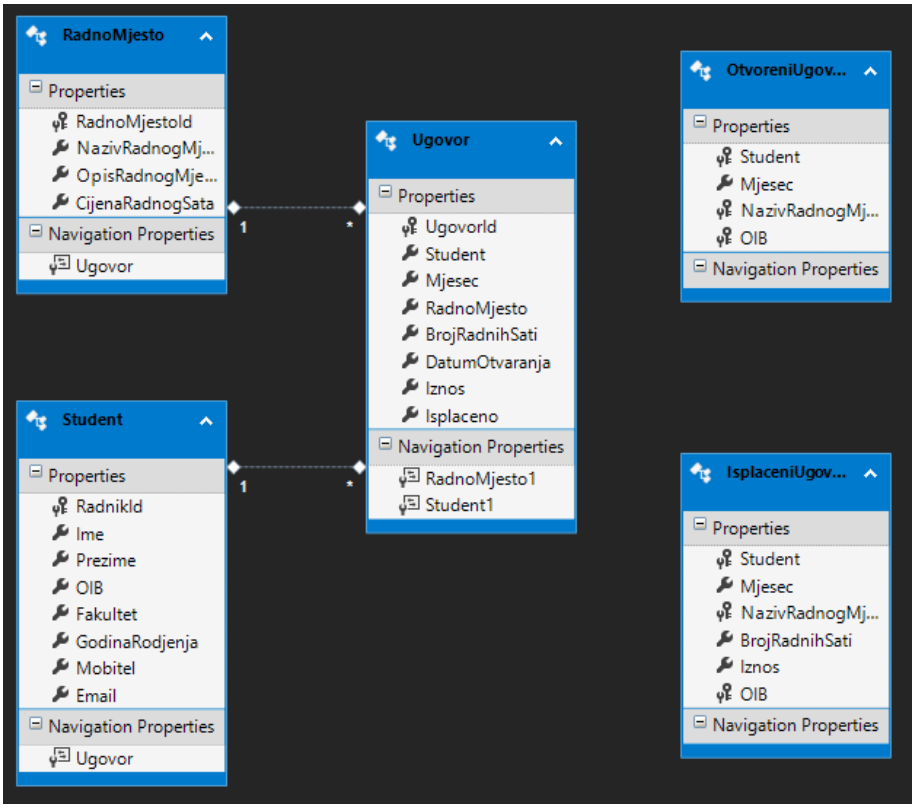
```
CREATE TRIGGER OpenContract ON Ugovor AFTER INSERT
AS
BEGIN
   SET NOCOUNT ON;
```

```
   UPDATE Ugovor SET Isplaceno = 0 WHERE UgovorId =
      (SELECT UgovorId FROM inserted);
   UPDATE Ugovor SET Mjesec = (SELECT CONVERT(varchar, MONTH(DatumOtvaranja))
      + '/' + CONVERT(varchar, YEAR(DatumOtvaranja)));
   WHERE UgovorId = (SELECT UgovorId FROM inserted);
END;
GO
CREATE TRIGGER CloseContract ON Ugovor AFTER INSERT
AS
BEGIN
   SET NOCOUNT ON;
   DECLARE @hrs AS int;
   SELECT @hrs = (SELECT BrojRadnihSati FROM inserted);
   IF @hrs IS NOT NULL
   BEGIN
      UPDATE Ugovor SET Isplaceno = 1 WHERE UgovorId =
         (SELECT UgovorId FROM inserted);
      UPDATE Ugovor SET Mjesec =
         (SELECT CONVERT(varchar, MONTH(DatumOtvaranja))
        + '/' + CONVERT(varchar, YEAR(DatumOtvaranja)))
      WHERE UgovorId = (SELECT UgovorId FROM inserted);
   END;
END;
GO
```

Kako bi se omogućilo odvajanje otvorenih i isplaćenih ugovora, sljedećim SQL naredbama napravljena su dva pogleda koji razvrstavaju ugovore:

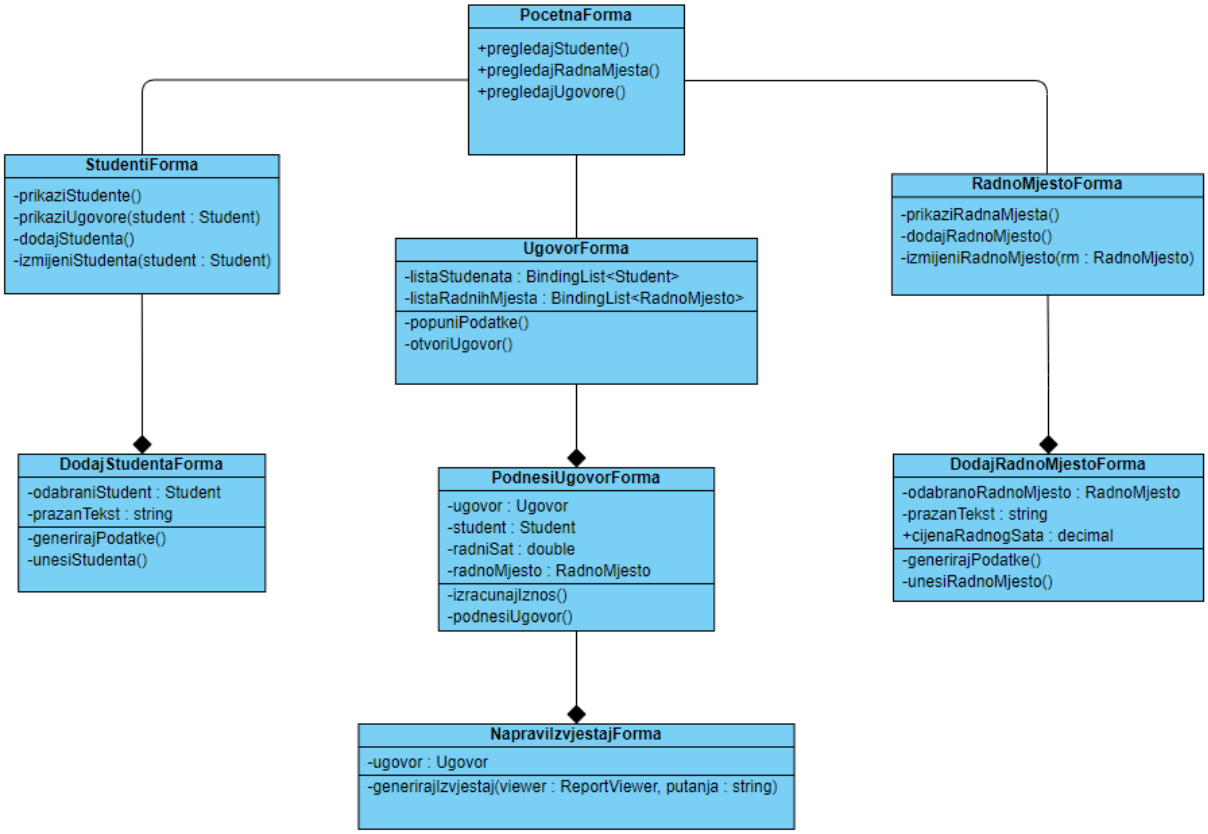
```
CREATE VIEW OtvoreniUgovori AS
   SELECT Student.Ime + ' ' + Student.Prezime AS Student, Student.OIB,
   Ugovor.Mjesec, RadnoMjesto.NazivRadnogMjesta FROM RadnoMjesto
   INNER JOIN Ugovor ON RadnoMjesto.RadnoMjestoId = Ugovor.RadnoMjesto
   INNER JOIN Student ON Ugovor.Student = Student.RadnikId
   WHERE (Ugovor.Isplaceno = 0)
CREATE VIEW IsplaceniUgovori AS
   SELECT Student.Ime + ' ' + Student.Prezime AS Student, Student.OIB,
   Ugovor.Mjesec, RadnoMjesto.NazivRadnogMjesta, Ugovor.BrojRadnihSati,
   CONVERT(varchar, Ugovor.Iznos) + ' kn' AS Iznos FROM RadnoMjesto
   INNER JOIN Ugovor ON RadnoMjesto.RadnoMjestoId = Ugovor.RadnoMjesto
   INNER JOIN Student ON Ugovor.Student = Student.RadnikId
   WHERE (Ugovor.Isplaceno = 1)
```

Po dovršetku baze, potrebno je spojiti bazu s aplikacijom, gdje je kao način spajanja odabran Entity Framework. Entity Framework prilikom kreiranja konekcije nudi četiri mogućnosti: za novu bazu podataka moguće su opcije prvo model i prvo kôd – nova baza, dok su za spajanje na već izrađenu bazu ponuđene opcije prvo baza podataka i prvo kôd – postojeća baza. Kako je u ovom projektu baza podataka kreirana prije početka programiranja, odabir je bio između druge dvije mogućnosti, pri čemu je odlučeno koristiti pristup prvo baza podataka (eng. *Database First*). Ovim načinom spajanja klase su u rješenju projekta generirane automatski, nije ih potrebno ručno pisati, odnosno klase su temeljene na tablicama u bazi. Nakon što je aplikacija spojena s bazom podataka, u Visual Studiju je generiran entitetni podatkovni model temeljen na bazi, prikazan na sljedećoj slici:



Slika 16. Generirani model podataka (autorska izrada, isječak iz alata MS Visual Studio)

Nakon spajanja na bazu podataka, potrebno je implementirati forme i njihovu navigaciju kroz aplikaciju. Konceptualni dijagram klasa za forme sa slike 13 proširen je zasebnim klasama za operacije nad studentima i radnm mjestima koje zahtijevaju povezivanje s bazom podataka. Veze s novim klasama također su prikazane kao kompozicija, budući da se ovisno o ulaznom parametru pokreću na temelju instance klase s kojom su povezane. Prilikom dodavanja novog studenta ili radnog mjesta, klase za dodavanje pokreću se bez ulaznog parametra, dok prilikom uređivanja zapisa koriste označenog studenta ili radno mjesto kao ulazni parametar. Nova, detaljna verzija dijagrama klasa prikazana je na sljedećoj slici:

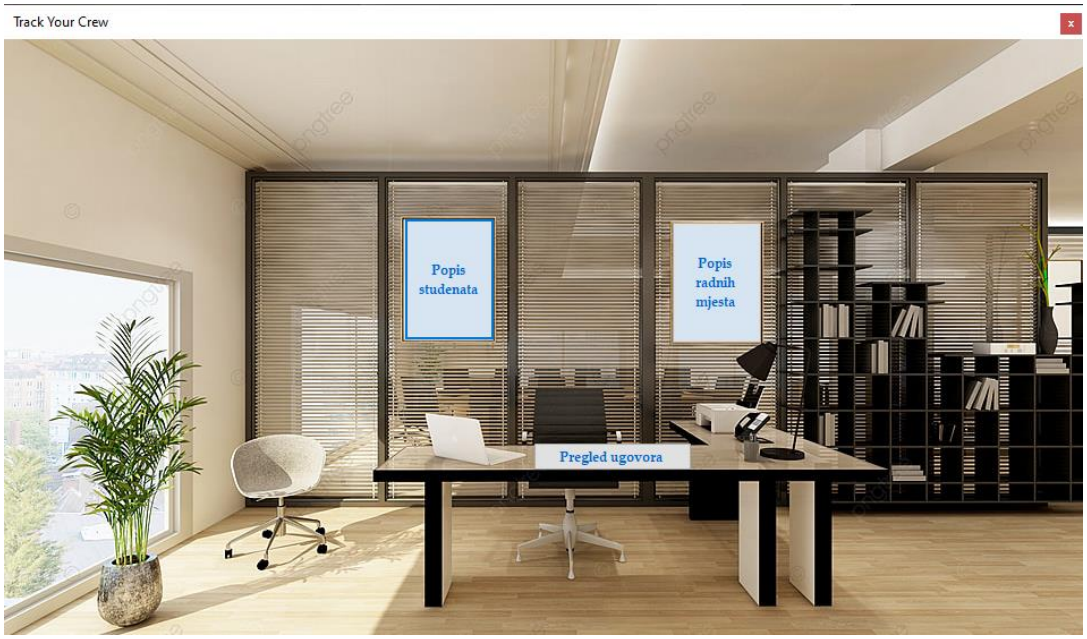


Slika 17. Ažurirani dijagram klasa (autorska izrada)

Ovaj način implementacije formi objašnjen je u sljedećem isječku koda iz klase DodajStudenta:

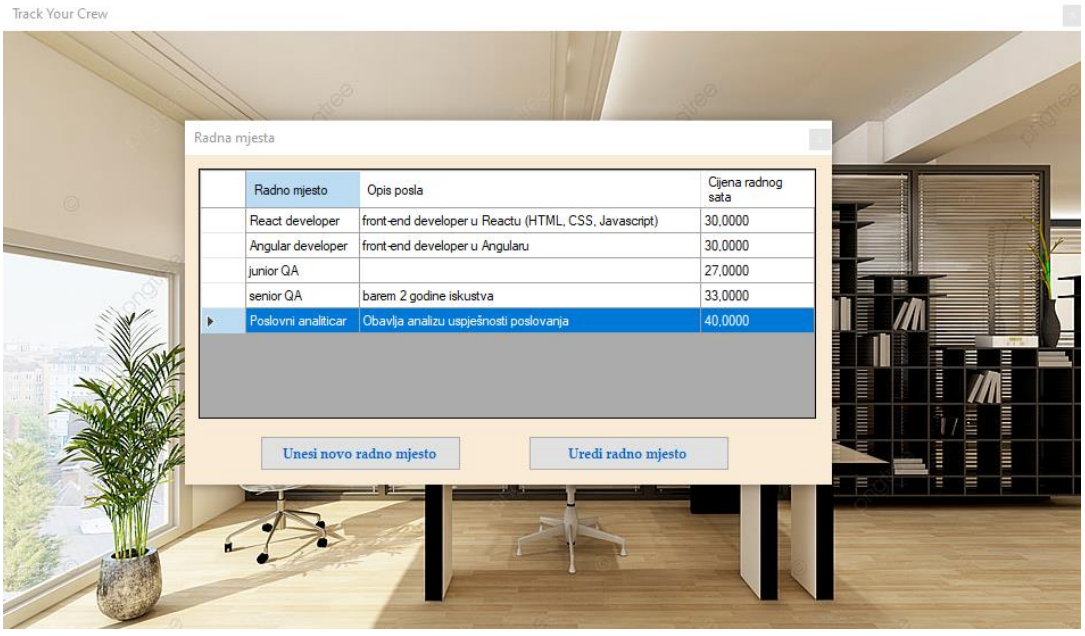
```
private Student selectedStudent;
public DodajStudenta()
{
   InitializeComponent();
}
public DodajStudenta(Student student)
{
   InitializeComponent();
   selectedStudent = student;
}
```

Prikazanim isječkom koda vidljivo je da postoje dva konstruktora, s parametrom i bez njega. Konstruktor bez parametra poziva se prilikom dodavanja studenta, dok se konstruktor s parametrom poziva prilikom uređivanja podataka o studentu, gdje je odabrani student ulazni parametar. Na identičan način implementiran je konstruktor za formu RadnoMjesto te se aplikacija može pokrenuti. Prvi korak pri pokretanju aplikacije prikaz je početne forme na zaslonu, kako je prikazano na slici:



Slika 18. Pokrenuta aplikacija (autorska izrada, isječak iz alata MS Visual Studio)

Za testiranje su u bazu podataka unešeni podaci za nekoliko studenata i radnih mjesta. Sljedeća slika prikazuje otvaranje forme RadnaMjesta pritiskom na gumb „Popis radnih mjesta“. Identično je za gumbe „Popis studenata“ i „Pregled ugovora“, s tim da potonji prikazuje praznu formu, budući da je implementacija ugovora predviđena za treći skup funkcionalnosti.

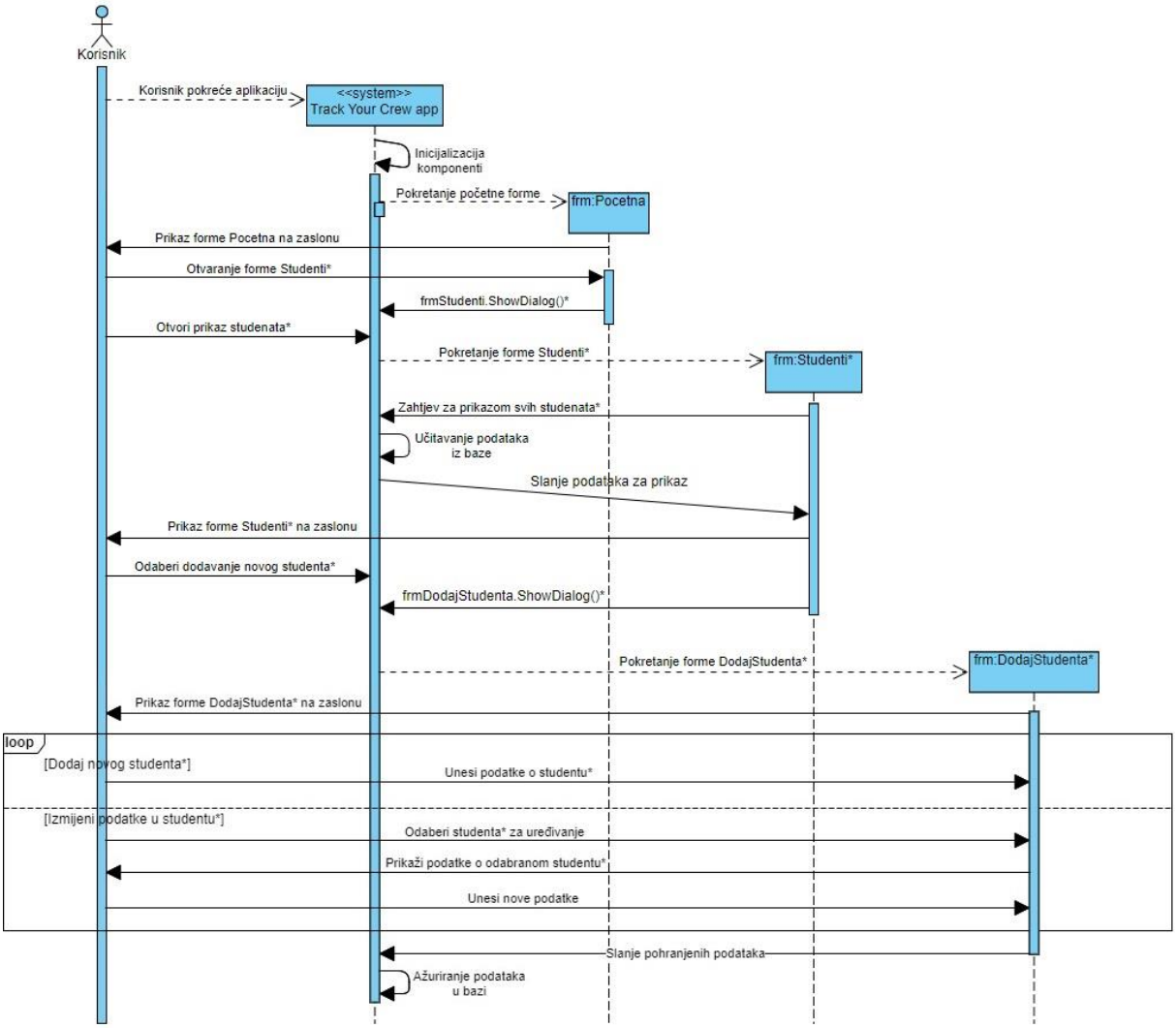


Slika 19. Pregled radnih mjesta (autorska izrada, isječak iz alata MS Visual Studio)

Kako se sve forme u aplikaciji ispravno pokreću, a podaci o radnim mjestima prikazani, potvrđeno je da je dijagram klasa uspješno implementiran, kao i konekcija na bazu podataka. Stoga glavni programer označava skup funkcionalnosti dovršenim te se prelazi na idući skup.

7.5. Skup funkcionalnosti „dodavanje studenata i radnih mjesta u sustav“

U ovom skupu implementirat će se operacije nad bazom podataka za studente i radna mjesta. Kako su funkcionalnosti dodavanja i uređivanja studenata kompleksnije od implementacije navigiranja između formi, napravljen je dijagram slijeda:

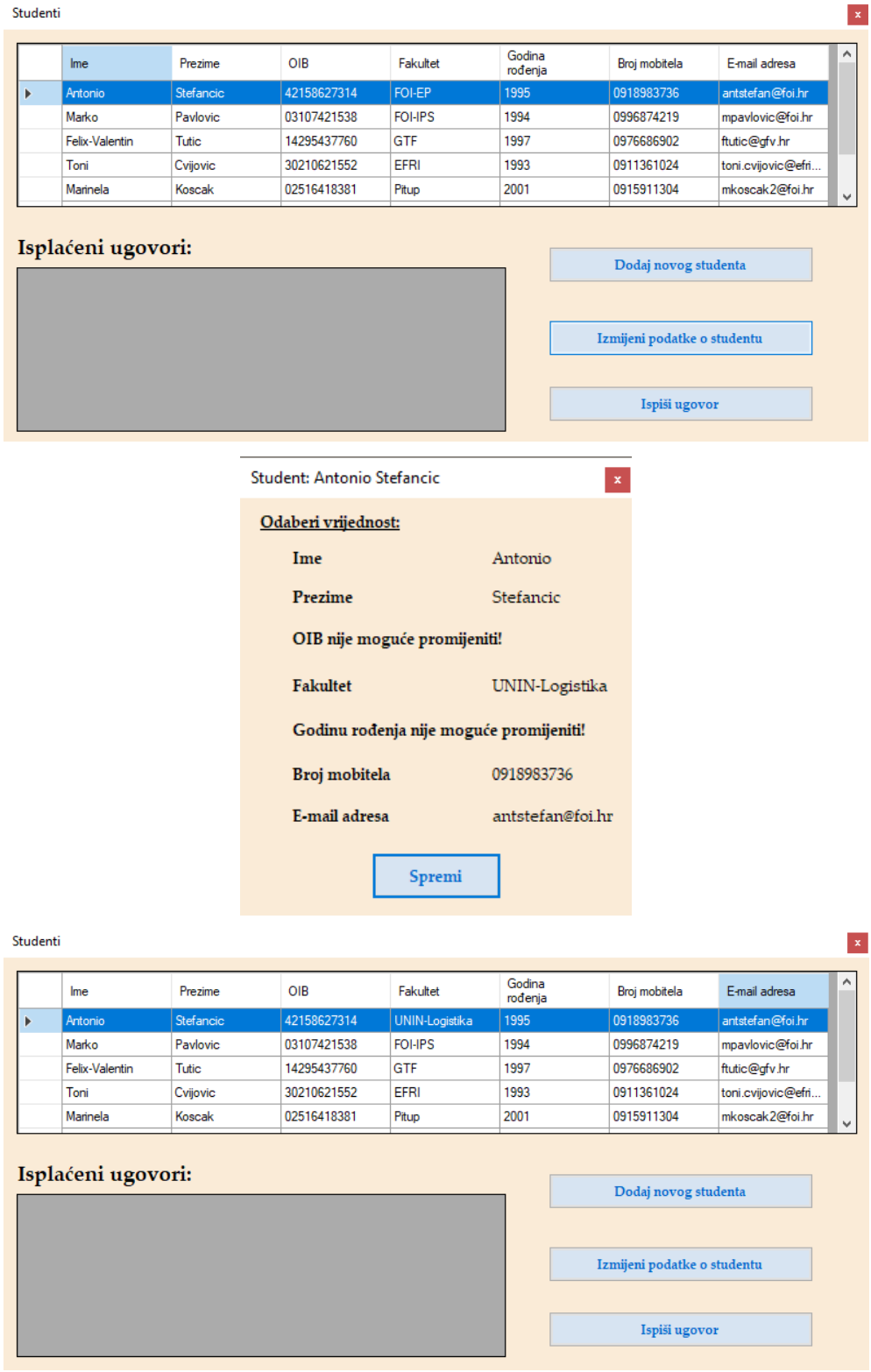


Slika 20. Dijagram slijeda za funkcionalnost „Unijeti novog ili izmijeniti podatke o postojećem studentu“ (autorska izrada)

Dijagram opisuje interakciju korisnika i aplikacije prilikom dodavanja ili izmjene studenta. Programsko rješenje implementacije studenta napravljeno je pomoću iznimke koja se aktivira na ažuriranje baze podataka i koja bi sprječavala prekid programa ako se unesu dva studenta s istim OIB-om:

```
try
{
   using (var db = new RadniSatiEntities())
   {
      if (selectedStudent == null)
      { /*Ako se kreira novi student*/
         if (/*uvjeti za ispunjenost forme*/)
         {
            Student student = new Student {/*popunjavanje atributa*/};
            db.Student.Add(student);
            db.SaveChanges();
            Close();
         }
         else{
            /*MessageBox upozorenja o neispravnim podacima*/
         }
      }
      else /*Učitavaju se podaci odabranog studenta za uređivanje*/
      {
         if (/*uvjeti za ispunjenost forme*/)
         {
            db.Student.Attach(selectedStudent);
            /*Čitanje novih vrijednosti s forme*/
            db.SaveChanges();
            Close();
         }
         else
         {
            /*MessageBox upozorenja o neispravnim podacima*/
         }
      }
   }
}
catch (DbUpdateException)
{ /*MessageBox upozorenja o identičnom OIB-u*/ }
```

Na trodijelnoj slici slijedi prikaz implementirane funkcionalnosti u aplikaciji:



Slika 21. Uređivanje studenta (autorska izrada, isječak iz alata MS Visual Studio)

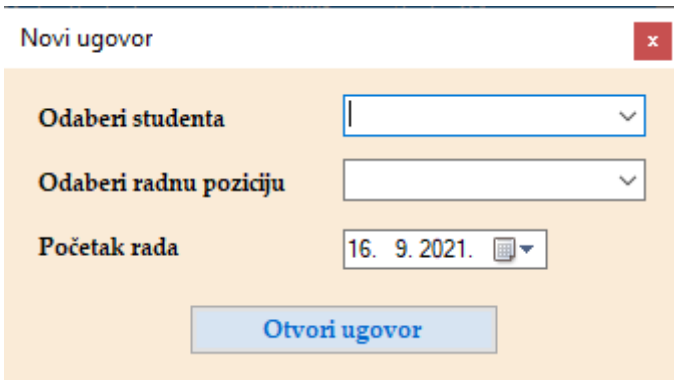
Označeni student nedavno se prebacio na drugi fakultet, što je potrebno promijeniti u sustavu. Promjena je izvršena te se nakon spremanja novi fakultet prikazuje u tablici.

Dijagram slijeda za funkcionalnost dodavanja i uređivanja radnih mjesta izgleda identično, uz razliku da se na mjestima označenim znakom * umjesto Student upisuje RadnoMjesto. Također je funkcionalnost za radna mjesta implementirana na identičan način te testirano uspješna, tako da se i ovaj skup funkcionalnosti može smatrati dovršenim.

7.6. Skup funkcionalnosti „potpisivanje ugovora“

Otvaranje ugovora zahtijeva omogućen odabir studenta, radnog mjesta na kojemu radi i mjeseca u kojemu se odvija rad. Prilikom zatvaranja upisuje se broj sati te se automatski izračunava iznos zarade u ugovoru.

Prilikom izrade funkcionalnosti „učitati popis svih studenata i ugovora iz baze podataka“, uočeno je da na dijagramu klasa nedostaje forma za unos novog ugovora, stoga je potrebno kreirati novu formu. Za otvaranje ugovora potrebno je učitati studente i radna mjesta iz baze podataka te omogućiti kalendar s odabirom datuma, pri čemu se mjesec rada iščitava iz datuma otvaranja ugovora. Forma za otvaranje ugovora napravljena je od dva ComboBoxa u kojima se izlistaju svi studenti i sva radna mjesta u bazi podataka, dok je za odabir datuma stavljen DatePicker, pri čemu je početni (eng. *default*) datum današnji:



Slika 22. Otvaranje novog ugovora (autorska izrada, isječak iz alata MS Visual Studio)

Vrijednosti se u ComboBoxove učitavaju metodom FillTheCombo(), koja se poziva kod učitavanja forme:

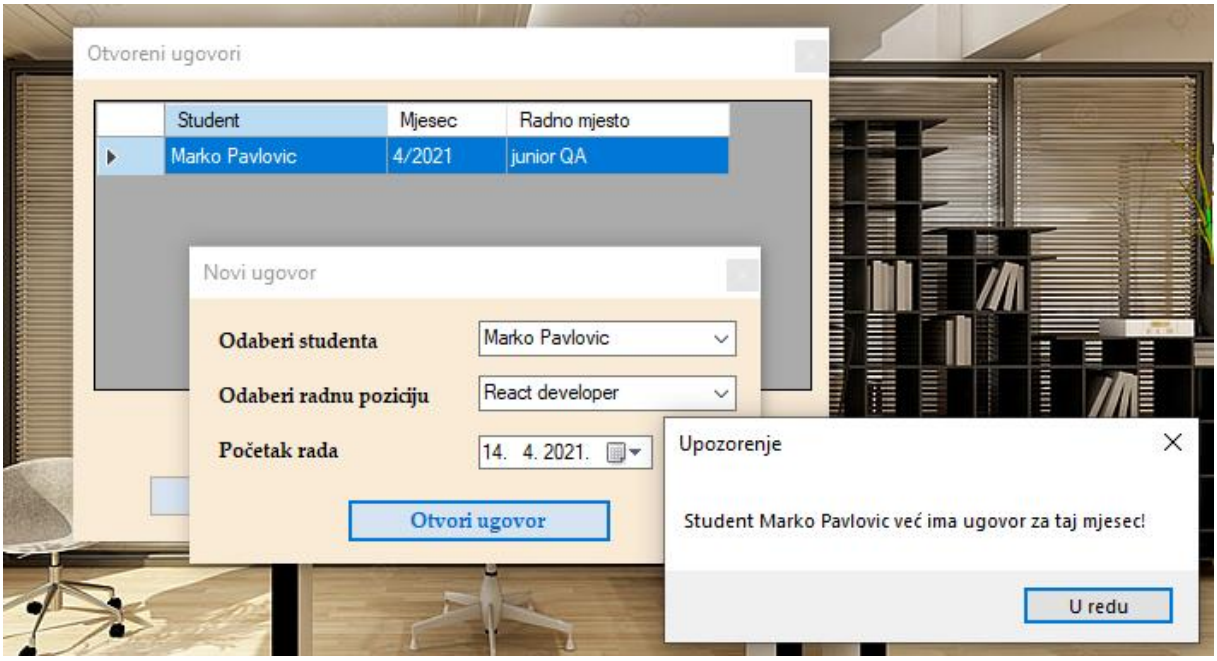
```
private BindingList<Student> listaStudenata;
private BindingList<RadnoMjesto> listaRM;
private void FillTheCombo()
{
   listaStudenata = null;
   listaRM = null;
```

```
    using (var db = new RadniSatiEntities())
    {
        listaStudenata = new BindingList<Student>(db.Student.ToList());
        listaRM = new BindingList<RadnoMjesto>(db.RadnoMjesto.ToList());
    }
    foreach (Student i in listaStudenata)
    {
        ComboBoxItem item = new ComboBoxItem
        {
            Value = i.RadnikId,
            Text = i.Ime + ' ' + i.Prezime
        };
        cbStudent.Items.Add(item);
    }
    foreach (RadnoMjesto i in listaRM)
    {
        ComboBoxItem item = new ComboBoxItem
        {
            Value = i.RadnoMjestoId,
            Text = i.NazivRadnogMjesta
        };
        cbRadnoMjesto.Items.Add(item);
    }
    dtp.Value = DateTime.Today;
}
```

Sljedeća funkcionalnost je ograničenje o jednom ugovoru po studentu mjesečno. U bazi podataka to je automatski osigurano stavljanjem složenog primarnog ključa u tablicu Ugovori. Kao i kod unošenja OIB-a, potrebno je osigurati da se aplikacija ne sruši ukoliko korisnik pokuša izvaditi ugovor za studenta u mjesecu u kojem već ima ugovor, što je riješeno iznimkom za ažuriranje baze podataka:

```
try
{
    /*Odabir studenta, radnog mjesta i početka rada*/
}
catch (DbUpdateException)
{
    /*MessageBox koji šalje poruku da odabrani student već ima ugovor za
    odabrani mjesec.*/
}
```

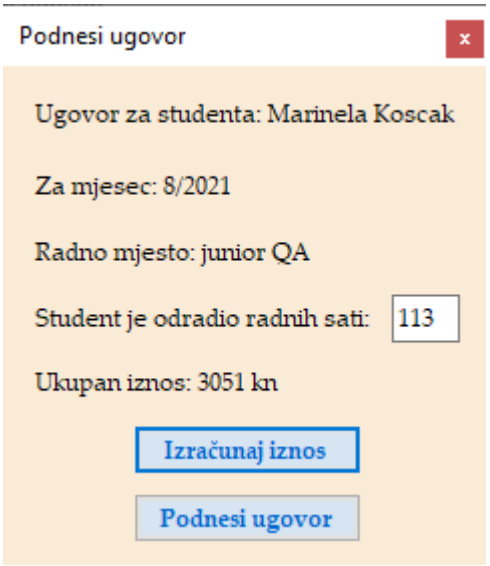
Na sljedećoj slici prikazana je aktivacija iznimke:



Slika 23. Iznimka za dvostruki ugovor (autorska izrada, isječak iz alata MS Visual Studio)

Student Marko Pavlović već ima otvoren ugovor za mjesec travanj 2021. godine. Prilikom novog otvaranja odabrano je drugo radno mjesto, no aplikacija ne dozvoljava otvaranje ugovora. Kako je i ova funkcionalnost dovršena, prelazi se na zatvaranje ugovora.

Forma za podnošenje ugovora napravljena je na način da se unosi broj sati, dok se cijena radnog sata iščitava iz baze podataka za radno mjesto koje je navedeno u ugovoru. Zbog pravila student servisa, stavljeno je ograničenje od najviše 160 radnih sati mjesečno. Prije podnošenja ugovora, moguće je provjeriti iznos pritiskom na gumb „Izračunaj iznos“.

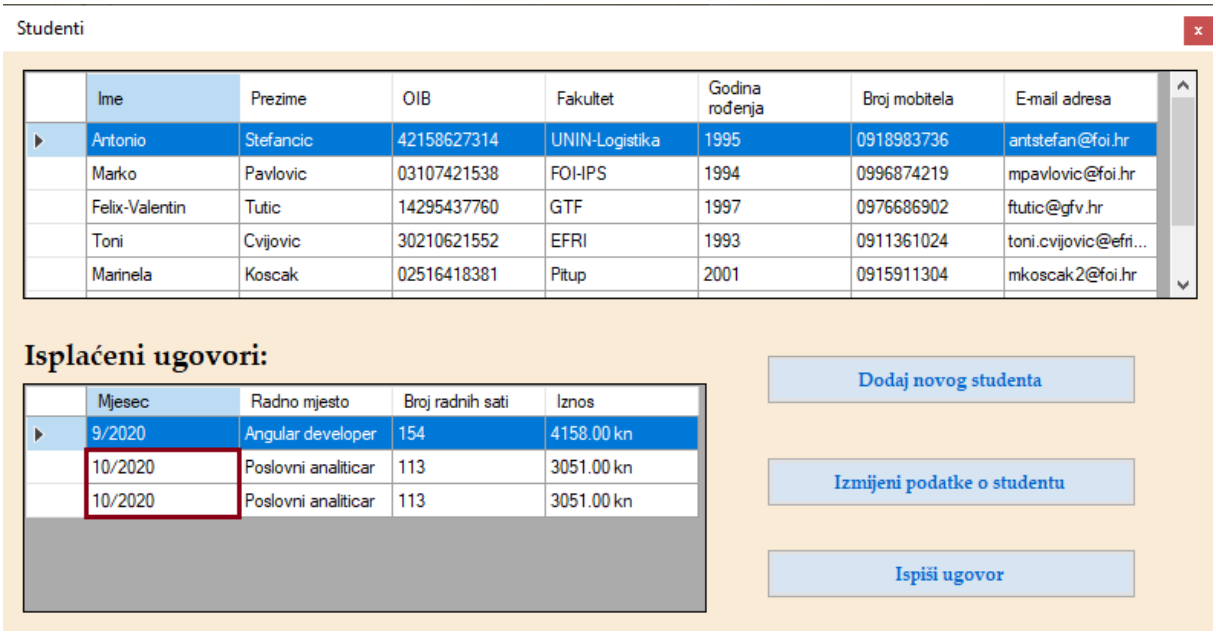


Slika 24. Forma za podnošenje ugovora (autorska izrada)

Provjera radnih sati implementirana je funkcijom provjeriBrojSati() povratnog tipa bool, koja vraća vrijednost TRUE ako su sati unutar ograničenja ili javlja poruku upozorenja ako se pređe gore spomenuti limit. Funkcija prvotno nije stavljena u dijagram klasa te je naknadno dodana u kôd forme, a poziva se kod izračuna iznosa, kao i kod podnošenja ugovora.

```
private bool provjeriBrojSati(int brSati)
{
   if (brSati > 160)
   {
      MessageBox.Show("Student ne smije raditi preko 160 sati mjesečno!");
      return false;
   }
   else
   {
      return true;
   }
}
```

Budući da je podnošenje ugovora implementirano, funkcionalnost se daje glavnom programeru na testiranje. Glavni programer uočava sljedeću grešku:



Slika 25. Nepravilan prikaz isplaćenih ugovora (autorska izrada)

Naime, prikazano je da je dvaput isplaćen identičan ugovor, pri čemu su dva moguća razloga: dvaput je podnesen identičan ugovor jer sustav nije spriječio otvaranje, što označava pogrešku u bazi podataka, ili je greška u tabličnom prikazu, što je pak pogreška u programskom kodu aplikacije. Kako bi se otkrio uzrok, potrebno je vratiti se u dizajn funkcionalnosti „zatvoriti ugovor“. Najprije se provjerava valjanost baze podataka, testira se pogled IsplaceniUgovori upitom koji bi trebao ispisati ugovore za studenta s odabranim OIB-om:

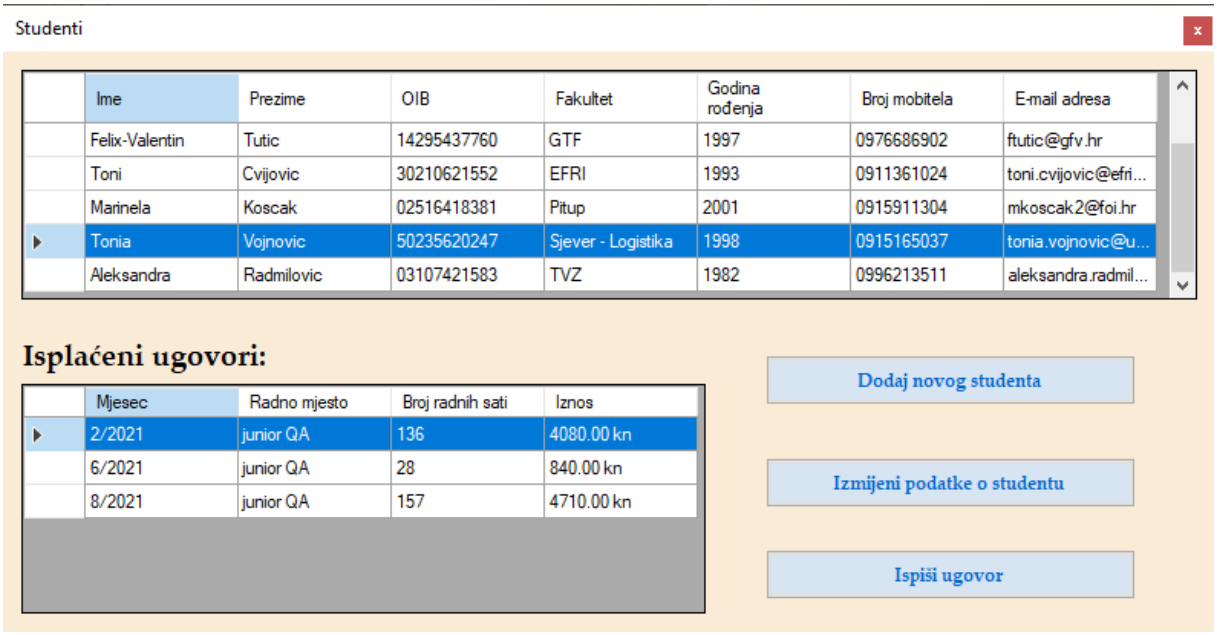
```
SELECT * FROM IsplaceniUgovori WHERE OIB='42158627314';
```

Upit je dao sljedeći rezultat:

	Student	OIB	Mjesec	NazivRadnogMjesta	BrojRadnihSati	Iznos
1	Antonio Stefancic	42158627314	9/2020	Angular developer	154	4158.00 kn
2	Antonio Stefancic	42158627314	10/2020	Poslovni analiticar	113	3051.00 kn
3	Antonio Stefancic	42158627314	6/2021	Poslovni analiticar	160	4320.00 kn

Slika 26. Pogled za isplaćene ugovore studenta u bazi podataka (autorska izrada)

U bazi podataka prikazano je da je student odradio tri ugovora za različite mjesece. Usporedbom prikaza za ostale studente uočeno je da je ispisan ispravan broj podignutih ugovora, ali se u slučaju identičnog radnog mjesta ponavljaju podaci s prvog podignutog ugovora. Debugiranjem koda uočeno je da pogreška nastaje već prilikom učitavanja isplaćenih ugovora iz baze, odnosno ne razlikuju se unosi za isto radno mjesto. Testiranjem aplikacije uočeno je da se identična stvar događa s popisom otvorenih ugovora na formi Ugovori. Proučavanjem problema i mogućih uzroka otkriveno je za ispravan dohvat podataka iz baze potrebno imati jedinstven identifikator, odnosno primarni ključ, što pogled (eng. *view*) nema, te je problem riješen na način da su pogledi zamijenjeni istoimenim tablicama. Na sljedećoj slici prikazuje se ispis isplaćenih ugovora nakon korekcije baze podataka:



Slika 27. Nepravilan prikaz isplaćenih ugovora (autorska izrada)

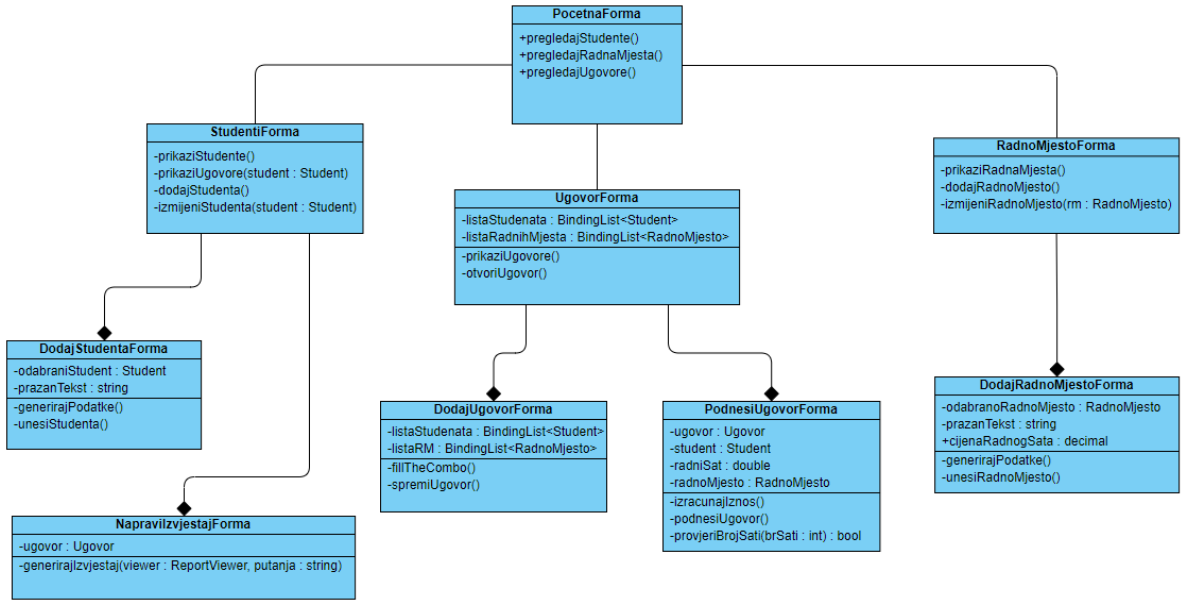
Kako je u svim ugovorima kao radno mjesto navedeno „poslovni analitičar“, a mjeseci rada i radni sati su različiti, greška je otklonjena te glavni programer označava funkcionalnost zatvaranja ugovora kao dovršenu. Kako bi se osiguralo da se obrišu ugovori iz tablice

OtvoreniUgovori prilikom otkazivanja ugovora, potrebno je u bazi podataka napraviti novi okidač koji bi služio za brisanje zapisa iz tablice OtvoreniUgovori, a aktivirao bi se na brisanje redova iz tablice Ugovor:

```
create trigger DeleteContract on Ugovor for delete as
begin
   declare @student as varchar(13);
   select @student=(select OIB from Student where
      Student.RadnikId=(select deleted.Student from deleted));
   delete from OtvoreniUgovori where
      Mjesec=(select deleted.Mjesec from deleted) and OIB=@student;
end;
go
```

Za spravak pogrešnog ispisa ugovora bila su potrebna dva radna sata, odnosno pola radnog dana prema planu razvoja, što pomiče očekivani završetak projekta nakon pola radnog dana na datum 17.7.2021. Potom se prelazi na posljednju funkcionalnost iz ovoga skupa, a to je ispis izvješća.

Već u početku izrade klase za prikaz izvješća uočena je nova pogreška. Iako se u dijagramu klasa forma za ispis izvješća pokreće iz forme koja prikazuje ugovore, izvješće bi se trebalo pokretati s forme za studente, budući da se na formi Student prikazuju već isplaćeni ugovori. Stoga je prije izrade izvješća ažuriran dijagram klasa, pri čemu su dodane prethodno spomenute forma DodajUgovor i metoda provjeriBrojSati() u formi PodnesiUgovor te je ispravljena veza na klasu NapraviIzvjestaj:



Slika 28. Ažurirani dijagram klasa (autorska izrada)

Nakon korekcije dijagrama klasa, može se započeti s implementacijom izvješća. Za prikaz izvješća korištena je kontrola Report Viewer koja se u okviru forme prikazuje na temelju odabranog ugovora. Detalji koji se prikazuju su ime i prezime i OIB studenta te radno mjesto na kojemu je student radio, uz broj odrađenih sati i iznos plaće. U zaglavlju je istaknuto u kojem je mjesecu rad obavljen, dok je u podnožju vrijeme ispisa ugovora. Prikaz izvješća vidljiv je na sljedećoj slici:

Slika 29. Izvješće isplaćenog ugovora (autorska izrada)

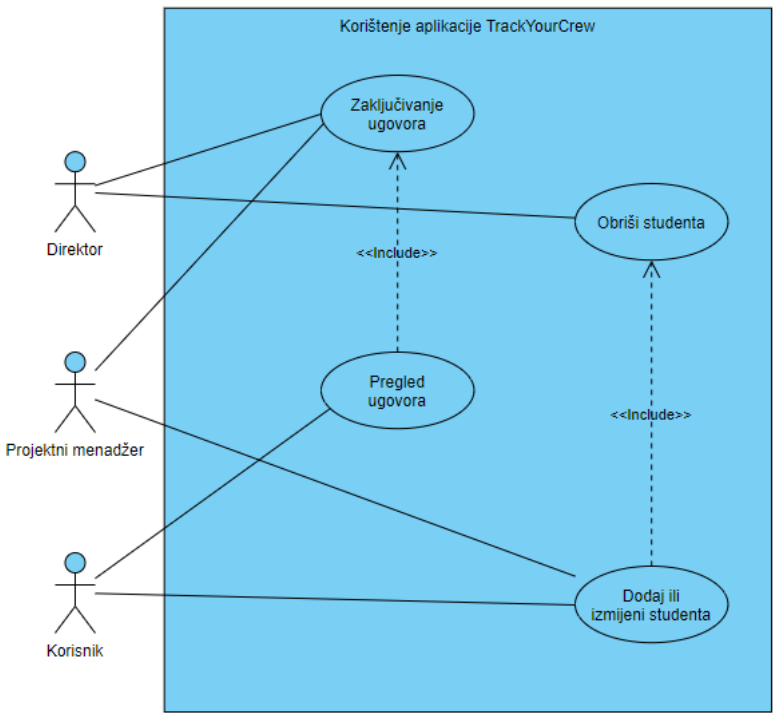
Pritiskom na gumb „Spremi“ izvješće se ispisuje u PDF dokument, koji se pohranjuje u folder preuzimanja (eng. *downloads*) na računalu. Kako je izrada izvješća uspješna, skup funkcionalnosti “potpisivanje ugovora“ označava se dovršenim.

Ispravak dijagrama klasa također je uzeo pola radnog dana, čime bi projekt inicijalno završio na kraju dana 17.7.2021., odnosno na zadnji dan zadanog roka. Unatoč tome, na predzadnji dan projekta stigao je zahtjev za novom funkcionalnosti sustava, podjelom na korisničke uloge.

7.7. Izmjena u korisničkim zahtjevima

Tokom predzadnjeg dana trajanja projekta, 15.7.2021., došao je zahtjev od strane naručitelja za implementacijom korisničkih uloga u aplikaciji. Određena su tri tipa korisničkih uloga: direktor, projektni menadžer i zaposlenik. Direktor ima sve ovlasti, uključujući i brisanje studenata iz baze studenata, pri čemu se mogu obrisati samo oni studenti koji nemaju niti jedan isplaćen ugovor. Projektni menadžer nema mogućnost brisanja studenata, ali može podnositi i otkazivati otvorene ugovore. Zaposlenik je uloga s kojom se prijavljuju svi ostali zaposlenici organizacije koja koristi aplikaciju, može dodavati nove studente i radna mjesta te otvarati ugovore, ali ih ne može isplaćivati niti otkazivati. Rok za implementaciju je tri radna dana, što znači da funkcionalnost, a tako i nova verzija aplikacije, mora biti gotova 22.2.2021.

Kako je riječ o novim korisničkim zahtjevima, a tako i o funkcionalnostima koje nisu uključene u prethodni postupak razvoja sustava, potrebno je opet se vratiti na prvi korak FDD-a, razvoj konceptualnog modela, čemu je poslužio sljedeći dijagram slučajeva korištenja, koji prikazuje funkcionalnosti koje se razlikuju po ulogama:



Slika 30. Dijagram slučajeva korištenja za korisničke uloge (autorska izrada)

Potom se izrađuje popis funkcionalnosti. Skupom funkcionalnosti proglašena je podjela na korisničke uloge, a nove funkcionalnosti su sljedeće:

- Ažuriraj bazu podataka tablicom korisničkih uloga.
- Kreiraj formu za prijavu ovisno o različitim ulogama.
- Ovisno o ulozi prijave, o(ne)mogući određene funkcionalnosti.

Sve funkcionalnosti su od ključne važnosti za aplikaciju, odnosno sve su označene prioritetom A, stoga prioritet nije posebno naglašen kod izrade popisa. U sljedećoj tablici prikazan je plan projekta za implementaciju novog skupa funkcionalnosti:

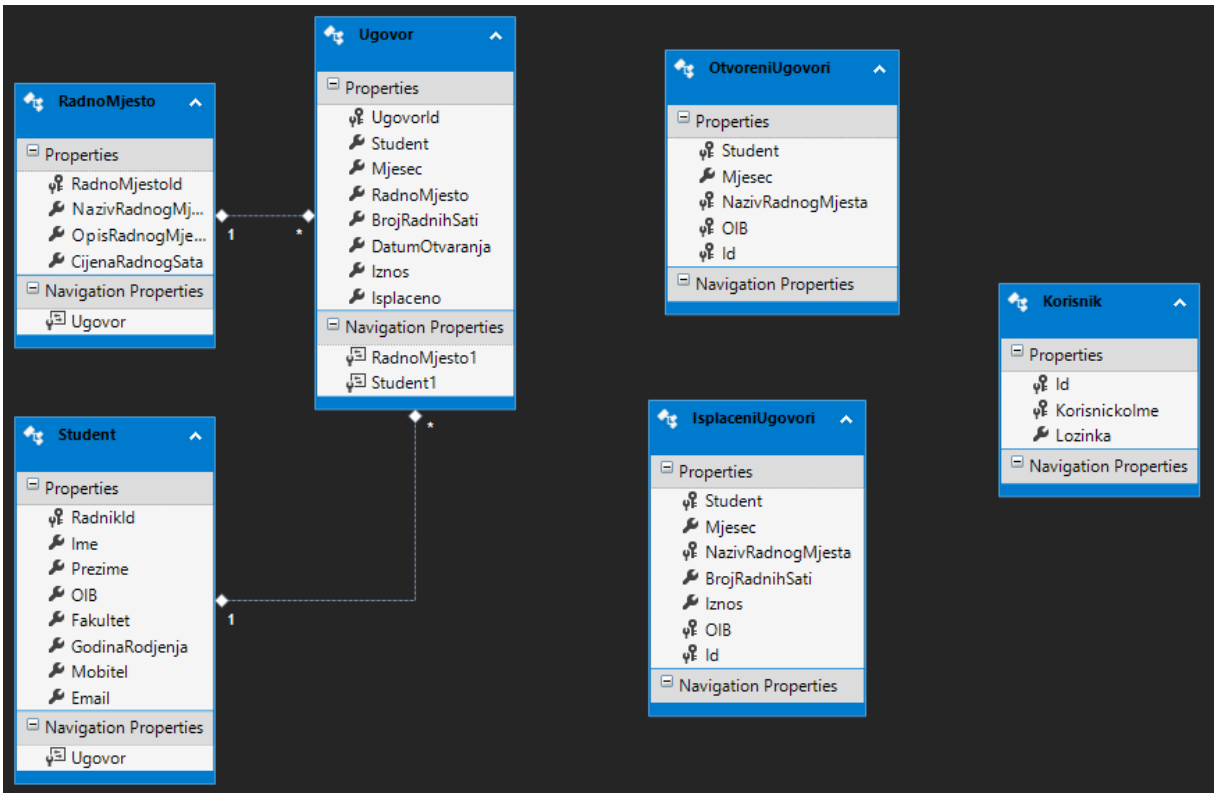
Tablica 3. Vremenski plan projekta za novu funkcionalnost

Funkcionalnost	Datum početka	Datum završetka	Trajanje u danima
Ažurirati bazu podataka.	21.2.2021.	21.2.2021.	0,5
Kreirati formu za prijavu.	21.2.2021.	22.2.2021.	1
O(ne)mogućiti korisničke funkcionalnosti.	22.2.2021.	22.2.2021.	0,5
Ukupno	**21.2.2021.**	**22.2.2021.**	**2**

Izvor: autorska izrada

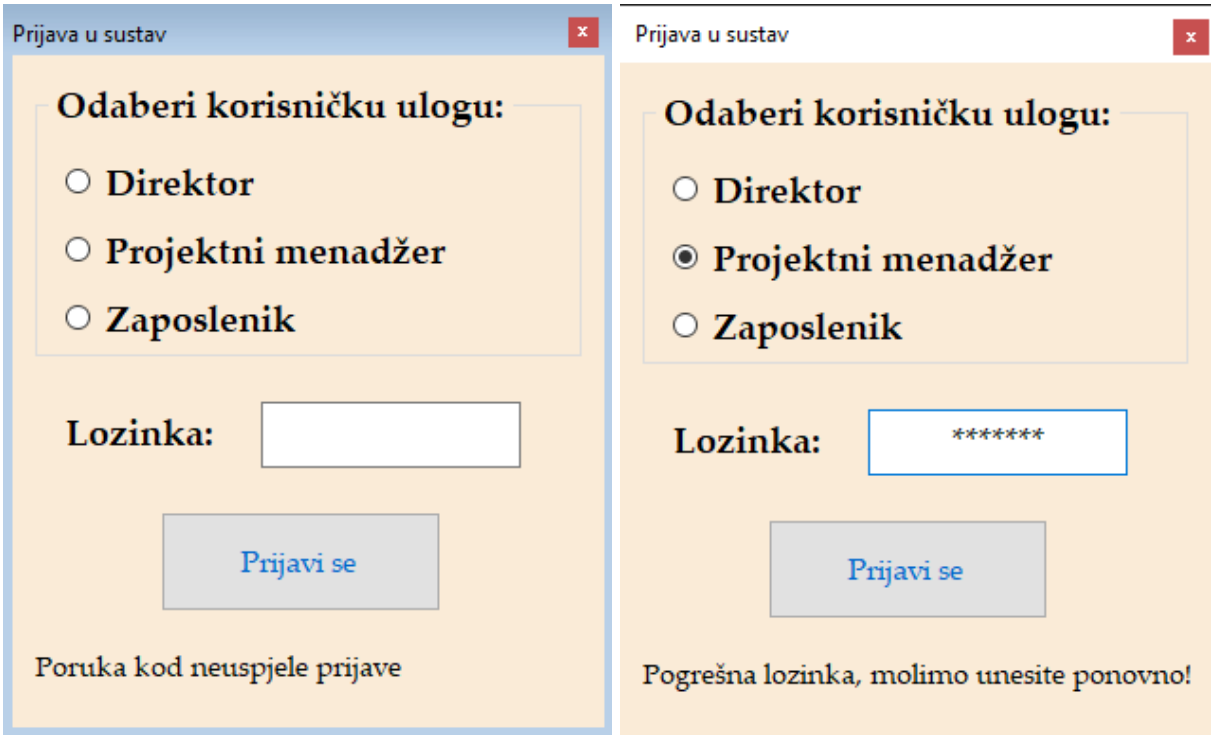
Jedan radni dan (petak, 18.2.2021.) iskorišten je na razvoj konceptualnog modela i izradu popisa funkcionalnosti, te sastavljanjem tima za planiranje. Tim je napravio plan implementacije skupa funkcionalnosti koja započinje 21.2.2021. i završava 22.2.2021., što za implementaciju daje potrebno vrijeme od osam radnih sati.

Prva funkcionalnost iz skupa napravljena je kreiranjem nove tablice nazvane Korisnik, koja se sastoji od dva atributa: korisnička uloga i lozinka. Sukladno ažuriranju u MS SQL Management Studiju, isto je potrebno provesti i u modelu podataka u rješenju aplikacije, kako bi se podaci iz baze mogli primijeniti u aplikaciji:



Slika 31. Ažurirani podatkovni model (autorska izrada)

Potom se kreira nova forma na kojoj se preko kontrole tipa RadioButton odabire uloga za prijavu, za koju se potom unosi i provjerava lozinka:



Slika 32. Forma za prijavu u aplikaciju (autorska izrada)

Lijevi dio slike prikazuje dizajn forme za prijavu u Visual Studiju, dok je na desnoj polovici prikaz pokrenute forme. U slučaju pogreške kod unosa lozinke. u donjem lijevom kut u ispisuje se poruka da je lozinka netočna. Kao početna uloga, ona koja bude automatski označena prilikom pokretanja aplikacije, odabrana je uloga zaposlenik.

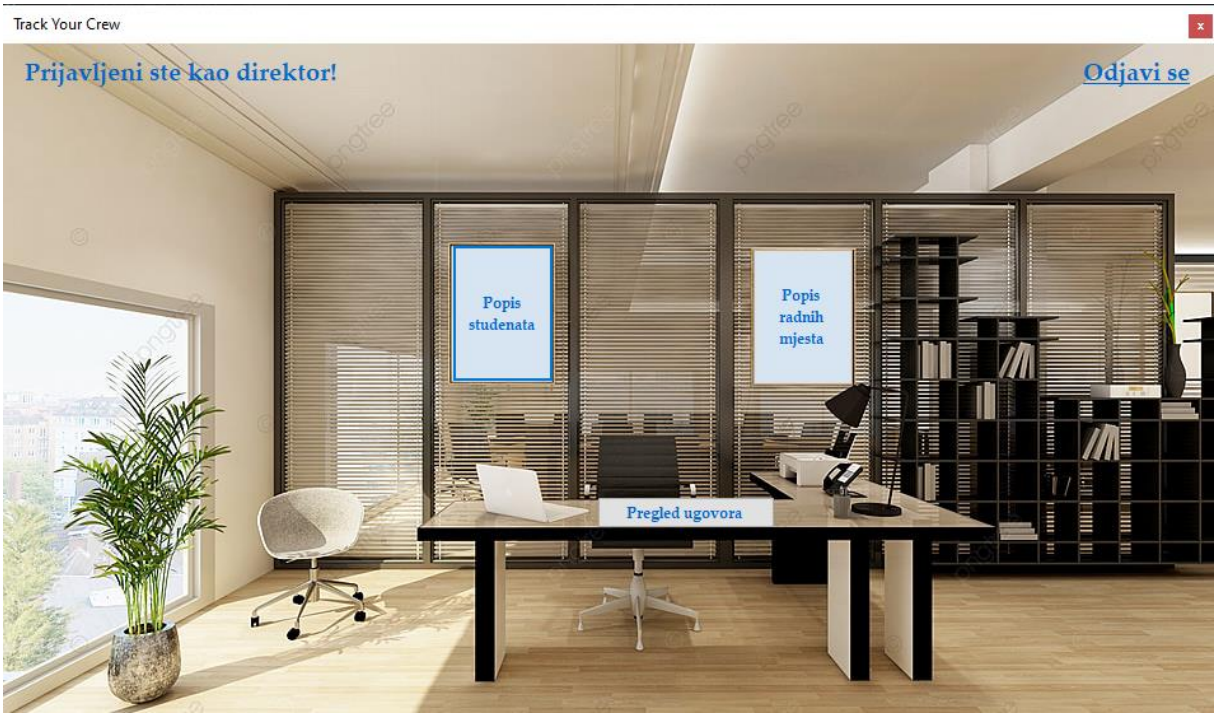
Kako se u novoj verziji programa ova forma prva prikazuje po pokretanju aplikacije, potrebno je u klasi Program.cs konfigurirati početnu formu, tako da se linija koda

```
Application.Run(new Pocetna());
```

izmijeni u

```
Application.Run(new LoginForm());
```

Forma za prijavu je uspješno kreirana i implementirana te se započinje s dodjelom ovlasti u aplikaciji ovisno o ulozi u koju se korisnik prijavio. Ovisno o ulozi u koju se korisnik prijavio, na zaslonu se ispisuje poruka kako bi korisnici uvijek znali koju ulogu koriste, što može biti korisno ako aplikacija ostane otvorena dulje vrijeme. Također postoji opcija odjave, čime se ponovno prikazuje forma za prijavu:



Slika 33. Naslovna forma nakon implementacije uloga (autorska izrada)

Iako se forma zove Pocetna, više nema ulogu početne forme, stoga će u nastavku biti navedena kao naslovna forma, koja joj je i uloga.

U konstruktore formi Pocetna, Studenti i Ugovori dodane su varijable tipa string iz kojih se iščitava korisnička uloga. Primjer koda za formu Pocetna:

```
public partial class Pocetna : Form
{
   private string korisnik;
   public Pocetna(string user)
   {
      InitializeComponent();
      korisnik = user;
   }
   /*Metode u klasi*/
}
```

Naslovna forma se iz forme za prijavu poziva prosljeđivanjem naziva uloge:

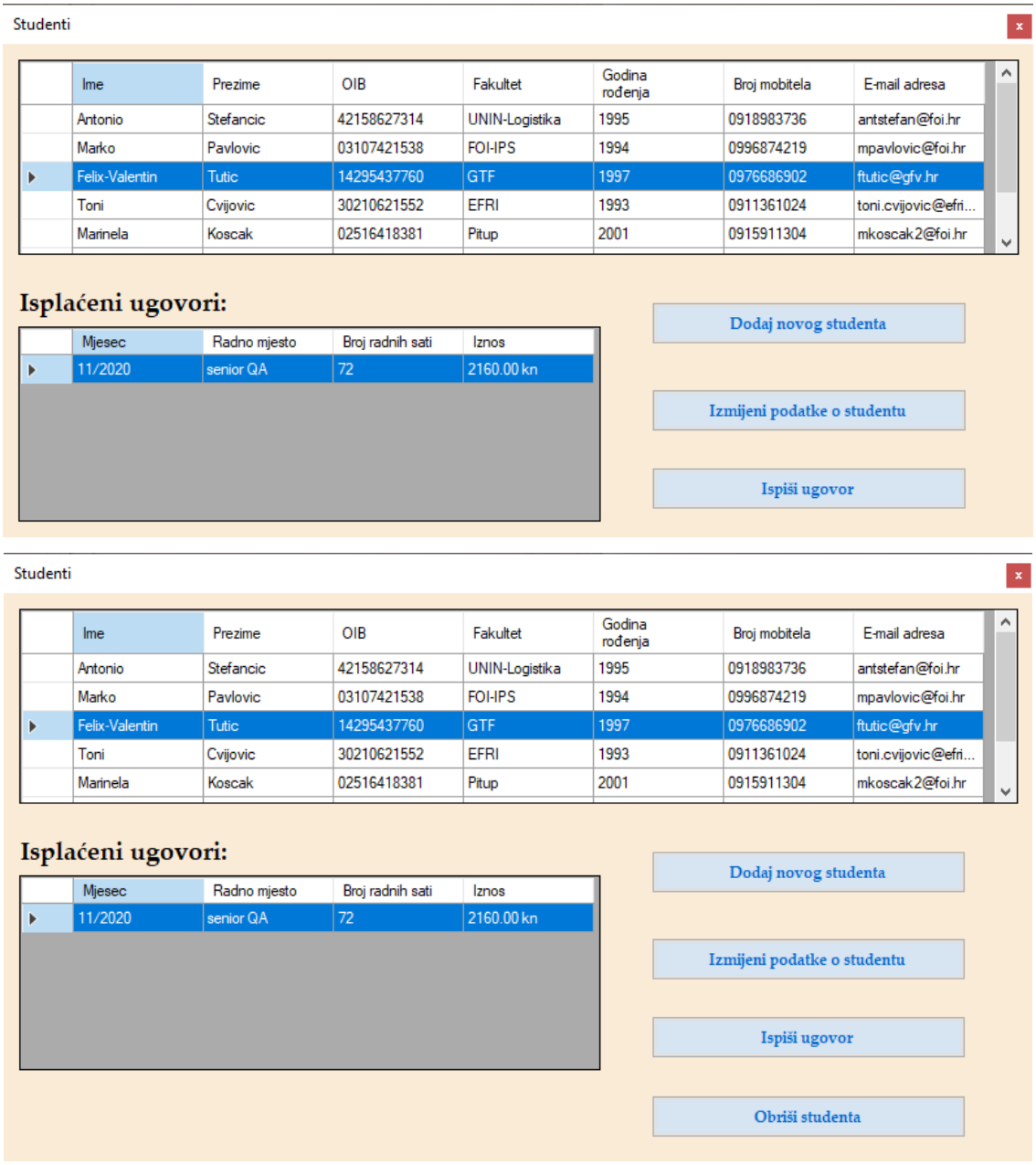
```
   Pocetna forma = new Pocetna(user);
```

gdje je user varijabla u koju se kod prijave prilikom dohvata iz baze učitava korisničko ime odabrane uloge. Varijabla korisnik iz forme Pocetna se na isti način dalje prosljeđuje kod poziva formi Studenti i Ugovori.

Način na koji se funkcionalnosti onemogućavaju određenim ulogama implementiran je na način da se jednostavno ne prikažu određene opcije. Sljedeći kôd poziva se u formi studenti, dio je metode za učitavanje, Studenti_Load:

```
if (korisnik != "direktor")
{
   Height = 418;
   btnObrisi.Hide();
}
```

Gumb btnObrisi služi za brisanje studenata, što je ovlast koja je dozvoljena isključivo direktoru. Iz tog razloga gumb je skriven za ostale uloge, a forma smanjena iz estetske prirode, kako ne bi ostalo prazno mjesto na mjestu guma. Slijedi primjer prikaza takve prijave:



Slika 34. Prikaz studenata kao direktor i kao zaposlenik (autorska izrada)

U gornjem dijelu slike prikazan je način prijave kao zaposlenik, dok je u donjem dijelu prijavljen direktor. Na isti način je uređena i metoda za pozivanje forme Ugovori:

```
if (korisnik == "zaposlenik")
{
   btnOtkaziUgovor.Hide();
   btnPodnesiUgovor.Hide();
   Height = 268;
}
```

Kako za podnošenje ili otkazivanje studentskog ugovora ovlast ima i projektni menadžer, opcija je skrivena samo od uloge zaposlenik.

Po dovršetku odvajanja funkcionalnosti uloga, aplikacija se može smatrati dovršenom te je spremna za izbacivanje gotove verzije. U nastavku je prikazan konačni plan projekta:

Tablica 4. Finalni vremenski plan projekta

Funkcionalnost	Datum početka	Datum završetka	Trajanje u danima
Izraditi bazu podataka.	7.2.2021.	8.2.2021.	2
Spojiti aplikaciju na bazu podataka.	9.2.2021.	9.2.2021.	0,5
Napraviti navigaciju među formama.	9.2.2021.	10.2.2021.	1,5
Unijeti novog ili izmijeniti podatke o postojećem studentu.	11.2.2021.	11.2.2021.	0,5
Unijeti novo ili izmijeniti podatke o postojećem radnom mjestu.	11.2.2021.	11.2.2021.	0,5
Učitati popis svih studenata i ugovora iz baze podataka.	14.2.2021.	14.2.2021.	1
Staviti ograničenje o jednom ugovoru mjesečno po studentu.	15.2.2021.	15.2.2021.	1
Izračunati iznos prilikom sklapanja ugovora.	16.2.2021.	16.2.2021.	0,5
Ispisati izvješće.	16.2.2021.	16.2.2021.	0,5
Poglede u bazi podataka zamijeniti tablicama.	17.2.2021.	17.2.2021.	0,5
Ažurirati dijagram klasa.	17.2.2021.	17.2.2021.	0,5
*Napraviti vremenski plan za implementaciju novog skupa funkcionalnosti.	18.2.2021.	18.2.2021.	1
Ažurirati bazu podataka.	21.2.2021.	21.2.2021.	0,5
Kreirati formu za prijavu.	21.2.2021.	22.2.2021.	1
O(ne)mogućiti korisničke funkcionalnosti.	22.2.2021.	22.2.2021.	0,5
Ukupno	**7.2.2021.**	**22.2.2021.**	**12**

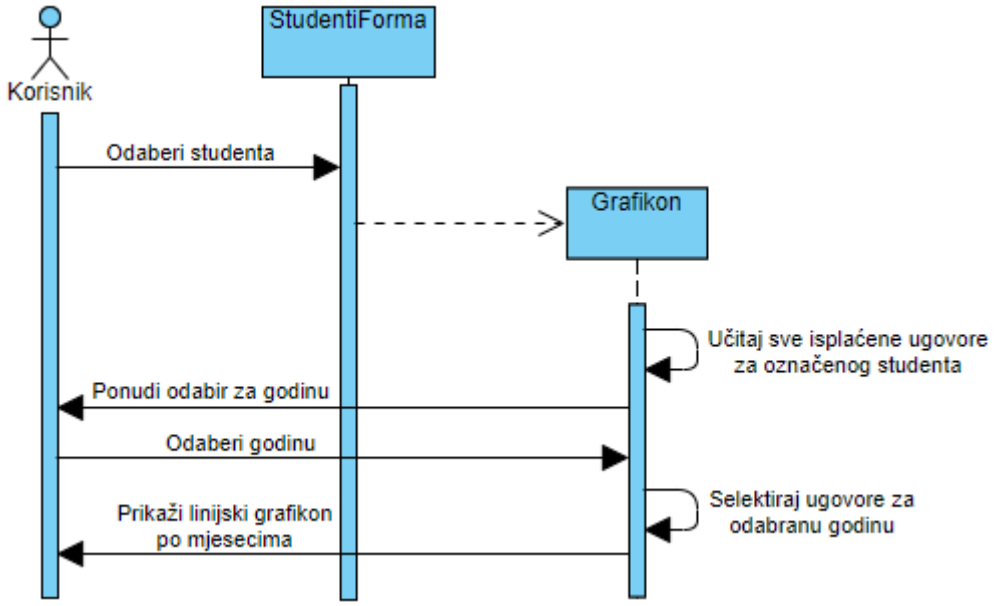
Izvor: autorska izrada

Za izradu projekta bilo je potrebno dvanaest dana, odnosno 48 radnih sati. Redak označen znakom * ne odnosi se na izradu funkcionalnosti, nego na planiranje izrade sljedeće iteracije, ali je stavljen u tablicu jer je proveden nakon inicijalnog dizajna i planiranja, gdje je kreirana originalna tablica.

Kako je aplikacija dovršena i sve funkcionalnosti za ovu verziju uspješno implementirane, smatra se da je projekt uspješno obavljen.

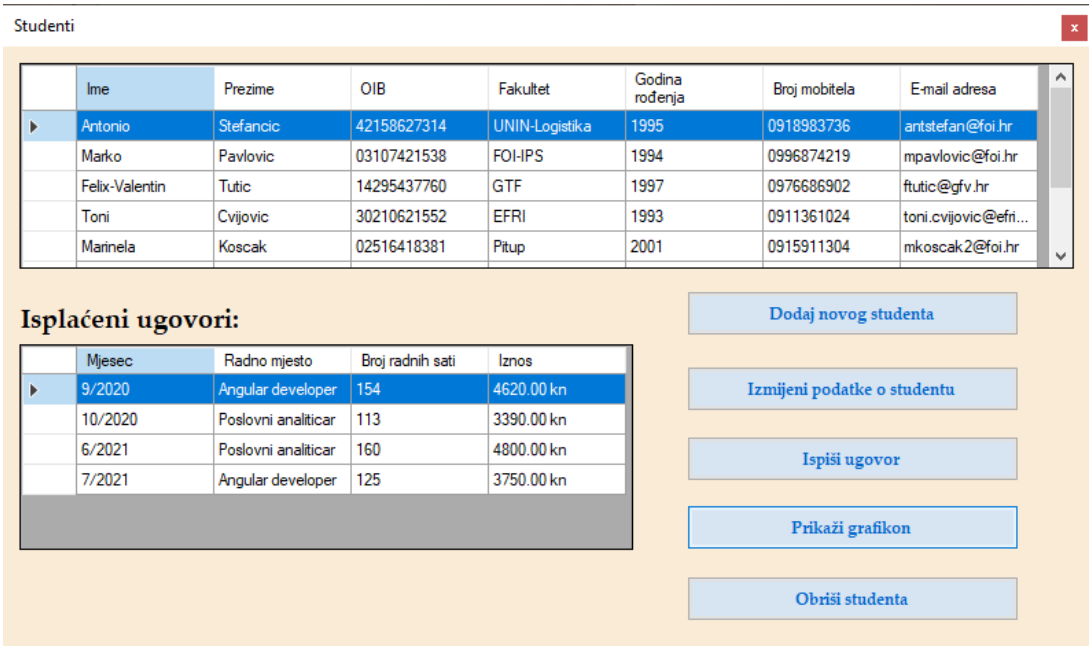
7.8. Ažuriranje programa

Nakon određenog vremena kako je aplikacija izbačena, nastala je potreba za implementacijom funkcionalnosti koja je u izradi prethodne verzije aplikacije označena slovom D – grafički prikazati broj radnih sati mjesečno kroz godinu. Budući da ova funkcionalnost ne zahtijeva izmjene u dosadašnjim funkcionalnostima ili promjene u bazi podataka, odmah se prelazi na fazu dizajniraj po funkcionalnosti, za što je napravljen dijagram slijeda:



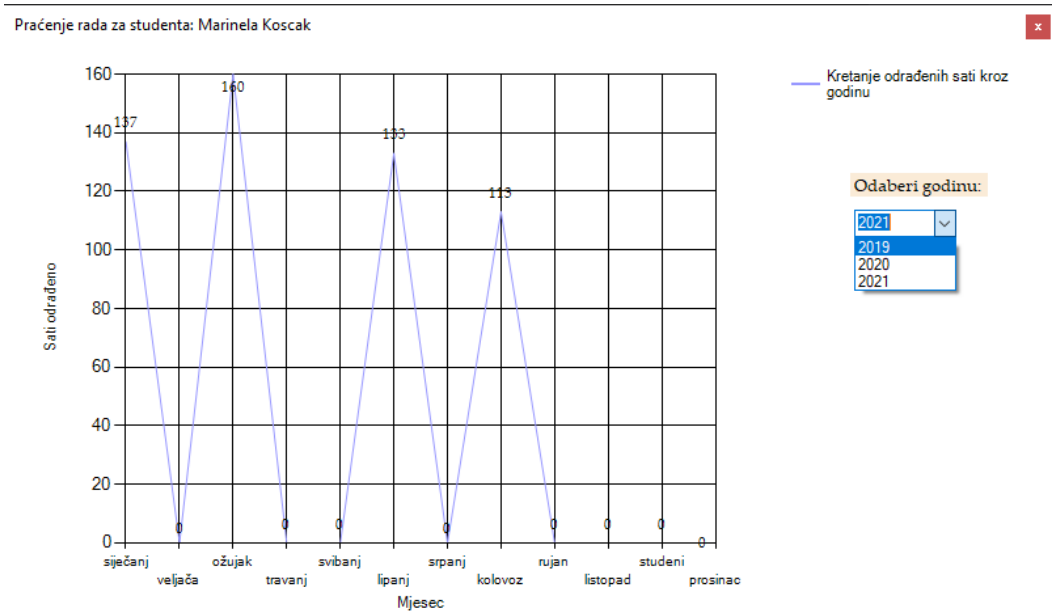
Slika 35. Dijagram slijeda za grafički ispis (autorska izrada)

Kako nije bilo zahtjeva za ovlasti grafičkog prikaza, za pokretanje dijagrama je na formu Studenti dodan novi gumb vidljiv svim korisničkim ulogama, a u novu formu se prosljeđuje selektirani student s forme Studenti. Na sljedećoj slici prikazan je novi izgled forme Studenti:



Slika 36. Dodan gumb za otvaranje grafikona (autorska izrada)

Grafikon se po otvaranju prikaza prikazuje za 2021. godinu, dok je iz ComboBoxa moguće odabrati dvije godine ranije:



Slika 37. Grafikon u pokrenutoj aplikaciji (autorska izrada)

Kako bi grafikon prilikom promjene vrijednosti u ComboBoxu obrisao stare vrijednost, potrebno je na ComboBox dodati događaj (eng. *event*) tipa SelectedValueChanged i u njega staviti sljedeće linije koda:

```
chart1.Series["Kretanje odrađenih sati kroz godinu"].Points.Clear();
ucitajGraf();
```

Prva linija briše stare vrijednosti iz serije podataka, dok druga iznova poziva metodu za učitavane vrijednosti. Kako je testiranje grafičkog prikaza prošlo uspješno, i ova verzija aplikacije može ići u izbacivanje (eng. *release*).

Aplikaciju je moguće preuzeti na poveznici navedenoj u popisu priloga na kraju rada. Testne lozinke za korisničke uloge navedene su u sljedećoj tablici:

Tablica 5. Korisničke uloge i lozinke

Korisnička uloga	**Lozinka za prijavu**
direktor	direktor
projektni menadžer	menadzer123
zaposlenik	lozinka

Izvor: autorska izrada

Osim izvornog koda aplikacije, s repozitorija je moguće preuzeti i bazu podataka. Prednost metode FDD ovdje se očitovala u ranom otkrivanju pogreške u ispisu otvorenih i isplaćenih ugovora, gdje je nepravilnost uočena na testiranju ubrzo nakon što je isprogramirana povezanost s bazom podataka, a podjelom zadataka aplikacije na funkcionalnosti i njihovim grupiranjem u skupove funkcionalnosti napravljen je logičan slijed implementacije koji stvara smislenu cjelinu.

8. Zaključak

Feature-driven development relativno je nova agilna metoda za razvoj softvera, stara tek dvadesetak godina. Metoda se temelji na funkcionalnostima, funkcijama koje koje klijentu predstavljaju vrijednost, na način da se kroz proces od pet koraka softver izrađuje funkcionalnost po funkcionalnost. Izrada jedne funkcionalnosti traje najviše dva tjedna.

Popis funkcionalnosti neformalno se sastavlja na samom početku projekta, u procesu modeliranja domene, dok se detaljan popis kreira tek nakon što je gotov model. Prilikom izrade softvera, funkcionalnosti se dodjeljuju vlasnicima, koji time dobivaju odgovornost dovršenja funkcionalnosti, te formiraju tim programera za njihovo izvršenje. Kako bi se dijelilo znanje među članovima tima, uloge nisu striktno definirane prema radnom mjestu koje član obavlja, a optimalno je za svaku novu funkcionalnost sastavljati timove nanovo. O redoslijedu implementacije funkcionalnosti odluku donosi projektni menadžer zajedno s glavnim programerima, a na odluku mogu utjecati i klijenti, u slučaju da već imaju isplaniran redoslijed implementacije.

Kvaliteta koda osigurava se inspekcijama, recenzijama i povratnim informacijama od strane klijenata te se na taj način osigurava da projekt ne ode u pogrešnom smjeru, što bi dovelo do propasti projekta, pa tako i softverskog rješenja koje se izrađuje. Upravo u tome leži najveća snaga ove metode.

Ono što bi skeptične menadžere i programere moglo privući ovoj metodi je činjenica da je razvijena usputno, na projektu razvoja softvera, a time i dokazana u praksi. Također je odlika i fleksibilnost, odnosno FDD se može provesti na ranije započetom projektu.

FDD je bio temelj brojnih studentskih projekata u kojima se kombinirao s nekim drugim metodama za razvoj softvera, između kojih je i udruživanje s ekstremnim programiranjem radi povećanja konkurentnosti na tržištu. Kritičan nedostatak FDD-a je nepripadajuća softverska podrška, odnosno nepostojanje odgovarajućeg alata za provođenje metode, što bi programere i analitičare koji za praćenje tijeka projekta vole koristiti gotove programe moglo odvući od metode.

Treba li koristiti ovu metodu ili ne, nema konkretnog odgovora. Ovisi o raznim čimbenicima: opsegu projekta, prethodnom iskustvu, pa i o načinu rada na koji su navikli sudionici u projektu. FDD je pogodan za projekte koji se za vrijeme trajanja neće proširivati za više od 10% opsega, budući da se takvo povećanje neće odraziti na vremenski plan projekta. Znatnija povećanja za sobom povlače izmjene koje traže intervenciju vrhovnog menadžmenta. Koraci metode lako se nauče, no zbog spiralnog modela, u praksi može doći do nejasnoća prilikom raspodjele posla, stoga je poželjno da bar dio razvojnog tima budu članovi s iskustvom

u razvoju po funkcionalnostima. Također, u FDD-u nije prisutno rigorozno testiranje kao kod ekstremnog programiranja, već je naglasak na dizajnu. Zbog toga bi se na projektima gdje je sigurnost ključan faktor, a pogreška u kodu nedopustiva, kao npr. bankarstvo ili avioindustrija, moglo okrenuti drugim metodama, dok je za većinu ostalih projekata metoda primjenjiva.

Sve u svemu, riječ je o zanimljivoj metodi koja se lako nauči, kreće linearno, a potom se prebacuje u iterativni način. Zbog svoje prilagodljivosti FDD je i pogodan temelj za studentski rad ili pak nekakav znanstveni projekt, gdje bi npr. izrada novog softvera za potporu metodi bila prilika za dobar početak znanstvene karijere. Pozitivno mišljenje o metodi potvrđeno je i izradom programskog rješenja njezinim provođenjem, gdje se redovitim izmjenama faza dizajna i izgradnje funkcionalnosti postigla dinamika kojom je implementacija provedena brzo, sukladno prethodno napravljenim UML dijagramima. Možda će netko preferirati neku drugu agilnu metodu, npr. ekstremno programiranje ili dinamični razvoj sustava, no svakako je preporuka isprobati ovu metodu u praksi te steći navike podjele posla na manje iteracije i provođenja čestih recenzija napravljenog dijela, koje su se u programskom dijelu pokazale veoma korisne.

Popis literature

Palmer, S. R. i Felsing, J. M. (2002). *A Practical Guide to Feature-Driven Development*. Upper Saddle River, NJ, SAD: Prentice-Hall

Hunt, J. (2006). *Agile Software Construction*. Chippenham, Engleska, Ujedinjeno Kraljevstvo: Springer

Coad, P., Lefebvre, E. i De Luca, J. (1999). *Java Modeling in Color with UML: Enterprise Components and Process*. Upper Saddle River, NJ, SAD: Pearson Ptr

Sommerville, I. (2011). *Software Engineering, 9th Edition*. Addison-Wesley, Boston, MA, SAD

Khramtchenko, S. (2005). *A Project Management Application for Feature Driven Development*. Cambridge, MA, SAD: Harvard University

Cause, G. (2004). *Delivering Real Business Value using FDD*. IT Project Services Pty. Ltd. Pristupano 6.9.2021. na https://www.methodsandtools.com/archive/archive.php?id=19

Khramtchenko, S. (2006). Guide for Chief Programmers. SourceForge. Preuzeto 22.8. 2021. s http://fddpma.sourceforge.net/help/fddpma_cp_guide.pdf

Roock, S. (2007). *Jeff De Luca on Feature Driven Development – Interview*. Preuzeto 13.6.2021. s https://www.it-agile.de/fileadmin/docs/FDD-Interview_en_final.pdf

Larman, C. i Vodde, B. (2008). *Scaling Lean & Agile Development: Thinking and Organizational Tools for Large-Scale Scrum, 1st Edition*. Preuzeto 4.8.2021. s https://www.craiglarman.com/content/feature-teams/feature-teams.htm

Karanth, D. (2016). *Collective Code Ownership in Agile Teams*. Preuzeto 31.7.2021. s https://dzone.com/articles/collective-code-ownership-in-agile-teams

Chursin, O. (2017). *A Brief Introduction to Domain Modeling*. Preuzeto 31.7.2021. s https://olegchursin.medium.com/a-brief-introduction-to-domain-modeling-862a30b38353

Schults, C. (2021). *What Is Collective Code Ownership? An Introduction*. Preuzeto 4.8.2021. s https://www.coscreen.co/blog/what-is-collective-code-ownership/

Thakur, S., Singh, H. (2014). *FDRD: Feature Driven Reuse Development Process Model*. Department of Computer science and Engineering, Lovely Professional University, Jalandhar, Indija

Rychlý, M. i Tichá , P. (2007). *A Tool for Supporting Feature-Driven Development*. Preuzeto 12.7.2020. s https://link.springer.com/chapter/10.1007/978-3-540-85279-7_16

Microsoft (2021). Cognizant Feature Driven Development. Pristupano 20.8.2021. na https://marketplace.visualstudio.com/items?itemName=Santosh4.CognizantFeatureDrivenDevelopment

SourceForge (bez dat.). *FDD Project Management Application – User Guide*. Pristupano 26.8.2021. na http://fddpma.sourceforge.net/tutorials.html

Shindhe, S. (2007). *Feature Driven Development using VSTS and Cognizant FDD templates*. Cognizant Technology Solutions. Pristupano 20.8.2021. na https://www.slideserve.com/keola/feature-driven-development-using-vsts-and-cognizant-fdd-templates

Doshi, V. P. i Patil, V. (2016). *Competitor Driven Development – Hybrid Of Extreme Programming and Feature Driven Reuse Development*. University of Mumbai, India. Preuzeto 14.12.2020. s https://ieeexplore.ieee.org/document/7602985

Mahdavi-Hezave, R. i Ramsin, R. (2015). *FDMD: Feature-Driven Methodology Development*. Preuzeto 9.12.2020. s https://www.scitepress.org/papers/2015/53842/53842.pdf

Chowdury, A. F. i Nazmul Huda, M. (2011). *Comparison between Adaptive Software Development and Feature Driven Development*. Preuzeto 13.3.2021. s https://ieeexplore.ieee.org/document/6181977

Literatura za projekt

Mijač, M., Švogor, I. i Tomaš, B. (2016). *Odabrana poglavlja programskog inženjerstva*, verzija 1.2. Programsko inženjerstvo [Moodle]. Sveučilište u Zagrebu, Fakultet organizacije i informatike, Varaždin

Popis slika

Popis tablica

Prilozi

GitHub repozitorij koji sadrži programski kôd i bazu podataka:
https://github.com/CROmarcos/Track-Your-Crew