



Miloš Antonijević

UPRAVLJANJE SOFTVERSKIM PROJEKTOM



Beograd, 2024.



UNIVERZITET SINGIDUNUM

Miloš Antonijević

UPRAVLJANJE SOFTVERSKIM PROJEKTOM

PRVO IZDANJE

Beograd, 2024.

UPRAVLJANJE SOFTVERSKIM PROJEKTOM

Autor:

dr Miloš Antonijević

Recenzenti:

dr Nebojša Bačanin-Džakula

dr Boško Nikolić

Izdavač:

Univerzitet Singidunum

Danijelova 32

www.singidunum.ac.rs

Za izdavača:

dr Milovan Stanišić

Priprema za štampu:

Miloš Višnjić

Dizajn korica:

Aleksandar Mihajlović, MA

Godina izdanja:

2024.

Tiraž:

500

Štampa:

Birograf, Beograd

ISBN: 978-86-7912-824-9

Copyright:

© 2024. Univerzitet Singidunum

Izdavač zadržava sva prava.

Reprodukcija pojedinih delova ili celine ove publikacije nije dozvoljena.

Sadržaj



1. Osnovni pojmovi i koncepti razvoja softverskog projekta	3
1.1. Definicija i osobine projekta	3
1.2. Upravljanje projektima	6
1.3. Programi i Portfoliji	7
1.4. Zainteresovane strane	8
1.5. Gvozdeni trougao (Iron triangle)	11
1.6. Vrednost projekta	15
1.7. Trajanje aktivnosti	16
1.8. Koštanje projekta	20
1.9. Isplativost projekta	22
1.10. Rizici u razvoju softverskog projekta	24
1.11. FUS (Functionality, Usability, Security) trougao	28
1.12. Klasifikacija softverskih projekata	30
2. Analiza i specifikacija zahteva	35
2.1. Vrste zahteva	35
2.2. Kreiranje i upravljanje zahtevima	39
2.2.1. Prikupljanje zahteva	40
2.2.2. Analiza zahteva	44
2.2.3. Dokumentovanje zahteva	45
2.2.4. Validacija zahteva	53
2.2.4.1. Rešavanje konflikata	54
3. Modeli razvoja softverskog projekta	59
3.1. TPM, APM, xPM, Mpx	61
3.1.1. Tradicionalni pristup upravljanja projektom (TPM)	62
3.1.2. Agilni pristup upravljanja projektom (APM)	63
3.1.2.1. Lean pristup	64
3.1.2.2. SCRUM	68
3.1.2.2. Kanban metod	74
3.1.3. Ekstremni pristup upravljanja projektom (xPM)	79
3.1.4. Emurci (Emertxe) pristup upravljanja projektom (MPx)	80

3.2. Razvojni tim softverskog projekta	81
3.2.1. Članovi projektnog tima	82
3.2.2. Izbor članova projektnog tima	84
4. Softverski alati za upravljanje projektom	91
4.1. Mrežni dijagram	92
4.2. Gantov dijagram (Gantogram)	94
4.3. GIT	96
4.3.1. Instalacija Git-a	98
4.3.2. Osnovni pojmovi i koncepti	99
4.3.3. Dodavanje izmena u pripremni prostor	101
4.3.4. Komitovanje (potvrđivanje) izmena (u repozitorijum)	102
4.3.5. Upravljanje granama repozitorijuma	103
4.3.6. Vraćanje prethodnih verzija i putovanje kroz vreme	110
4.3.7. Kolaboracija (saradnja) na Git projektima	115
5. Testiranje softverskog projekta	121
5.1. Verifikacija i validacija	122
5.2. Vrste testiranja	123
5.3. PHPUNIT TEST	124
5.4. BEHAT	130
6. Izrada softverske dokumentacije	141
6.1. phpDocumentor	146
7. Isporuka i održavanje	155
7.1. Kontinuirana integracija i Kontinuirana isporuka (CI/CD)	159
Literatura	164

OSNOVNI POJMOVI I KONCEPTI RAZVOJA SOFTVERSKOG PROJEKTA



1. OSNOVNI POJMOVI I KONCEPTI RAZVOJA SOFTVERSKOG PROJEKTA

1.1. DEFINICIJA I OSOBINE PROJEKTA

Institut za menadžment projekata (Project Management Institute) definiše projekat kao **privremeni poduhvat koji se preduzima da se stvori jedinstven proizvod ili usluga**¹. Pomenuta definicija navodi nekoliko važnih karakteristika projekta, od kojih su neke eksplicitno pomenute, a neke se mogu izvesti.

Projekte karakteriše njihova **privremena** priroda, što znači da imaju jasno definisan početak i kraj. Projekti su konačni, odnosno, imaju određen datum završetka koji se dostiže ili nakon uspešnog ostvarenja ciljeva projekta, odluke o prekidu zbog nedostižnih ciljeva, donošenja odluke da je projekat postao nepotreban, ili ukoliko klijent izrazi želju da ga okonča. Ovaj datum predstavlja ograničenje koje najčešće nameće klijent i koji nije pod kontrolom tima koji razvija projekat. Osobina privremenosti projekta kao posledicu ima postojanje nekoliko faza u razvoju projekta:

1. Inicijalnu fazu u kojoj se izrađuje okvirna projektna infrastruktura i definišu opšti ciljevi projekta.
2. Fazu planiranja, tokom koje se ciljevi projekta preciziraju, identifikuju i raspoređuju aktivnosti, i pripremaju operacije koje predstavljaju podršku razvoju projekta.
3. Fazu izvršavanja, koja predstavlja praktično izvođenje aktivnosti razvoja projekta.
4. Fazu praćenja, koja se odvija paralelno sa fazom izvršavanja i predstavlja posmatranje i merenje napretka i beleženje odstupanja realnog stanja u kome se projekat nalazi u datom trenutku u odnosu na definisani plan razvoja.
5. Završnu fazu, kada se isporučuju rezultati projekta (proizvod) i projekat se zatvara.

1 Project Management Institute, A guide to the project management body of knowledge (PMBOK guide). 2017

Svaki projekat stvara **jedinstven** proizvod, uslugu ili rezultat. Iako neke aktivnosti mogu sadržati elemente koji se ponavljaju, ovo ponavljanje ne menja jedinstvene karakteristike projektnog rada kao celine, koji sam po sebi predstavlja određenu novinu. Ova osobina sugerise da je svaki poduhvat uvek povezan sa nekim rizikom, bilo da je reč o prilikama ili pretnjama. **Rizici** proističu iz jedinstvenih karakteristika rezultata projekta, koji su ponekad nejasni ili nepoznati pre nego što projekat započne. Dodatna ograničenja projekta, kao što su klijentsko nametanje nerazumno kratkog roka za završetak projekta ili veoma visoki kriterijumi kvaliteta, stvaraju dodatne opasnosti.

Četvrta karakteristika koja proizilazi iz definicije projekta jeste da projekat **isporučuje rezultat** u vidu proizvoda ili usluge koji su opipljivi i njihove karakteristike merljive. Na taj način, projekat može biti osmišljen i organizovan počevši od opisa rezultata (proizvoda ili usluge) i karakteristika rezultata koje treba da isporuči. Opis projektnih rezultata takođe definiše kriterijume za završetak projekta, odnosno, projekat se završava kada se rezultati isporuče prema navedenim specifikacijama.

Ljudi često nazivaju svaki posao koji moraju obaviti „projektom“. Međutim, projekti zapravo imaju vrlo specifičnu definiciju. Ako skup zadataka ili poslova koji treba obaviti ne ispunjava strogu definiciju, tada se ne može nazvati projektom. Pored navedene najjednostavnije definicije projekta od strane Instituta za menadžment projekta, projekat se drugačije može opisati kao **niz jedinstvenih, složenih i povezanih aktivnosti koje imaju jedan cilj ili svrhu i koje moraju biti završene do određenog vremena, unutar budžeta i prema specifikacijama**².

Ovakva definicija projekta podrazumeva da se projekat sastoji iz niza aktivnosti koje moraju biti završene prema određenom redosledu. Da bi se odredio redosled aktivnosti, potrebno je razmišljati o ulazima i izlazima aktivnosti, na način da izlaz jedne aktivnosti ili skupa aktivnosti postaje ulaz u drugu aktivnost ili skup aktivnosti. Ova povezanost podrazumeva da postoji logički odnos između parova aktivnosti i postojanje redosleda kojim se one moraju završiti. U praksi može postojati skup nepovezanih aktivnosti koje sve moraju biti završene kako bi se kompletirao neki posao, a koje ne predstavljaju projekat. Na primer, razmotrimo posao uređenja knjiga u biblioteci. Aktivnosti koje mogu biti sadržane u okviru ovog posla mogu uključivati: pravljenje kataloga knjiga, označavanje knjiga identifikacionim nalepnicama i stavljanje knjiga na police. Sa nekim izuzecima, ove aktivnosti mogu biti završne bilo kojim redosledom.

2 R. K. Wysocki, Effective Project Management: Traditional, Agile, Extreme (Seventh Edition). Wiley. 2013.

Tako, označavanje knjiga može biti završeno pre ili posle pravljenja kataloga, a ređanje knjiga na police je povezano sa označavanjem, ali ne zavisi od katalogizacije. Iako je potrebno izvršiti sve navedene aktivnosti da bi se uređenje knjiga smatralo završenim, pa ovaj proces predstavlja svojevrsan skup aktivnosti, one ne predstavljaju projekat jer im nedostaje kompleksna povezanost i međuzavisnost. Ovi zadaci ne moraju biti završeni određenim redosledom već se mogu obaviti u zavisnosti od trenutno raspoloživih resursa ili prema prioritetima.

Projekti moraju imati jedan jedini cilj. U slučaju da je projekat suviše kompleksan, neophodno ga je podeliti u više manjih projekata kako bi se zadovoljila ova osobina. Ovakva veštačka podela složenog projekta na podprojekte često pojednostavljuje raspoređivanje resursa i upravljanje razvojem. Projekat je ograničen resursima. Na raspolaganju je ograničeno vreme za izgradnju rezultata projekta. Takođe su ograničeni i drugi resursi projekta, kao što su budžet i tim. Ljudski resursi su u najvećem broju slučajeva fiksni u okviru jednog projekta. Važna posledica je da menadžer projekta i tim moraju pronaći ostvarivo rešenje, poštujući sva ograničenja projekta. Na taj način, rezultat projekta retko predstavlja najbolje moguće rešenje, već pre najbolje rešenje s obzirom na ograničenja.

Projekat mora biti realizovan u skladu sa utvrđenom i odobrenom specifikacijom. Klijent ima određene zahteve za funkcionalnost i kvalitet projekta. Specifikacija može varirati iz različitih razloga, čak i ako je projektni menadžer smatra fiksnom. Na primer, zahtevi možda nisu bili potpuno definisani od strane klijenta kada je projekat započeo. Dodatni problem za projektnog menadžera je činjenica da je nerealno očekivati da specifikacija ostane nepromenjena tokom trajanja projekta. Najčešći uzrok neuspeha projekata je upravo neispunjavanje definisane specifikacije.

1.2. UPRAVLJANJE PROJEKTIMA

Institut za menadžment projekata formalno definiše upravljanje projektima kao **primenu znanja, veština, alata i tehnika na aktivnosti projekta kako bi se ispunili zahtevi projekta**. Pitanje kojim započinje svaki proces upravljanja projektom jeste koja poslovna situacija se rešava konkretnim projektom? Poslovna situacija može predstavljati neiskorišćenu (otvorenu) priliku ili problem koji treba rešiti. Ako je u pitanju problem, rešenje može biti jasno definisano i njegova implementacija će biti prilično jednostavna. Ako rešenje nije potpuno poznato, strategija upravljanja projektom mora iterativno uključivati učenje i pronalaženje odgovora. Uprkos najboljim naporima klijenta i projektnog tima, rezultati projekta u ovom drugom tipu projekta nisu jasno definisani i možda neće biti pronađeni, što ga čini rizičnijim od prvog. Nakon što se odredi problem ili prilika koji se tiču datog projekta, neophodno je definisati opšte korake koji vode do rešenja. Ovo se najčešće obavlja kroz izradu dokumenta koji sadrži cilj i izjavu o namerama koji predstavlja osnovu za sastavljanje detaljnijeg plana u vidu projektne specifikacije. U zavisnosti od tipa projekta, plan može biti potpuno dokumentovan od samog početka ili se može razvijati iterativno, u toku trajanja projekta, ukoliko rešenje kome se teži nije u potpunosti očigledno na početku projekta.

Svaki projekat ima za cilj da doprinese vrednosti organizacije. Ova činjenica omogućava da se nivo uspešnosti projekta može meriti na kvantitativan način koji ne ostavlja sumnju da li je postignut očekivan poslovni rezultat. Drugi aspekt provere koja se obavlja po završetku projekta jeste učinak projektnog tima. U toku razvoja projekta, projekti tim prati neku metodologiju razvoja, po osnovu koje je moguće utvrditi stepen realizacije projektnih zadataka u skladu sa postavljenom metodologijom. U vezi sa ovim, potrebno je utvrditi u kojoj meri je projektni menadžer odabrao pravi pristup (model) za razvoj konkretnog projekta. Na kraju, utvrđuje se koje su lekcije naučene u toku procesa razvoja, a da se mogu primeniti na buduće projekte.

Upravljanje projektom obično obuhvata, ali se ne ograničava na:

- Identifikaciju zahteva;
- Upravljanje različitim potrebama, rizicima i očekivanjima zainteresovanih strana u planiranju i izvođenju projekta;
- Uspostavljanje, održavanje i sprovođenje komunikacije među zainteresovanim stranama koja je aktivna i efikasna;

- Balansiranje između suprotstavljenih ograničenja projekta, koja uključuju:
 - Opseg
 - Kvalitet
 - Vreme
 - Budžet
 - Resurse
 - Rizike.

1.3. PROGRAMI I PORTFOLIJI

Program je u osnovi skup povezanih projekata koji se sprovode kako bi se postigli određeni ciljevi organizacije ili pojedinca. Projekti unutar programa mogu biti međusobno povezani i mogu zahtevati određeni redosled izvršavanja kako bi se postigli željeni rezultati. Budući da programi obuhvataju više projekata, obim njihovog delovanja je obično veći i složeniji nego obim jednog pojedinačnog projekta. Jedna od ključnih karakteristika programa je da mogu imati mnogo ciljeva, koji mogu biti međusobno povezani ili se mogu razlikovati po prirodi i opsegu. Ciljevi programa obično odražavaju širu strategiju organizacije. Na primer, program digitalne transformacije poslovanja restorana može da uključuje nekoliko projekata od kojih svaki ima jedinstveni cilj: razvoj mobilne aplikacije za rezervisanje mesta i naručivanje, izbacivanje programa lojalnosti za najvernije kupce, izrada softvera za upravljanje poslovima u kuhinji i treniranje osoblja za korišćenje novih digitalnih alata. Osim činjenice da svi doprinose poboljšanju zadovoljstva korisnika i efikasnosti rada kroz digitalizaciju poslovanja, projekti su nezavisni jedan od drugog, imaju svoje zasebne ciljeve i zajedno definišu program. Sa druge strane, ponekad upravljanje programima obuhvata koordinaciju i upravljanje svim projektima unutar programa kako bi se osiguralo da se postižu ciljevi programa u skladu sa zadatim vremenom, budžetom i resursima. U tom slučaju se zahteva integrisani pristup upravljanju projektima, koji može uključivati deljenje resursa, rizika i informacija između projekata unutar programa.

Jednostavna definicija **portfolija** projekata je da je to kolekcija projekata koji međusobno dele neku zajedničku vezu. Na nivou preduzeća, veza može biti samo činjenica da svi projekti pripadaju istoj kompaniji. Druge zajedničke veze obuhvataju:

- Projekti mogu poticati iz iste poslovne jedinice
- Projekti mogu svi biti projekti istog tipa (razvoj novih proizvoda, istraživačko-razvojni projekti...)
- Projekti mogu biti istog stepena rizika.
- Projekti mogu tražiti finansijsku podršku iz istog budžeta.

Svakom portfoliju će se nezavisno dodeliti resursi (vreme, novac i osoblje) kako bi se sproveli svi projekti koji su odobreni. Jedna organizacija može imati više portfolija kojima se dodeljuju resursi prema prioritetu. Odnos između portfolija, programa i projekata je takav da se portfolio odnosi na skup projekata, programa, podportfolija i operacija kojima se upravlja kao grupom kako bi se postigli strateški ciljevi. Programi su grupisani unutar portfolija i sastoje se od podprograma, projekata ili drugih aktivnosti koji se razvijaju koordinisano u podršci portfoliju.

1.4. ZAJINTERESOVANE STRANE

Zainteresovane strane u projektu nisu ograničene samo na tim koji radi na projektu, već uključuju sve pojedince ili grupe koji imaju interes ili mogu biti pogođene projektom. Identifikacija ovih zainteresovanih strana, razumevanje njihovih očekivanja i zahteva, kao i upravljanje njihovim uticajem su ključni aspekti upravljanja projektima. Na primer, u projektu razvoja novog tržnog centra u centru grada, postoje različite zainteresovane strane sa različitim interesima i uticajima. Među pozitivnim zainteresovanim stranama su građevinski investitori, koji vide priliku za profit kroz izgradnju i zakup komercijalnih prostora unutar tržnog centra. Lokalni državni zvaničnici takođe mogu pozitivno gledati na projekat, jer bi mogao doneti ekonomski rast, mogućnosti za zapošljavanje i povećanje poreskih prihoda. Takođe, potencijalni zakupci koji žele da iznajme prostor u tržnom centru mogu videti prednosti projekta, kao što su veći broj prolaznika i veća vidljivost za njihove poslove.

Sa druge strane, postoje negativne zainteresovane strane koje bi mogle biti negativno pogođene razvojem tržnog centra. Lokalni mali biznisi mogu doživeti tržni centar kao pretnju, jer bi mogao doneti povećanu konkurenciju i potencijalno dovesti do smanjenja prodaje ili čak njihovog zatvaranja. Stanovnici koji žive u blizini takođe mogu imati svoje brige, kao što su povećan saobraćaj, buka i promene u okolini. Grupacije za zaštitu životne sredine mogu biti još jedna negativna zainteresovana strana, sa zabrinutošću u vezi sa ekološkim uticajem izgradnje, uključujući uništavanje staništa i povećanu zagađenost.

Upravljanje ovim zainteresovanim stranama podrazumeva angažovanje sa svakom od ovih grupa kako bi se sagledale njihove brige i osiguralo da prednosti projekta nadmaše negativne uticaje. To može uključivati sprovođenje mera kao što su planovi upravljanja saobraćajem i procene uticaja na životnu sredinu kako bi se ublažili negativni efekti projekta. Aktivnim upravljanjem odnosima sa zainteresovanim stranama, menadžeri projekta mogu pomoći u osiguravanju uspešnog planiranja i realizacije projekta.

Neki od primera zainteresovanih strana su:

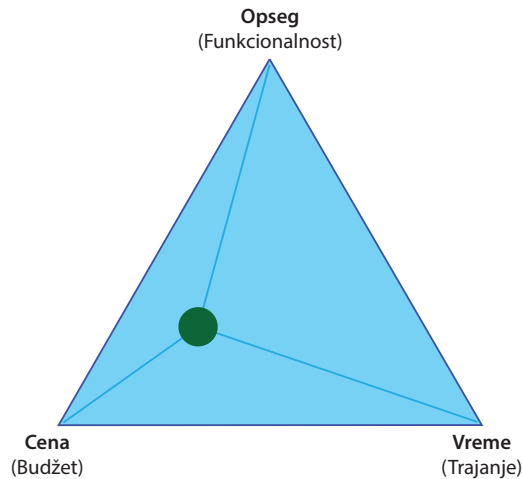
- Sponzor, odnosno organizacija ili pojedinac koji nose ključnu ulogu u upravljanju projektima kroz pružanje potrebnih resursa i podrške projektu i koji nose krajnju odgovornost za njegov uspeh. Od početka projekta do njegovog zatvaranja, sponzor igra ključnu ulogu u promovisanju i zagovaranju projekta. Može biti uključen u različite aktivnosti projekta, kao što su odobravanje promena u opsegu, praćenje stepena realizacije na kraju faza i donošenje odluka o daljem napretku kada su rizici posebno visoki. Osim toga, sponzor može učestovati u procesu rešavanja konflikta koji su van kontrole menadžera projekta ili rešavanju problema koji se tiču ograničenja u resursima.
- Klijenti i korisnici igraju različite ali povezane uloge u projektu. Klijenti su osobe ili organizacije odgovorne za odobravanje i upravljanje krajnjim rezultatom projekta, bilo da je reč o proizvodu ili usluzi. Obično su oni ti koji u najvećoj meri isporučuju zahteve projekta i određuju da li je projekat ispunio svoje ciljeve. Klijenti mogu biti interni ili eksterni u odnosu na organizaciju koja sprovodi projekat, i može postojati više nivoa klijenata u zavisnosti od obima i složenosti projekta. S druge strane, korisnici su osobe ili organizacije koje će direktno koristiti krajnji rezultat projekta nakon što bude isporučen. Korisnici mogu takođe pružati važne informacije tokom razvoja projekta kako bi se osiguralo da se uzmu u obzir njihove potrebe i očekivanja.

Odnos između klijenata i korisnika može varirati u zavisnosti od projekta i industrije. U nekim slučajevima, korisnici i klijenti mogu biti sinonimi, posebno u projektima gde krajnji rezultat koristi ista grupa koja ga je odobrila. Međutim, u drugim slučajevima može postojati razlika. Na primer, u softverskom razvojnom projektu, klijent može biti kompanija koja kupuje softver, dok su korisnici zaposleni koji će zapravo koristiti softver u svakodnevnom radu. Menadžeri projekata moraju efikasno komunicirati sa obe grupe kako bi se osiguralo da projekat donese vrednost klijentu, istovremeno ispunjavajući potrebe krajnjih korisnika.

- Prodavci, odnosno, dobavljači, snabdevači ili izvođači, su eksterne kompanije koje ulaze u ugovorni sporazum kako bi obezbedile komponente ili usluge potrebne za projekat.
- Poslovni partneri koji predstavljaju eksterne organizacije koje pružaju specijalizovano znanje ili obavljaju određenu ulogu kao što je instalacija, prilagođavanje, obuka ili podrška.
- Organizacione grupe, odnosno, interne zainteresovane strane unutar organizacije koje su pogođene aktivnostima projektnog tima. Ove grupe obuhvataju različite poslovne elemente koji mogu uticati na projekat, kao što su marketing i prodaja, ljudski resursi, pravni poslovi, finansije, operacije, proizvodnja i korisnička podrška. One kreiraju poslovno okruženje u kojem se projekti izvršavaju i kao rezultat toga pogođene su rezultatima projekta. Na primer, tim za marketing i prodaju može biti prinuđen da prilagodi svoje strategije na osnovu novog proizvoda razvijenog kroz projekat ili se odeljenje ljudskih resursa može suočiti sa implementacijom novih programa obuke za zaposlene pogođene promenama u projektu. Zbog njihovog učešća, obično postoji značajna interakcija između ovih organizacionih grupa i projektnog tima. Ova saradnja je ključna za postizanje ciljeva projekta i obezbeđivanje da projekat bude u skladu sa širim ciljevima organizacije. Osim toga, organizacione grupe pomažu u olakšavanju glatke tranzicije rezultata projekta u poslovanje organizacije čime osiguravaju ostvarivanje vrednosti koje projekat donosi.
- Ostali zainteresovani subjekti, poput finansijskih institucija, vladinih regulatornih tela, stručnjaka za određene oblasti, konsultanata i drugih.

1.5. GVOZDENI TROUGAO (IRON TRIANGLE)

„Brzo, kvalitetno, jeftino, molim odaberite dva“. Ovu rečenicu često možemo čuti u praksi od strane iskusnih menadžera projekta u početnoj fazi pregovora i definisanja projekta. Opseg, vreme i novac su tri ključna aspekta u upravljanju projektima, i postizanje suptilne ravnoteže između njih je neophodno za uspeh projekta. Ovaj međusobno povezani trio komponenata često se naziva „Gvozdeni trougao“ (Iron triangle). Da bi osigurali ostvarenje ciljeva projekta uz održavanje harmonične ravnoteže, projektni menadžeri su odgovorni za balansiranje ovih komponenti. Ove tri promenljive čine strane trougla i predstavljaju međusobno zavisni skup. Ako se bilo koja od njih promeni, barem jedna druga promenljiva mora takođe da se promeni kako bi se obnovila ravnoteža u projektu.



Pored navedenih osnovnih aspekata u procesu upravljanja projektom, postoje dodatne komponente koje utiču na razvoj projekta i njegove rezultate, a koje su međusobno zavisne. Najčešće se kao ograničenja projekta navode sledeće karakteristike:

- Opseg
- Kvalitet
- Cena
- Vreme
- Resursi
- Rizik

Navedeni ograničavajući faktori čini međusobno zavisni skup - promena jednog ograničenja može zahtevati promenu jednog ili više drugih ograničenja kako bi se obnovila ravnoteža projekta. U ovom kontekstu, skup od pet parametara (izuzev rizika) čini sistem koji mora ostati u ravnoteži kako bi projekat bio uravnotežen. **Opseg** je iskaz koji definiše granice projekta. U okviru njega se definišu funkcionalnosti koje će činiti sastavni deo finalnog proizvoda. U kontekstu informacionih sistema, opseg je osnova za izradu funkcionalne specifikacije softvera. U toku rada na projektu opseg projekta se može promeniti. Detektovanje potrebe za promenom i prilagođavanje plana projekta predstavljaju jedan od najvećih izazova za menadžera projekta. Kada se govori o **kvalitetu**, obično se pravi razlika između dva tipa:

- Kvalitet *proizvoda*, koji predstavlja kvalitet isporučivog rezultata projekta.
- Kvalitet *procesa* upravljanja projektom koji se fokusira na kvalitet samih aktivnosti koje čine sastavni deo razvojnog ciklusa projekta. Ovaj aspekt kvaliteta se odnosi na efikasnost i efektivnost praksi upravljanja projektima u cilju osiguranja uspešnog završetka projekta. Implementacija snažnog programa upravljanja kvalitetom koji prati projektne aktivnosti je ključna jer ne samo da povećava zadovoljstvo klijenata obezbeđujući visokokvalitetne rezultate, već i optimizuje korišćenje resursa organizacije. Smanjenjem gubitaka i minimiziranjem potrebe za revizijama, organizacije mogu postići veću efikasnost i isplativost. Ulaganje u upravljanje kvalitetom značajno povećava verovatnoću uspeha projekta i zadovoljstvo klijenata, što ga čini suštinskom komponentom svake strategije upravljanja projektom.

Troškovi u kontekstu projektnih ograničenja označavaju budžet koji je uspostavljen za projekat. To je posebno važno za projekte koji stvaraju isporučive proizvode koji se prodaju komercijalno ili eksternom kupcu. U toku ciklusa upravljanja projektom pitanje troškova se razmatra kontinuirano. Prvo razmatranje se dešava u ranom i neformalnom stadijumu života projekta. Klijent može ponuditi sumu novca koji je spreman da potroši, što može biti prilično blizu stvarnog troška projekta ili znatno od njega odstupati. U drugim slučajevima, izvođač projekta priprema predlog troškova u odnosu na konkretan projekat. Taj predlog uključuje procenu i ponudu ukupnog troška izrade projekta. Ovo omogućava klijentu da donese odluku o nastavku ili odustajanju od projekta ili angažovanje druge organizacije kao izvođača na osnovu bolje ponude.

U praksi je najčešća situacija da klijent definiše **vremenski okvir** ili rok do kojeg projekat mora biti završen. Do određenog stepena, troškovi i vreme su obrnuto proporcionalni. Vreme potrebno za završetak projekta može biti smanjeno, ali to dovodi do povećanja troškova. Cilj menadžera projekta je da buduće vreme koje je dodeljeno projektu koristi na najefikasniji i najproduktivniji način. U nekim situacijama menadžer projekta može biti osoba koja će u inicijalnoj fazi predložiti vremenski okvir neophodan da se projekat završi u skladu sa dogovorenim ograničenjima (budžet, veličina i tip projekta, resursi itd.).

Resursi u upravljanju projektima odnose se na sredstva potrebna za završetak projekta, što uključuje ljude, opremu, fizičke objekte i inventar. Njihova dostupnost je često ograničena i u nekim slučajevima je potrebno iznajmiti ih od spoljnih izvora. Resursi se mogu kategorisati u dva tipa: fiksni resursi, koji ostaju konstantni, i varijabilni resursi, koji se mogu menjati, ali obično samo na duži rok. Pravilno upravljanje i raspoređivanje ovih resursa su od ključne važnosti za uredan i blagovremen završetak projektnih aktivnosti. U projektima razvoja softverskih sistema, ljudi su primarni resurs. Kvalifikovano osoblje kao što su programeri, tester, analitičari i menadžeri projekata igraju ključnu ulogu u uspehu ovih projekata. Još jedan dragocen resurs prilikom projektovanja informacionih sistema je vreme koje je na raspolaganju za obradu podataka od strane računara za potrebe testiranja, analize ili primene veštačke inteligencije. Obezbeđivanje adekvatnog vremena obrade može biti značajan izazov za menadžere projekata, jer direktno utiče na planiranje projekta. Stoga je efikasno upravljanje resursima od vitalnog značaja za prevazilaženje ovih izazova i osiguravanje da se projekti uspešno završe.

Rizik, iako nije eksplicitno deo tradicionalnog gvozdenog trougla predstavlja prožimajući element koji utiče na sve aspekte projekta. Njime se opisuje stepen neizvesnosti koje se mogu pojaviti tokom životnog ciklusa projekta, pod uticajem spoljašnjeg i unutrašnjeg okruženja projekta. Pošto rizici mogu uticati na bilo koju komponentu projekta, njihovo prisustvo i upravljanje su ključni za efikasno rukovanje sa ostalim pet ograničenja. Eksterno, rizici mogu proisteći iz faktora kao što su promene na tržištu, regulatorne promene ili problemi sa dobavljačima, potencijalno menjanje vremenskih rokova projekta, povećanje troškova ili zahtevi za prilagođavanjem opsega projekta. Interno, rizici mogu uključivati dostupnost resursa ili tehnološke izazove, što sve može ometati napredak i kvalitet projekta. Efikasno upravljanje rizikom uključuje identifikovanje, procenu i ublažavanje ovih neizvesnosti kako bi se njihov uticaj minimizirao. Proaktivnim rešavanjem rizika, menadžeri projekata mogu bolje da upravljaju promenama, da ostanu u okviru budžeta, da se pridržavaju rasporeda, održavaju standarde kvaliteta i optimizuju raspodelu resursa, na kraju povećavajući verovatnoću uspeha projekta.

U praksi se često sreće situacija da nerealna očekivanja od strane klijenata i prihvatanje takvog stanja od strane organizacije koja radi na projektu dovedu do neuspeha u realizaciji projekta. Jedan od alata koji se može koristiti u inicijalnoj fazi pregovora sa klijentom da olakša razumevanje međuzavisnosti faktora koji utiču na uspešnost realizacije projekta, jeste tabela ograničenja.

	1 (kritična važnost)	2	3	4 (najmanja važnost)
Opseg (Funkcionalnosti)			X	
Cena	X			
Vreme		X		
Kvalitet				X

U svojoj uobičajnoj formi tabela sadrži projektna ograničenja po redovima i ocenu njihove kritičnosti za klijenta po kolonama. Osnovno pravilo u popunjavanju tabele glasi da se ne mogu imati dve odabrane ćelije u istoj koloni (i redu). Tabelu ograničenja popunjava klijent. U prikazanom primeru, klijent se odlučio za cenu, odnosno budžet, kao primerno, najvažnije ograničenje. U ovom slučaju, budžet namenjen za izradu projekta se ne sme premašiti, čak i na uštrb funkcionalnosti i kvaliteta proizvoda. Drugo ograničenje po važnosti predstavlja vreme, što ukazuje na potrebu da projekat bude završen u skladu sa predviđenom dinamikom u što kraćem roku. Drugim rečima, klijent je spreman da žrtvuje funkcionalnosti koje će softverski projekat imati i kvalitet projekta, u smislu mogućih grešaka, lepog dizajna, intuitivnog interfejsa, zarad novca koji mu je na raspolaganju i vremena koje je potrebno da bi se projekat završio.

Iako zadatak koji klijent ima pred sobom prilikom odlučivanja u odnosu na kritičnost ograničenja nije ni malo jednostavan, zadatak menadžera projektom je da ne podleigne pritisku klijenta i da insistira na ispravnom popunjavanju tabele ograničenja, jer se jedino na ovaj način održava potreban balans između komponenti projekta koji omogućava da projekat bude uspešan. Preporuka prilikom razmatranja važnosti ograničenja je da se najpre odaberu najvažniji i najmanje važan aspekt projekta, a nakon toga popune preostala dva polja.

1.6. VREDNOST PROJEKTA

U troškove projekta spadaju svi rashodi koje organizacija ima za vreme trajanja projekta, a koji uključuju:

- Troškove osoblja uključenog u projekat.
- Troškovi materijala i opreme potrebnih za proizvodnju rezultata projekta. Kod razvoja softvera, ovi troškovi obično su vrlo niski.
- Troškovi određenog hardvera i softvera potrebnog za sprovođenje projekta.
- Putovanja, sastanci i događaji, odnosno troškovi potrebni za sastanke sa klijentima i ostalim zainteresovanim stranama.
- Troškovi vezani za rad konsultanata i podugovarača.
- Ostali troškovi, kao što su obuke i iznajmljivanje opreme.

Budžet projekta obuhvata predviđeni protok novca (prihode i rashode) koji se odnose na konkretan projekat. Osnovni zadatak upravljanja budžetom jeste da osigura da je novac dostupan u trenutku kada se ukaže potreba za projektnim troškovima i da se prati nivo troškova kako bi projekat ostao u okviru planiranog budžeta kao i preduzimanje potrebnih akcija za postizanje tog cilja. Protok novca se formira na osnovu predviđenih prihoda i rashoda za svaki period izveštavanja. Period izveštavanja zavisi od veličine projekta i politike kompanije. Raspodela prihoda i rashoda tokom vremena zavisi od plana projekta i ugovornih sporazuma sa podizvođačima, koji mogu zahtevati avansno plaćanje ili se isplata može obaviti u potpunosti nakon isporuke.

	K1	K2	K3	K4	Ukupno
Troškovi proj. tima	100,000	300,000	500,000	100,000	1,000,000
Eksterni troškovi	200,000	400,000	600,000		1,200,000
Ukupni troškovi	300,000	700,000	1,100,000	100,000	2,200,000
Uplate	500,000			2,000,000	2,500,000
Ukupni prihodi	500,000	0		2,000,000	2,500,000
Bilans	200,000	-700,000	-1,100,000	1,900,000	300,000
Finansijske potrebe		-500,000	-1,600,000		

U prikazanom primeru strukture budžeta projekta možemo razlikovati tri osnovne kategorije – troškove, prihode i bilans. Po kolonama se najčešće navode vrednosti za određene periode izveštavanja (u primeru, kvartale u okviru godine). U konkretnom slučaju, u pitanju je projekat u trajanju od godinu dana sa dva tipa troškova – troškovi osoblja zaposlenog unutar organizacije i eksterni troškovi prema trećim licima. Pod prihodima se podrazumevaju planirane uplate od strane sponzora. Bilans predstavlja razliku između prihoda i rashoda i računa se na kraju svakog perioda izveštavanja. Možda najvažniju stavku predstavlja red „finansijske potrebe“ koji pokazuje koliko novca projekat treba da „pozajmi“ kako bi sproveo aktivnosti za dati period. Na primeru projekta prikazanog u tabeli možemo videti da je organizacija akumulirala 200,000 dinara na kraju prvog kvartala, ali usled troškova od 700,000 dinara za drugi kvartal, organizacija potrebuje 500,000 dinara kredita kako bi uspešno završila projektne aktivnosti planirane za taj period. Za uspešan završetak trećeg kvartala organizacija mora da obezbedi 1,600,000 dinara akumuliranog bilansnog minusa, a taj novac će povratiti od sponzora na kraju četvrtog kvartala, kada će ostvariti i zaradu.

1.7. TRAJANJE AKTIVNOSTI

U kontekstu aktivnosti koje čine konkretan projekat možemo razlikovati tri faktora – napor, trajanje (aktivnosti) i resurse, čija međusobna zavisnost omogućava kvantitativno određivanje jednog od njih na osnovu vrednosti druga dva. Ova činjenica omogućava da odredimo broj neophodnog osoblja koje će raditi na projektu, stepen njihovog angažovanja i ukupno vreme trajanja aktivnosti.

Napor je količina posla potrebna za izvršenje zadatka i meri se korišćenjem čovek-sat, čovek-dan, čovek-mesec ili čovek-godina metrike, odnosno označava količinu posla koju jedan radnik obavi za jedan sat, dan, mesec ili godinu. Na taj način, na primer, aktivnost koja zahteva 40 čovek-sati može biti završena ako jedna osoba radi 40 sati. **Trajanje** predstavlja ukupno vreme neophodno za završetak aktivnosti, odnosno vremenski interval od početka do kraja aktivnosti. Meri se u kalendarskim jedinicama, kao što su dani, nedelje, meseci i godine. Glavni **resurs** potreban za projekte razvoja softvera je ljudski rad, koji se izražava u terminima jedinica rada, odnosno napora koji se može ostvariti tokom kalendarskog perioda. Ljudski rad se dodatno kvalifikuje identifikacijom vrste resursa potrebnog za obavljanje posla, odnosno utvrđivanjem koje vrste kompetencija i kakvo osoblje je potrebno. Na primer, razvoj softvera za automatsko detektovanje malignih bolesti može zahtevati angažman stručnjaka iz medicine.

Ljudski rad se određuje radnim kalendarom i procentom dostupnosti. Prvi faktor određuje maksimalan broj jedinica rada koje se mogu ostvariti tokom kalendarske jedinice. Za uslužnu industriju, tipične vrednosti za jedan resurs (jednu osobu) su 8 čovek-sati dnevno, odnosno 40 čovek-sati nedeljno, uzimajući u obzir 2 dana odmora nedeljno. Procentat dostupnosti smanjuje maksimalno prisustvo resursa i koristi se kada resurs nije dostupan puno radno vreme. Na primer, ako resurs radi na pola radnog vremena za organizaciju ili projekat, procentat dostupnosti može biti 50%.

U najvećem broju slučajeva, zavisnosti koja postoji između navedenih faktora se može izraziti sledećom jednačinom:

$$T = \frac{N}{R}$$

gde T predstavlja trajanje aktivnosti, N je napor potreban za završetak aktivnosti, a R označava ljudski rad neophodan kako bi se aktivnost uspešno sproveda. Od ove tri promenljive, obično se jedna procenjuje, druga bira, a treća izračunava.

Na primer, ukoliko želimo da izračunamo trajanje aktivnosti, neophodno je da procenimo napor koji je potreban kako bi se aktivnost završila. Neka to bude 120 čovek-sati, odnosno, jednoj osobi bi bilo neophodno 120 sati rada za uspešan završetak posla. Ukoliko imamo na raspolaganju tri zaposlena, koji su angažovani na različitim zadacima, te stoga njihova raspoloživost za rad na konkretnoj aktivnosti iznosi 30%, 45%, odnosno 60% njihove dostupnosti, koja pojedinačno iznosi 8 čovek-sati-dan, možemo izračunati vreme potrebno da se aktivnost završi.

$$T = \frac{120 \times \text{čovek}_{\text{sat}}}{0.3 \times 8 \text{čovek}_{\text{sat dan}} + 0.45 \times 8 \text{čovek}_{\text{sat dan}} + 0.6 \times 8 \text{čovek}_{\text{sat dan}}}$$

Odnosno:

$$T = \frac{120 \times \text{čovek}_{\text{sat}}}{2.4 \times \text{čovek}_{\text{sat dan}} + 3.6 \times \text{čovek}_{\text{sat dan}} + 4.8 \times \text{čovek}_{\text{sat dan}}} = \frac{120}{10.8} \approx 11 \text{ dana}$$

Kao drugi primer možemo navesti izračunavanje potrebnog angažovanja osoblja, ukoliko imamo ograničeno ukupno vreme za završetak aktivnosti. U ovom slučaju koristimo pomenutu formulu u obliku $R=N/T$. Ukoliko je aktivnost iz prethodnog primera ($N = 120$ čovek-sat) potrebno završiti u periodu od 2 nedelje, proizilazi da je potreban ljudski rad jednak 60 čovek-sat-nedelja. Drugim rečima, jedna osoba bi morala da radi 60 sati nedeljno, odnosno 150% svog radnog vremena. Ukoliko je na raspolaganju više zaposlenih, dobijeni ljudski rad se može preraspodeliti na njih tako da jedan od njih bude angažovan 40 sati, a drugi 20 sati nedeljno.

Nije na odmet napomenuti da, kada su u pitanju vrste projekata koji se ne odnose na razvoj informacionih sistema, postoje aktivnosti koje zahtevaju posebne vrste resursa:

- Materijal potreban za određene aktivnosti koji se troši tokom rada. Merna jedinica za materijal zavisi od vrste korišćenog materijala. Na primer, u fabrici za proizvodnju automobila, aktivnosti na montažnoj liniji zahtevale bi materijale kao što su metalne ploče, plastični delovi i elektronske komponente. Ovi materijali se koriste u izgradnji vozila.
- Oprema koja uključuje alate potrebne za izvođenje posla. Snimanje filmova, na primer, zahteva kamere, osvetljenje i uređaje za snimanje zvuka. Oprema se ne troši izvođenjem aktivnosti, to jest, nakon što je aktivnost završena, oprema se može koristiti u druge svrhe. Dostupnost opreme uvodi ograničenja u plan realizacije aktivnosti.

U praksi, napor je faktor koji se najčešće procenjuje od strane menadžera projekta, pri čemu uspešnost proračuna zavisi od iskustva i umeća koje menadžer poseduje. Međutim, nekada se mogu pojaviti projekti kod kojih je čak i uspešnim menadžerima teško da procene napor potreban da se projekat završi, što znatno otežava korišćenje anije pomenute formule. Iz tog razloga, 50-ih godina prošlog veka, razvijena je statistička tehnika koja omogućava procenu vremena kompletiranja projekta u situaciji kada je stepen neizvesnosti visok, pod nazivom **PERT** (Program Evaluation and Review Technique).

Korišćene algoritamske metode se u procesu procene softverskog sistema oslanjaju na matematičke modele koji uzimaju u obzir podatke prikupljene tokom godina. S druge strane, agilne metodologije razvoja radikalizuju ovaj proces kroz ponovljene procene u toku trajanja projekta dok se ne postigne konvergencija ili dok se projekat ne završi. Cilj tehnike je proceniti verovatnoću da se plan završi određenog datuma, uzimajući u obzir najverovatnije procene trajanje aktivnosti koje čine plan.

Da bi se koristio PERT, tri vrednosti moraju biti dodeljene svakoj aktivnosti: optimistična vrednost, najverovatnija vrednost i pesimistična vrednost. Zatim se izračunava najverovatnije trajanje aktivnosti:

$$T = \frac{O + 4 \times N + P}{6}$$

pri čemu je:

- O - optimistično vreme, odnosno, najkraće vreme koje aktivnost može da traje ako sve ide izuzetno dobro.
- P - pesimistično vreme, odnosno, najduže vreme koje aktivnost može da traje ako sve krene po zlu.
- N - najverovatnije vreme, to jest, najbolja procena vremena koje će aktivnost trajati u normalnim uslovima.

Dodatno, PERT izračunava meru promenljivosti ili nesigurnosti u procenjenom vremenu, poznatu kao standardna devijacija (SD) na sledeći način:

$$SD = \frac{P - O}{6}$$

Glavna prednost PERT-a je što uzima u obzir neizvesnosti i varijacije u zadacima projekta, što ga čini korisnim za projekte sa visokim stepenom neizvesnosti ili one koji uključuju složen, nov i tehnološki napredan rad.

Na kraju, nije na odmet pomenuti još jednu metodu procene vremena za izradu softverskog projekta koja se često primenjuje u slučaju „tipičnih“ projekata. Ako prihvatimo pretpostavku da jednostavan projekat zahteva malo planiranja, dok složen projekat zahteva više planiranja, korišćenje vremena provedenog u fazi planiranja može predstavljati podatak za procenu ukupnog vremena trajanja projekta. Na osnovu trajanja faze planiranja, ovaj pristup koristiti industrijske standarde za izračunavanje procena potrebnog vremena za ostale faze razvoja projekta. Industrijski standardi sugerišu da se u slučaju standardnog softverskog sistema troši 15% vremena u fazi planiranja, 20% u fazi analize, 35% u fazi dizajna, i 30% u fazi implementacije. To bi značilo da ako je za planiranje projekta potrošeno 60 dana, celokupan projekat će biti završen za 400 dana ili nešto više od godine, pod uslovom da je broj osoba koje rade na projektu nepromenjen sve vreme trajanja projekta. Slično tome, mogu se izračunati vremena ostalih faza – faza analize 80 dana, dizajna 140 dana i implementacije 120 dana.

1.8. KOŠTANJE PROJEKTA

Koštanje (cena) projekta predstavlja sumu novca koja se naplaćuje klijentu za završetak projekat. Prema Bruxam et al. (2013), postoje tri načina za određivanje cene softverskog projekta:

- Na osnovu troškova
- Na osnovu vrednosti
- Na osnovu konkurencije.

Cena zasnovana na troškovima određuje cenu projekta na osnovu troškova razvoja softvera. Troškovi povezani sa softverom uključuju proizvodnju i distribuciju. Cena zasnovana na vrednosti određuje cenu na osnovu toga koliko je kupac spreman da plati. Ovo omogućava fleksibilnije definisanje strategije cena, budući da se oslanja na percipiranu vrednost sistema. Na primer, Davidson (2012) predlaže korišćenje reputacije kompanije, pružanje bolje usluge i stvaranje brenda kako bi se postigla prednost u odnosu na konkurenciju. Napokon, cena zasnovana na konkurenciji određuje cenu na osnovu toga koliko konkurentna organizacija naplaćuje za izradu sličnog sistem. Pored ovakvog pristupa, najjednostavniji i verovatno najčešće korišćeni pristupi proceni koštanja projekta su stručna procena i analogija. Menadžer projekta u saradnji sa timom ili drugim stručnjacima, pruža procene koristeći svoje iskustvo ili analizirajući slične projekte. Ovaj pristup ima prednost što je vrlo brz i jednostavan.

U odnosu na vlasništvo nad projektom primenljive su tri šeme: licenciranje, zakup i prodaja. Kada kompanija licencira softverski sistem, daje pravo korisniku da koristi softver, dok zadržava vlasništvo nad softverom. Licenca se obično prodaje, a novac ostvaren od licenci koristi se za plaćanje troškova razvoja i održavanja. Licence se prodaju na različite načine od kojih je najčešći prodaja po kopiji. Veće organizacije često imaju koristi od masovnih kupovina, gde se uz popust kupuje pravo korišćenja više instanci softvera. Masovne kupovine mogu biti po korisniku, po licenci ili po instanci, u zavisnosti od toga da li se broj veže za ljude koji imaju pravo korišćenja sistema, za računare na kojima se softver može pokrenuti ili za maksimalni broj instanci koje mogu raditi u isto vreme.

Poseban tip licence predstavljaju **softveri otvorenog koda** (open source) i **slobodni softveri** (free software). Softver otvorenog koda i slobodan softver su često korišćeni izrazi koji opisuju modele licenciranja i razvoja softvera koji omogućavaju korisnicima pristup, izmenu i distribuciju izvornog koda softvera. Pored softvera otvorenog koda i slobodnog softvera (free software), često se u

praksi sreće termin **besplatan softver** (freeware). Slobodan softver se odnosi na softver koji se distribuira pod uslovima koji korisnicima pružaju slobodu da pokreću, proučavaju, menjaju i distribuiraju softver i njegov izvorni kod. Termin „free“ ovde se odnosi na slobodu, a ne nužno na cenu. Slobodan softver se pridržava skupa principa poznatih kao "četiri slobode":

- sloboda da se program koristi u bilo koju svrhu.
- sloboda da se proučava kako program radi i da se modifikuje po želji.
- sloboda da se distribuira kopija originalnog softvera.
- sloboda da distribuira izmenjena verzija softvera.

Slobodan softver često koristi licence poput GNU javne licence (GPL), koja obezbeđuje da softver i njegove modifikacije ostanu slobodni i otvoreni.

Sa druge strane, besplatan softver se odnosi na softver koji se pruža besplatno, što znači da ga ne morate platiti da biste ga koristili. Međutim, freeware nužno ne pruža isti nivo slobode kao slobodan softver. Termin „free“ ovde se odnosi na cenu, a ne nužno na sposobnost izmene ili redistribucije softvera. Freeware može imati ograničenja u vezi sa načinom korišćenja ili distribucije softvera. Na primer, neki freeware može biti namenjen samo za ličnu upotrebu, dok bi komercijalna distribucija mogla biti ograničena. Korisnici možda neće imati pristup izvornom kodu, što ograničava njihovu sposobnost da menjaju ili razumeju kako softver funkcioniše.

U zaključku, ključna razlika između slobodnog i besplatnog softvera je u slobodama koje pružaju. U prvom slučaju, licenca se odnosi na pružanje korisnicima slobode da koriste, proučavaju, menjaju i distribuiraju softver i njegov izvorni kod, dok druga jednostavno podrazumeva da je softver dostupan bez troškova, bez obaveznog pružanja ovih dodatnih sloboda. Organizacija Free Software Foundation (FSF) zagovara ove principe putem svoje GPL i drugih licenci. Fokus je na slobodi korisnika i kontroli nad softverom koji koriste.

Softver otvorenog koda se odnosi na softver čiji je izvorni kod dostupan javnosti, omogućavajući programerima da pregledaju, menjaju i distribuiraju kod. Organizacija Open Source Initiative (OSI) definiše kriterijume poput slobodne redistribucije, pristupa izvornom kodu i slično, koji moraju da budu zadovoljeni u slučaju softvera otvorenog koda. Iako i slobodan softver i softver otvorenog koda dele princip pristupa izvornom kodu, ključna razlika leži u njihovim osnovnim filozofijama. Slobodan softver proističe iz filozofije slobode korisnika, osiguravajući da korisnici imaju slobodu da kontrolišu i menjaju softver, dok softver otvorenog koda ističe transparentnost, saradnju i praktične koristi deljenja koda. Distinkcija može biti suptilna, a izbor između ovih pristupa često zavisi od prioriteta i vrednosti koje programer ili organizacija promovišu.

Zakup je druga šema u kojoj se pravo korišćenja softvera prodaje na određeni period. Korisnici moraju redovno obnavljati svoje licence. Slično kao i u mnogim drugim tržištima, u nekim slučajevima početni trošak je postavljen nisko kako bi se smanjila prepreka za ulazak na tržište. Zakup je veoma popularan kod veb aplikacija, jer pristup sistemu direktno kontroliše kompanija koja izdaje softver u zakup, koja može dodeliti pristup ili ga opozvati po potrebi. U odnosu na prethodni model, zakup ima neke prednosti kako za klijenta, tako i za prodavca. Klijent može dobiti proizvod po nižem početnom trošku, a organizacija ima model prihoda koji garantuje stalne, stabilnije prihode. Na kraju, softver se može prodavati, pri čemu klijent preuzima vlasništvo nad projektom.

1.9. ISPLATIVOST PROJEKTA

U praksi se najčešće primenjuju dve metode za izračunavanje isplativosti projekta. Najjednostavnija finansijska procena isplativosti projekta je metoda **vraćanje uloženog**, koja meri koliko će vremena biti potrebno da se povрати ono što je u projekat investirano. Prilikom korišćenja ove metode, procena troškova i prihoda projekta određuje dobitke i gubitke na kraju svake godine projekta. Vraćanje uloženog je godina u kojoj projekt pokriva sve troškove i počinje da ostvaruje dobit. Što je kraći povraćaj uloženog, to je bolje. Metoda vraćanja uloženog favorizuje projekte koji minimiziraju finansijski rizik. Jedan od problema sa ovom metodom je da ne uzima u obzir ukupnu dobit. Drugi problem je da ne meri efikasnost kojom se uloženi novac u projekt vraća.

Da bi se prevazišle neke od ograničenja metode vraćanja uloženog, često se koristi još jedna metoda, a to je **povraćaj investicije** (return of investment - ROI), koja meri koliko dobijamo nazad za svaki uloženi dinar. ROI se računa iz godišnje dobiti, definisane kao prosečne dobiti po godini koja se dobija deljenjem dobiti sa trajanjem projekta:

$$GD = \frac{P - T}{G}$$

gde je GD tražena godišnja dobit, P predstavlja prihode projekta, T označava nastale troškove, a G obeležava trajanje projekta u godinama kao kalendarskim jedinicama.

Na osnovu godišnje dobiti može se izračunati povraćaj investicije deljenjem dobijene vrednosti sa ukupnim troškovima na sledeći način:

$$ROI = \frac{GD}{T}$$

Razmotrimo dva projekta - "Projekat A" i "Projekat B", za koje imamo procenjene troškove i prihode kako je opisano u Tabeli 3. Konkretno, tabela pokazuje da Projekat A nije profitabilan u prve 2 godine, a zatim počinje da donosi dobit. Projekat B ima slično ponašanje, ali i troškovi i prihodi su veći. Pretpostavimo da imamo resurse da započnemo samo jedan od ova dva projekta i koristimo finansijsku metodu za odabir koji projekat aktivirati.

Ako koristimo metodu vraćanja uloženog, bираmo Projekat A, jer ima kraći period povraćaja investicije. Naime, za Projekat A se predviđa da će završiti sa gubicima na kraju godine 0 i godine 1. Međutim, na kraju godine 2, finansijski izveštaj pokazuje dobit od 100,000 dinara. Sa druge strane, u slučaju Projekta B, godine 0 i 1 završavaju sa gubicima. U godini 2, Projekat B ostvaruje dobit od 700,000 dinara, ali ovi prihodi još uvek nisu dovoljni da pokriju troškove prve 2 godine, dok u godini 3 projekat vraća početnu investiciju i zarađuje. Dakle, pošto će se uloženo u projekat A vratiti za tri godine, a u Projekat B za četiri godine, na osnovu metode vraćanja uloženog novca započinjemo Projekat A.

	Projekat A	Projekat B
Godina 0	- 300,000 rsd	- 500,000 rsd
Godina 1	- 200,000 rsd	- 300,000 rsd
Godina 2	600,000 rsd	700,000 rsd
Godina 3		1,000,000 rsd

Drugačija je situacija ako koristimo ROI metod. Izračunavanje vrednosti za Projekat A bi izgledalo ovako:

$$GD = \frac{600\,000 - (200\,000 + 300\,000)}{3} = 33333.33$$

$$ROI = \frac{33\,333.33}{500\,000} = 0.6 = 6\%$$

U slučaju Projekta B, računica pokazuje drugačiju vrednost:

$$GD = \frac{1000\,000 + 700\,000 - (300\,000 + 500\,000)}{4} = 225\,000$$

$$ROI = \frac{225\,000}{800\,000} = 0.28 = 28\%$$

Zaključak koji se može izvesti korišćenjem finansijske metode povraćaja uloženog govori da treba izabrati Projekat B, jer za uloženi novac pruža veću vrednost.

1.10. RIZICI U RAZVOJU SOFTVERSKOG PROJEKTA

U najširem smislu, **rizik** bi se mogao definisati kao verovatnoća ostvarenja pretnje koja dovodi do gubitka, oštećenja ili uništenja svojine (ili neke vrednosti) organizacije. Barry Boehm je razvio prvu i verovatno najpoznatiju listu rizika za softverske sisteme (Boehm, 1988). Njegova analiza obuhvata sledeće glavne razloge zbog kojih projekti ne uspevaju:

- Nedostatak osoblja – problemi u angažovanju osoblja sa odgovarajućim veštinama i iskustvom mogu dovesti do poteškoća na projektu.
- Nerealni raspored i budžet – izazovi u tačnom procenjivanju vremena i resursa potrebnih za projekat mogu rezultirati rasporedima i budžetima koji su teško ostvarivi.
- Razvoj pogrešnih softverskih funkcija i loš korisnički interfejs – nerazumevanje ili netačno tumačenje zahteva može dovesti do razvoja softvera koji se ne poklapa sa potrebama i očekivanjima korisnika.
- „Zlatno oblaganje“ (Gold-plating) – razvoj nevažnih funkcija ili karakteristika. To može poticati kako od klijenta, koji može tražiti funkcije koje zapravo nisu potrebne, tako i od tima, koji se može baviti funkcijama nižeg prioriteta koje su zabavnije za razvoj.
- Nedostaci u eksterno obavljenim zadacima – situacije kada podugovarači ili spoljni timovi ne isporuče rad očekivanog kvaliteta, što može negativno uticati na ceo projekat.
- Nedostatak performansi i izazovi u dostupnim tehnologijama – izgradnja sistema visoke tehničke složenosti može dovesti do problema sa performansama i teškoća u prevazilaženju tehničkih prepreka.

Pored ovih rizika, takođe je važno pomenuti nedostatak uključenosti zainteresovanih strana, neadekvatno upravljanje promenama i nedostatak odgovarajuće metodologije upravljanja projektom.

Nakon identifikovanja rizika u projektu, sledeći korak predstavlja određivanje rizika koje treba pratiti. Najčešće se rizici razlažu na dve osnovne komponente:

- Verovatnoća (probability) da će se rizik dogoditi.
- Uticaj (impact) koji će rizik imati.

Kombinacija verovatnoće i uticaja određuje da li je rizik vredan daljeg praćenja ili ne. Rizik je direktno srazmeran stepenu pretnje, ozbiljnosti propusta i vrednosti svojine. Postoje dva osnovna pristupa određivanju verovatnoće i uticaja: kvalitativni i kvantitativni.

Matrica nivoa rizika, čiji je primer prikazan na slici, predstavlja najčešće korišćeni metod u kvalitativnoj analizi.

Verovatnoća ostvarenja pretnje		Uticaj na projekat (svojinu organizacije)				
		Beznačajna	Manja	Srednja	Ozbiljna	Veoma ozbiljna
81-100%	Veoma velika	NIZAK	SREDNJI	VISOK	EKSTREMAN	EKSTREMAN
61-80%	Velika	NIZAK	SREDNJI	VISOK	VISOK	EKSTREMAN
41-60%	Srednja	NIZAK	SREDNJI	SREDNJI	VISOK	VISOK
21-40%	Mala	NIZAK	NIZAK	SREDNJI	SREDNJI	VISOK
1-20%	Veoma mala	NIZAK	NIZAK	SREDNJI	SREDNJI	VISOK

Broj i nazivi redova i kolona u tabeli nisu strogo definisani i određuju se za svaki konkretan projekat. U prikazanom primeru, verovatnoća je klasifikovana u 5 kategorija – veoma mala, mala, srednja, velika, veoma velika, a na osnovu procenta da se pretnja ostvari. Po redovima je navedena ozbiljnost posledice (uticaj) koji bi ostvarenje pretnje imalo na projekat, odnosno organizaciju. Svaki identifikovani rizik se smešta u određenu klasu, određenu verovatnoćom i posledicom. Na primer, rizik po realizaciju projekta može biti bolest člana razvojnog tima. U tom slučaju, određuje se verovatnoća da se to desi. Verovatnoća oboljevanja može biti veća u periodima epidemije ili u jesen i zimu. Sa druge

strane, procenjuje se koliko je ozbiljna posledica po projekat ukoliko jedan član tima ne može da učestvuje u njegovom razvoju neko određeno vreme usled bolesti. Ukoliko bi se ovom riziku dodelila velika verovatnoća, a posledica po uspešnost projekta bila ozbiljna, pomenuti rizik bi se klasifikovao kao „visok“.

U navedenom primeru matrice korisno je napomenuti dva granična slučaja. Prvi je situacija kada je uticaj na organizaciju beznačajan, bez obzira na verovatnoću ostvarenja pretnje, rizik je nizak. Sa druge strane, ukoliko je uticaj veoma veliki, rizik je najmanje „visok“, čak i u slučajevima kada je verovatnoća da se pretnja ostvari minimalna.

Poželjno je da procene rizika prate razvojni ciklus aplikacije i da započnu što ranije. Dobra praksa je da testiranje vrši osoba koja nije član razvojnog tima. Izlaz procene predstavlja identifikacija rizika, pretnji i preporuka za merama za umanjene rizika. Jedna od popularnih metodologija razvijenih u ovu svrhu nosi naziv FRAAP (Facilitated Risk Analysis and Assessment Process). **FRAAP** predstavlja strukturiran pristup ubrzanoj proceni svake komponente sistema za kratko vreme. Model zahteva strog raspored i podelu dužnosti. Osnovni koraci FRAAP metodologije podrazumevaju:

- Pre-FRAAP sastanak koji traje oko sat vremena na kome učestvuju član višeg menadžmenta, vođa projekta i posrednik (vođa diskusije). Cilj je ustanoviti zadatke, agendu, format sesije i učesnike.
- FRAAP sesiju koja traje od 4 do 8 sati, uključuje članove razvojnog tima (do 15 članova) i koja za cilj ima razgovor o rizicima. Oslanja se na ekspertizu i znanje ljudi koji u njemu učestvuju. Kao izlaz sesije treba da se dobije:
 - identifikovanje rizika,
 - prioritete rizika,
 - identifikovanje kontrole za umanjene rizika najvišeg prioriteta,
- Jedan od načina vođenja sesije podrazumeva da svaki učesnik sesije redom (u intervalu od 3 minuta) nabroji rizike koje smatra značajnim.
- FRAAP analizu i pisanje izveštaja (traje 4 do 6 dana). Obavlja se na osnovu rezultata FRAAP sesije i podrazumeva analizu i kategorizaciju (korišćenjem matrice rizika npr.).
- Post-FRAAP sesiju koja traje oko sat vremena i ima iste učesnike kao i Pre-FRAAP sesija.

Kada se rizici odrede i klasifikuju po prioritetima vrši se odabir strategije postupanja sa rizikom sa ciljem smanjivanja verovatnoće ili posledice:

- Prihvatanje rizika – donošenje odluke da se rizik jednostavno prihvati i da se ne preduzimaju nikakve akcije dok se pretnja (potencijalno) ne ostvari.
- Izbegavanje rizika – prilagođavanje projektnog plana na način da se aktivnosti ili radnje koje mogu dovesti do ostvarenja pretnje uklone. Ovo je najoprezniji pristup, ali ga je nemoguće primeniti na sve rizike.
- Ublažavanje rizika – prilagođavanje projekta na način da se smanji verovatnoća ili posledica rizika. Izbegavanje rizika zapravo predstavlja poseban slučaj ublažavanja rizika u kome je verovatnoća ostvarenja pretnje svedena na nulu. Postoje tri načina da se umanjí rizik:
 - smanjivanje verovatnoće ostvarivanja pretnje
 - ograničavanje ozbiljnosti posledica koje nastaju
 - umanjivanje osetljivosti (važnosti) resursa
- Transfer (prenos) rizika – prebacivanje odgovornosti za rizik na neku treću stranu koja je sposobna da se suoči sa pretnjom ukoliko se ona pojavi. U slučaju da je odabrana opcija transfera rizika na treću stranu, može se angažovati osiguravajuća kuća da pokrije eventualne gubitke usled incidenta. Ova strategija se može posmatrati kao način da se uticaj po organizaciju svede na nulu, uz određeni trošak koji sa sobom nosi transfer rizika.

Ako bismo kao primer uzeli kategorizaciju rizika navedenu ranije u tabeli matrice rizika, primena strategija postupanja sa rizicima bi mogla da izgleda kao što je prikazano u narednoj tabeli:

NIZAK	SREDNJI	VISOK	EKSTREMAN	EKSTREMAN
NIZAK	SREDNJI	VISOK	VISOK	EKSTREMAN
NIZAK	SREDNJI	SREDNJI	VISOK	VISOK
NIZAK	NIZAK	SREDNJI	SREDNJI	VISOK
NIZAK	NIZAK	SREDNJI	SREDNJI	VISOK

ZONA 3

ZONA 2

ZONA 1

Kategorije rizika su podeljene u tri grupe na osnovu izračunatog stepena (prioriteta) rizika – Zona 1 (zelena), Zona 2 (žuta) i Zona 3 (crvena). Podela u zone omogućava da se za svaki rizik odredi odgovarajuća strategija. Tako, na primer, možemo odrediti sledeća pravila za postupanje sa rizikom:

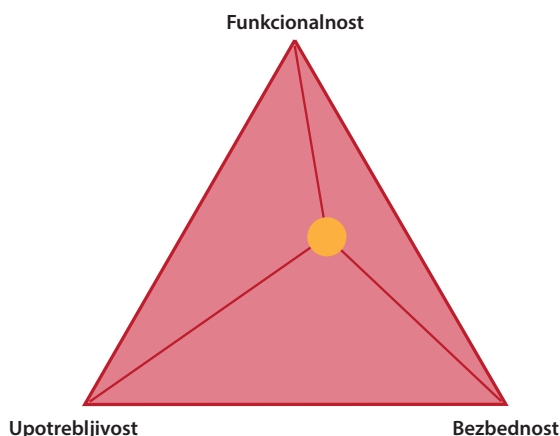
- Rizici koji pripadaju Zoni 1 će biti prihvaćeni. U ovu grupu spadaju rizici kod kojih je verovatnoća ostvarenja ili posledica takva da su troškovi ublažavanja veći od problema do kojih može doći ako se pretnja ostvari.
- Rizici iz Zone 3 moraju da budu uklonjeni iz Zone 3. Ovo su rizici na koje se mora primeniti jedna od tri strategije – izbegavanje, ublažavanje ili prenos. Primenom strategije izbegavanja, rizik se može potpuno ukloniti iz tabele (jer se uklanja pretnja, pa je samim tim verovatnoća ostvarenja jednaka nuli). Odabir strategije ublažavanja ili transfera rizika, za cilj ima premeštanje rizika iz Zone 3 u Zonu 2 ili Zonu 1.
- Rizicima iz Zone 2 se mora upravljati korišćenjem jedne od tri aktivne strategije – izbegavanje, ublažavanje ili transfer. Za svaki rizik pojedinačno se određuje najadekvatnija strategija sa ciljem smanjenja verovatnoće ili uticaja na projekat (organizaciju).

Određivanje zona prikazano u prethodnom primeru nije striktno definisano, već zavisi od potreba konkretnog projekta i iskustva osobe koja je zadužena za proces upravljanja rizicima.

1.11. FUS (FUNCTIONALITY, USABILITY, SECURITY) TROUGAO

Trougao Funkcionalnosti, Upotrebljivosti, Sigurnosti (FUS) jeste jedan od osnovnih koncepta u dizajniranju i razvoju softvera i digitalnih sistema. Predstavlja osetljivu ravnotežu koja mora biti očuvana kako bi se osiguralo da sistem zadovoljava potrebe i očekivanja korisnika, istovremeno štiteći njihove podatke i održavajući celokupni integritet sistema. Ovaj trougao je u srcu procesa donošenja odluka za razvojni tim softverskog projekta.

Funkcionalnost se odnosi na osnovne karakteristike i mogućnosti koje sistem treba da pruži. U osnovi, odgovara na pitanje „Šta ovaj softver može da radi?“. Da bi se osigurao uspešan proizvod, ključno je pravilno definisati funkcionalnosti na osnovu potreba korisnika i poslovnih zahteva. **Upotrebljivost** se fokusira na iskustvo korisnika pri interakciji sa softverom ili sistemom. Obuhvata aspekte kao što su dizajn korisničkog interfejsa, lakoća upotrebe i ukupno zadovoljstvo korisnika.



Sistem može imati snažnu funkcionalnost, ali ako nije pristupačan za korišćenje, možda neće biti široko prihvaćen ili može izazvati frustraciju korisnika. Dizajniranje usmereno na krajnjeg korisnika, obavljeno na osnovu sprovedenog istraživanja korisnika, predstavlja ključni faktor u kreiranju intuitivnog i efikasnog sistema. Upotrebljivost treba shvatiti kao kontinuiran proces. Prikupljanje povratnih informacija od korisnika i neprestano prilagođavanje korisničkog interfejsa može dovesti do značajnih poboljšanja u upotrebljivosti tokom vremena. **Sigurnost** je aspekt FUS trougla koji se bavi zaštitom sistema, podataka i korisnika od pretnji i ranjivosti. Siguran sistem je ključan za zaštitu osetljivih informacija, održavanje poverenja korisnika i sprečavanje neovlašćenog pristupa ili zlonamernih radnji. Ključni mehanizmi koje je poželjno primeniti obuhvataju implementiranje enkripcije podataka, pravilno upravljanje autentifikacijom i autorizacijom korisnika, kao i redovne (interne i eksterne) procene ranjivosti sistema.

Narušavanje ravnoteže između navedena tri faktora predstavlja jedan od najčešćih uzroka neuspeha projekta. Svaka funkcionalnost koja od korisnika zahteva unos određenih podataka podrazumeva dizajniranje korisničkog interfejsa prilagođenog konkretnom zadatku. Poznato je da svaki unos podataka nosi potencijalni bezbednosni rizik u slučaju kada se podaci ne proveravaju i obrađuju na ispravan način. Iz ovoga proizilazi da sa povećavanjem broja funkcionalnosti i usložnjavanjem interfejsa bezbednosni izazovi rastu. Sa druge strane, ograničavanje funkcionalnosti i smanjivanje broja ulaza kroz kreiranje jednostavnije interakcije korisnika sa softverom može značajno olakšati upravljanje bezbednosnim aspektima projekta. Zadatak vođe projektnog tima jeste da balansira između ovih zahteva kako bi projekat zadovoljio očekivanja korisnika, ne narušavajući pri tome podrazumevani princip bezbednosti sistema.

1.12. KLASIFIKACIJA SOFTVERSKIH PROJEKATA

Softverski projekti se mogu klasifikovati u različite kategorije na osnovu njihovog obima, veličine, svrhe i metodologije razvoja. Uobičajene podele projekata obuhvataju:

- Po nivou rizika – Softverski projekti se mogu podeliti na osnovu nivoa rizika koji se odnosi na neizvesnost ili potencijalni uticaj negativnih ishoda. Visokorizični projekti su oni koji imaju veći potencijal za neuspeh ili negativne posledice, dok niskorizični projekti imaju manji potencijal za neuspeh ili negativan uticaj.
- Po vrednosti koju donose organizaciji – Neki projekti se mogu proglasiti za strateški važne projekte, što označava da donose visoku vrednost organizaciji i smatraju se prioritetom.
- Po trajanju – Organizacija definiše kategorije projekata na osnovu vremena potrebnog za završetak projekta. Na primer, kratkoročni projekti se mogu završiti za nekoliko nedelja, a najviše za dva meseca. Druge kategorije mogu biti određene trajanjem projekta u mesecima kao jedinici vremena – dva do četiri meseca, tri do šest meseci i sl., dok dugoročni projekti mogu trajati godinama.
- Po složenosti – Softverski projekti se mogu klasifikovati na osnovu složenosti, što se odnosi na stepen težine razvoja. Primer kategorija može biti niska složenost, srednja složenost i visoka složenost. Jednostavni projekti imaju niži nivo složenosti i mogu se završiti sa manjim timom i manje resursa, dok složeni projekti zahtevaju više resursa i veći tim. Kao i u prethodnim slučajevima, nazivi i broj kategorija nisu striktno definisani, već ih svaka organizacija određuje u skladu sa svojim potrebama.
- Po vrsti korišćenih tehnologija – Ova vrsta podele se u najvećem broju slučajeva odnosi na stepen poznavanja tehnologija potrebnih za razvoj projekta. Organizacija poseduje kadar koji raspolaže različitim stepenom poznavanja programskih jezika, baza podataka ili operativnih sistema. U zavisnosti od toga projekti se mogu svrstati u kategoriju „dobro poznatih tehnologija“ ili „nikad korišćenih tehnologija“. Na osnovu ove podele organizacija može angažovati eksterne resurse ili zaposliti nove ljude sa potrebnim kompetencijama kako bi projekat mogao da bude uspešno realizovan.

- Po broju članova tima – Softverski projekti se mogu klasifikovati na osnovu broja članova tima koji su uključeni u razvoj. Tako, možemo imati male projekte sa nekoliko članova, projekte srednje veličine ili velike projekte sa velikim timom. Ova klasifikacija može pomoći organizacijama da razumeju potrebe za zapošljavanjem i resursima za projekat.

Kada se za konkretan projekat odrede sve definisane kategorije, može se odrediti opšta klasa kojoj projekat pripada. Na primer, možemo imati projekt koji je ekstremno rizičan i visokovredan, sa dugim trajanjem i visokom složenosti, sa korišćenjem nepoznatih tehnologija u razvoju i sa potrebom angažovanja velikog tima. Ovaj projekat bi se mogao klasifikovati kao **Tipa A**. Da bi se maksimalno uvećala verovatnoća uspeha, organizacija zahteva da ovi projekti koriste sve metode i alatke dostupne u njihovoj metodologiji upravljanja projektima. Slično tome, visokorizičan i visokovredan projekat, srednje kompleksnosti, trajanja od devet do dvanaest meseci, razvijan poznatim tehnologijama može se svrstati u kategoriju projekata **Tipa B**. Sa druge strane, možemo imati projekat srednje ili male vrednosti, niskog nivoa rizika i sa često korišćenim tehnologijama koji bi mogao pripadati projektima **Tipa C**. I na kraju, najčešće postoji i četvrta kategorija projekata, niske vrednosti i rizika, koji traju kratko i najčešće podrazumevaju izmenu ili održavanje postojećih sistema, koji se klasifikuju kao **Tip D**.

Kategorizacijom projekta na osnovu ovih klasifikacija se dobija bolje razumevanje karakteristika i zahteva projekta, na osnovu čega se mogu definisati strategije planiranja i upravljanja projektom. Važno je imati u vidu da pomenute podele nisu međusobno isključive i da projekat može imati karakteristike koje se uklapaju u više kategorija.

2. ANALIZA I SPECIFIKACIJA ZAHTEVA

Loša komunikacija sa klijentom, nedovoljno definisani zahtevi i nejasni poslovni ciljevi doprinose neuspehu mnogih IT inicijativa. Neki timovi za razvoj softvera se suočavaju sa poteškoćama u dobijanju zahteva od klijenata iz različitih razloga, kao što su nedostatak vremena ili strpljenja kod korisnika za učešće u neophodnim aktivnostima, ili čak neslaganje među učesnicima projekta oko definicije "zahteva". Zahtev određuje zadatke koje rešenje mora da ispuni, ali ne i način na koji to treba sprovesti. Definisanje softverskih zahteva podrazumeva metode za identifikaciju i opisivanje karakteristika sistema koji treba izgraditi. Glavni proizvod ove aktivnosti su rezultati koji opisuju ponašanje sistema, odnosno funkcije koje mora da implementira i svojstva koja mora da poseduje.

2.1. VRSTE ZAHTEVA

U najširem smislu, zahtevi se mogu podeliti u četiri kategorije: poslovni zahtevi, korisnički zahtevi, funkcionalni zahtevi i nefunkcionalni zahtevi.

Poslovni zahtevi u osnovi opisuju razloge zašto organizacija želi da implementira određeni softverski projekat i koja su očekivanja u kontekstu povećanja poslovne vrednosti. Poslovni zahtevi obično dolaze od sponzora projekta ili kupca koji nabavlja proizvod. Međutim, identifikovanje i definisanje poslovnih prednosti koje projekat donosi može biti izazovan proces. Može se desiti da članovi tima nisu sigurni šta projekat treba da postigne ili sponzori nisu postavili ciljeve na način koji omogućava efikasno merenje stepena njihove ispunjenosti. Na kraju, moguća je situacija da različite zainteresovane strane nisu saglasne oko toga šta bi ciljevi trebalo da budu.

Dva osnovna elementa poslovnih zahteva su **vizija** i **opseg** projekta. Vizija sažeto opisuje krajnji proizvod koji treba da postigne poslovne ciljeve. Opis vizije treba da odražava uravnotežen pogled koji će zadovoljiti očekivanja različitih zainteresovanih strana. Sa druge strane, opseg identifikuje na koji deo krajnje vizije proizvoda se trenutni projekat odnosi. Opsegom se razgraničava ono što je uključeno i ono što nije uključeno u konkretan projekat, a tiče se ostvarenja vizije. Kada neko iznese predlog za novim zahtevom projekta, najpre se mora utvrditi da li zahtev spada u okvir definisanog opsega projekta. Ako je to slučaj, novi zahtev se može uključiti u trenutni projekat pod uslovom da je visokog prioriteta u odnosu na druge zahteve koji već postoje. Uključivanje novih zahteva

često podrazumeva odluku o odlaganju ili otkazivanju drugih planiranih zahteva ili produženju trajanja projekta. U situaciji kada zahtev ne spada u opseg trenutnog projekta, on ne može postati njegov sastavni deo, već se može doneti odluka o odlaganju zahteva za neki naredni projekat.

Dokument u kome se navode poslovni zahtevi treba u sebi da sadrži opis poslovnog problema koji rešava ili procesa koji unapređuje, zatim, poslovnu priliku koja postoji na tržištu i koja se rešava implementiranjem projekta i koje su njegove prednosti u odnosu na konkurenciju, kao i metriku koja će se koristiti za merenje uspešnosti projekta. Metrike imaju za cilj da pokažu da li projekat ide u pravcu ispunjenja svojih poslovnih ciljeva i mogu se pratiti tokom testiranja projekta ili neposredno nakon puštanja proizvoda u upotrebu. U okviru dokumenta sa poslovnim zahtevima, neophodno je identifikovati *poslovne* rizike povezane sa razvojem projekta koji su različiti od *projektnih* rizika koji spadaju u opseg delovanja menadžera projekta. U poslednjem delu se najčešće ističu konkretne prednosti koje projekat donosi različitim zainteresovanim stranama, kao što su smanjenje troškova proizvodnje, ubrzavanje poslovnih procesa, automatizacija zadataka, uvođenje novih proizvodnih procesa, usaglašavanje sa pravnom regulativom i sl.

Korisnički zahtevi opisuju zadatke koje korisnici moraju biti u mogućnosti da obave sa sistemom i opis karakteristika proizvoda koji su važni za korisničko iskustvo. Jedan od popularnih načina predstavljanja ovih zahteva jeste u vidu korisničkih priča (eng. *user story*):

Želim da uradim [...] kao korisnik [...] da bih ostvario [...]

Prednost ovog metoda leži u tome što eksplicitno identifikuje zahtev na način da sadrži ulogu (privilegije) koje korisnik treba da ima zajedno sa funkcijom koja treba da se izvrši i razlogom zbog koga se izvršava.

Druga tehnika navođenja korisničkih zahteva jesu slučajevi korišćenja (eng. *use case*) koji su predstavljeni u vidu niza interakcija između sistema i spoljnog aktera (uloge) koje se zapisuju u obliku glagola praćenog imenicom. Akter može biti osoba ili softverski sistem koji obavlja interakciju sa sistemom kako bi izvršio slučaj korišćenja. Neki od primera slučajeva korišćenja su: „prijavljivanje na sistem“, „dodavanje korisnika“, „pomeranje figure na šahovskoj tabli“. Detaljnije objašnjenje i grafičke reprezentacije pomenutih tehnika će biti obrađene u narednim poglavljima.

Funkcionalni zahtevi određuju šta proizvod treba da radi ili kako da se ponaša kako bi ispunio potrebe korisnika. Za razliku od korisničkih zahteva koji se odnose na aktivnosti spoljnih aktera, funkcionalni zahtevi se bave aktivnostima *softvera* koje će omogućiti da se konkretna korisnička akcija uspešno izvrši. Na primer, ukoliko pretpostavimo da je akter student, a zadatak koji želi da obavi prijavljivanje ispita u junskom roku, funkcionalni zahtevi povezani sa ovim zadatkom bi mogli biti: proveriti da li je student izmirio obaveze prema fakultetu, proveriti da li je student ranije položio predmet, proveriti da li student ima pravo da polaže ispit iz željenog predmeta, proveriti da li je prijavljivanje u junskom roku aktivno, potvrdi prijavu ispita, pošalji obaveštenje predmetnom profesoru itd. Funkcionalni zahtevi omogućavaju programerima da razumeju tačno kako treba da implementiraju određene funkcionalnosti u sistemu. Dokument u kome se navode funkcionalni zahtevi kreira analitičar ili rukovodilac projekta i može nositi različite nazive, kao što su Specifikacija zahteva softvera, Funkcijska specifikacija, Dokument sa aplikativnim zahtevima i sl.

Nefunkcionalni zahtevi opisuju kvalitete ili karakteristike koje sistem mora da ima, ali se ne odnose direktno na specifična ponašanja ili funkcionalnosti sistema. Njima se definiše kako sistem treba da se ponaša, umesto šta sistem treba da radi. Primeri nefunkcionalnih zahteva uključuju performanse, bezbednost, pouzdanost, upotrebljivost i skalabilnost. Mnogi proizvodi su pod uticajem zahteva za usklađenost sa pravnom regulativom i sertifikacijom. Mogu postojati zahtevi za lokalizaciju proizvoda koji moraju uzeti u obzir kulturu, jezik, valutu, terminologiju, pravopis i druge karakteristike korisnika. U najvećem broju slučajeva postoje načini za merenje nivoa usklađenosti proizvoda sa definisanim nefunkcionalnim zahtevima. Osobine sistema koje je potrebno analizirati u procesu utvrđivanja ispunjenosti nefunkcionalnih zahteva obuhvataju: u kojoj meri su usluge sistema dostupne kada i gde su potrebne (dostupnost), koliko je jednostavno instalirati i deinstalirati softver, u kojoj meri sistem štiti od gubitka ili curenja podataka (bezbednost), kolika je sposobnost sistema da komunicira sa drugim softverima, koliko brzo sistem reaguje na korisničku akciju (performanse), kako sistem reaguje na nepredviđene primere korišćenja (robusnost ili otpornost na greške), koliko je sistem intuitivan za učenje i korišćenje (upotrebljivost), u kojoj meri sistem uspešno raspolaže dostupnim računarskim resursima (efikasnost), koliko je jednostavno održavanje, ažuriranje i izmena sistema, kako se sistem može koristiti na različitim operativnim sistemima (prenosivost) itd. S obzirom na veliki broj potencijalnih osobina koje sistem treba da zadovolji, zadatak rukovodioca projekta jeste da ih rangira po prioritetima u odnosu na zahteve zainteresovanih strana. Džim Brosu (Jim Brosseau) u svojoj knjizi „Software Quality Attributes: Following All the Steps“ iz 2010. godine prikazuje jedan od načina za određivanje relativne skale nefunkcionalnih karakteristika sistema.

RANG		Dostupnost	Bezbednost	Robusnost	Efikasnost	Prenosivost
2	Dostupnost		^	^	<	<
4	Bezbednost			<	<	<
3	Robusnost				<	<
0	Efikasnost					^
1	Prenosivost					

U prikazanoj tabeli po redovima i kolonama se nalaze odabrani nefunkcionalni zahtevi koji se odnose na hipotetički projekat. Kako bi se odredio značaj svakog od njih pristupa se njihovom poređenju u parovima na sledeći način: za svaku ćeliju koja se nalazi u preseku dva zahteva postavljamo pitanje „koji bismo od dva zahteva odabrali kada bismo birali samo između ta dva“. U slučaju da je važniji zahtev koji se nalazi u redu unosimo znak *manje od* „<“, dok u drugom slučaju kada prioritet dajemo zahtevu iz kolone upisujemo *cirkumfleks* ili *kapicu* „^“. Tako, na primer, u navedenoj tabeli, upoređujući robusnost i bezbednost, prednost je data bezbednosti (<), a u slučaju prenosivosti i efikasnosti veća važnost je dodeljena prenosivosti (^). Kolona RANG se popunjava kada se završi sa upisivanjem vrednosti svih parova zahteva tako što se sabira broj cirkumfleksa u koloni sa brojem znakova *manje od* u redu za svaki od zahteva pojedinačno. Na primer, u slučaju zahteva robusnosti, postoji jedan cirkumfleks u koloni i dva *manje od* znaka u redu, dok kod prenosivosti imamo samo jedan cirkumfleks u koloni, što određuje prioritet 3, odnosno 1, za robusnost i prenosivost. Prioritizacija omogućava pružanje ispravnog odgovora u situaciji kada se naiđe na konflikt u implementaciji različitih zahteva.

2.2. KREIRANJE I UPRAVLJANJE ZAHTEVIMA

Procedure za uspostavljanje i održavanje zahteva tokom vremena uključene su u oblast pod nazivom inženjering zahteva. Konkretnije, postoji četiri faze u procesu:

1. Prikupljanje zahteva
2. Analiza zahteva
3. Dokumentovanje zahteva
4. Validacija zahteva

Pre nego što se započne sa prikupljanjem zahteva, neophodno je definisati učesnike u tom procesu. To podrazumeva uključivanje svih ključnih zainteresovanih strana, odnosno, menadžera, zaposlenih, klijenata i dobavljača. U zavisnosti od karakteristika projekta, analitičar ili menadžer projekta definiše klase korisnika koje je potrebno implementirati kao zasebne entitete u softverski sistem. U ove klase ne moraju spadati isključivo ljudski korisnici, već to mogu biti softverski agenti koji obavljaju različite automatizovane operacije, poput botova. Najefikasniji način za identifikovanje korisničkih klasa je da se na samom početku projekta razmisli o zadacima koje će različiti korisnici obavljati sa sistemom. Jedna od metoda jeste pravljenje najšire moguće liste korisničkih klasa na osnovu razgovora sa sponzorom projekta o tome od koga očekuje da koristi sistem. Ova lista zatim prolazi proces sužavanja kroz nalaženje grupa sa sličnim potrebama i njihovim spajanje u jednu klasu. Rezultat ove prve aktivnosti bi trebalo da bude dokumentovanje korisničkih klasa i njihovih karakteristika i odgovornosti u planu zahteva projekta.

Za svaku klasu korisnika potrebno je odrediti konkretnu osobu (predstavnik) koji će učestvovati u celokupnom procesu upravljanja zahtevima, i kasnije, po potrebi u fazi razvoja i testiranja. Predstavnici predstavljaju glavnu spregu između članova jedne korisničke klase i poslovnog analitičara projekta. Njihov zadatak jeste da prikupljaju zahteve od drugih članova korisničkih klasa koje predstavljaju i rešavaju eventualne nejasnoće i konflikte. Iz tog razloga se očekuje da predstavnici korisnika imaju temeljno razumevanje radnog okruženja sa jedne strane, i uloge koju projekat ima u okviru poslovnog sistema organizacije. Najvažnija odgovornost predstavnika jeste da zajedno sa analitičarem precizira opseg i ograničenja projekta, identifikuje potencijalne eksterne sisteme sa kojima treba ostvariti interakciju, proceni uticaj proizvoda na poslovne operacije i identifikuje relevantnu pravnu regulativu i standarde. Neke agilne razvojne metode predviđaju samo jednog predstavnika zainteresovanih strana, odnosno vlasnika proizvoda (eng. *Product owner*).

2.2.1. Prikupljanje zahteva

U najopštijem smislu, postoje dva pristupa prikupljanju i definisanju projektnih zahteva: pristup orijentisan na funkcionalnosti **sistema** i pristup orijentisan na potrebe **korisnika**. Prva kategorija podrazumeva određivanje osobina softverskog sistema u odnosu na funkcionalnosti koje korisnici očekuju od sistema. U ovom slučaju se može javiti nekoliko potencijalnih problema. Najpre, korisnik možda ne zna prave mogućnosti i ograničenja informacionih tehnologija, naročito imajući u vidu njihov brzi napredak. U skladu sa tim, može se desiti da se pojave nerealni zahtevi (npr. zahtevati od sistema da autonomno stvara originalna umetnička dela, koja se mogu uporediti sa ljudskom kreativnošću) ili, da sa druge strane, neke od željenih karakteristika sistema ne budu spomenute od strane korisnika zbog nepostojanja svesti o trenutnim mogućnostima IT tehnologije (npr. klijent nije upoznat sa mogućnošću veštačke inteligencije da optimizuje poslovne procese). Drugi problem koji se može javiti se tiče teškoća redizajniranja načina upravljanja postojećim poslovnim procesima. Korisnici su navikli da stvari rade na određen način i nisu uvek voljni da razmišljaju da li bi se i kako mogla uvesti unapređenja. Zbog ovoga će novi sistem uvek podsećati na neki raniji, pri čemu stvarni doprinos projekta povećanju poslovne vrednosti organizacije može biti minimalan. Treći izazov leži u činjenici da korisnici često opisuju osobine sistema za koje *misle* da žele od novog sistema, a koje se mogu ispostaviti kao suvišne u kontekstu vrednosti koje sistem treba da pruži korisniku. Zbog pomenutih nedostataka, poželjno je da analitičar u ovim početnim fazama razvoja projektnih zahteva umesto „sistemskog“ pristupa koristi koncept koji se zasniva na perspektivi korisnika, odnosno onoga što korisnik želi da ostvari korišćenjem sistema. Drugim rečima, umesto da, prilikom prikupljanja zahteva, korisnika pita šta očekuje da sistem radi, potrebno je da odgovori na pitanje šta korisnik želi da postigne kroz softverski sistem. Cilj je da se opišu koraci koje korisnik treba da izvrši kako bi se dobio izlaz koji ima neku vrednost za korisnika. Na osnovu toga analitičar može da izvuče neophodne funkcionalnosti koje treba implementirati kako bi se ostvarila potrebna interakcija korisnika sa sistemom što će rezultirati ostvarenjem željenog scenarija.

U sklopu procesa prikupljanja informacija o projektnim zahtevima možemo razlikovati „zvanične“ aktivnosti koje se fokusiraju na otkrivanje poslovnih i korisničkih zahteva i koje se sprovode u saradnji sa zainteresovanim stranama, i „nezavisne“ aktivnosti koje podrazumevaju samostalni istraživački rad analitičara (ili menadžera projekta) na pronalaženju informacija sa akcentom na one funkcionalnosti kojih krajnji korisnici možda nisu svesni. Većina projekata koristi kombinaciju oba pristupa.

Jedna od tradicionalnih tehnika prikupljanja informacija je **intervju**. Ljudi na različitim organizacionim nivoima će interpretirati sistem na različite načine, stoga, da bismo dobili sagledavanje problema i na visokom i na niskom nivou, od ključne važnosti je da u proces intervjuu budu uključeni i menadžeri koji nadgledaju postupke koji su od značaja za projekat i zaposleni koji ih sprovode. Uopšteno posmatrajući, postoje dve grupe pitanja koja se mogu postavljati ispitanicima u procesu vođenja intervjuu – pitanja **zatvorenog tipa** i pitanja **otvorenog tipa**. Pitanja zatvorenog tipa zahtevaju konkretan odgovor. Kao primer se mogu navesti zadaci sa više izbora odgovora (eng. *multiple-choice questions*) ili matematički problemi. Kada analitičaru trebaju precizne, detaljne informacije, poželjno je da koristi zatvoreni tip. Tako, umesto da se postavi pitanje „Da li imate mnogo korisnika sistema?“, bolje je pitati „Koliko korisnika sistema imate/očekujete?“.

Pitanja koja pružaju mogućnost ispitaniku da detaljnije objasni svoj odgovor predstavljaju pitanja otvorenog tipa. Njihova svrha jeste da prikupe obimnije informacije, često u esejskoj formi, pri čemu je akcenat na konkretnom ponašanju ili funkcionisanju procesa ili sistema.

Prilikom kreiranja pitanja za konkretnu grupu korisnika poželjno je razmotriti tip informacija za koje je verovatno da ispitanici mogu da ih pruže. Na primer, zaposleni na nižem nivou su verovatno bolje osposobljeni da odgovore na pitanja o svakodnevnim aktivnostima i zadacima u okviru kompanije, dok su menadžeri pogodniji za pitanja o opštoj politici ili organizaciji poslovnih procesa. U većini intervjuu, dobra praksa jeste da se započne sa širokim, opštim pitanjima i postepeno prelazi na ona koja su konkretnija. Na ovaj način analitičar se postupno upoznaje sa problematikom kojom se projekat bavi iz ugla korisnika, što mu omogućava da se kasnije posveti pitanjima koja u početku nisu očigledna i koja bi možda propustio da postavi.

Intervju treba da započne uspostavljanjem dobrog odnosa sa ispitanikom kako bi se osiguralo međusobno poverenje i od ispitanika dobili istini odgovori. Poželjno je da se na početku objasni svrha intervjuu i razlog odabira konkretnog sagovornika, a zatim pređe na planirana pitanja. Važno je da se u toku razgovora pojasni i definiše terminologija kako bi se izbegle situacije da jedan od učesnika stekne utisak da se određene pretpostavke podrazumevaju ili da bi trebalo da budu opšte prihvaćene i jasne. Na kraju, korisno je pružiti mogućnost sagovorniku da pruži dodatne informacije koje on smatra relevantnim, a možda nisu sastavni deo plana intervjuu. Međuljudske veštine su ključne za uspostavljanje odnosa poverenja i efikasnu komunikaciju. Neke osobe razvijaju ove veštine spontano, dok kod drugih to zahteva mnogo truda.

Kao i sve druge, i međuljudske veštine se vremenom mogu naučiti i usavršiti. Po završetku intervjua, neophodno je zapažanja sumirati u strukturirani izveštaj, pri čemu se mora voditi računa da se činjenice dobijene od ispitanika razdvoje od njegovog ličnog mišljenja.

Drugi način prikupljanja informacija jesu **radionice**. Radionice su organizovani sastanci sa više učesnika koji uključuju zainteresovane strane, predstavnike korisnika i delegirane članove razvojnog tima. Najčešće, sastankom predsedava osoba koja ne učestvuje u diskusiji i čiji je zadatak da vodi sastanak u skladu sa unapred definisanom agendom i pravilima. Osim toga, u zapisivanju informacija do kojih se dođe tokom sastanka pomaže jedan ili više beležnika. Prednost radionica leži u mogućnosti prikupljanja podataka o potrebama i očekivanjima više zainteresovanih strana istovremeno i njihovom usklađivanju i rešavanju potencijalnih konflikata. Sastanci mogu da traju nekoliko sati, da se organizuju u toku nekoliko dana ili nekoliko nedelja dok se ne razmotre svi problemi i ne prikupe potrebne informacije. Jedan od nedostataka ove metode je što pati od tradicionalnih problema koji se javljaju prilikom rada u grupama, kada ljudi oklevaju da svoje mišljenje suprotstave mišljenju drugih, naročito nadređenih. Osim toga, najčešće postoji nekoliko osoba koje dominiraju diskusijom i sa druge strane oni koji u njoj ne učestvuju.

Kao i u slučaju intervjua, prilikom planiranja radionice, neophodno je napraviti agendu i sadržaj pitanja o kojima će se raspravljati. Pošto diskusija za cilj ima da podstakne raspravu o nekoj temi, u ovom slučaju pitanja otvorenog tipa predstavljaju primerenije sredstvo za prikupljanje informacija. Vođa sesije ima tri osnovna zadatka: da obezbedi da se prati dnevni red sastanka, da omogućiti da svi učesnici razumeju terminologiju i materiju o kojoj se govori i da sistematizuje rezultate diskusije na način da svi učenici mogu da ih razumeju. Izveštaj sa radionice se sačinjava na osnovu beleški sa sastanaka, vodeći računa da se osnovni zaključci formulišu na jasan i jednostavan način kako bi kasnije mogli da posluže kao osnova za kreiranje zahteva.

Upitnici ili ankete predstavljaju tehniku prikupljanja informacija pogodnu za istovremeno ispitivanje velike grupe korisnika. U najvećem broju slučajeva se kreiraju u formi spiska pitanja namenjenih pojedincima. Učesnici procesa anketiranja su predstavnici različitih grupa korisnika unutar organizacije ili klijenata koji ne pripadaju samoj organizaciji, a na koje se projekat odnosi. Osim što može sadržati pitanja koje se odnose na zahteve projekta, upitnik se može koristiti kao polazna tačka pri kreiranju drugih oblika ispitivanja zaposlenih. Na primer, korišćenjem upitnika se može saznati koje su najkritičnije aktivnosti postojećeg sistema ili koji procesi iziskuju najveći utrošak vremena i novca.

Na osnovu ovih podataka može se preciznije formulisati struktura radionice ili intervjua kako bi se posebna pažnja obratila na problematične aspekte ili prioritete funkcionalnosti projekta. Prilikom pripremanja pitanja za upitnik treba se pridržavati sledećih smernica: koristiti pitanja zatvorenog tipa kako bi se pojednostavila analiza i obrada odgovora; pitanja formulisati na način koji ne sugerise ni jedan odgovor kao tačan; pitanja treba sastaviti tako da su u odgovoru jasno razdvojene činjenice u odnosu na lično mišljenje; broj pitanja ne treba da bude preveliki, jer to može obeshrabriti učesnike da popune upitnik. Najveći izazov prilikom korišćenja upitnika kao tehnike prikupljanja informacija jeste motivisanje učesnika da odgovore na pitanja i pošalju upitnik nazad. Najjednostavniji način da se obezbedi visok procenat potpunjenih upitnika jeste nadzor nadređenog lica i podela ankete na grupnom sastanku koji je namenski organizovan u tu svrhu.

Metoda prikupljanja informacija pod nazivom **analiza korisničkog interfejsa** posmatra postojeće sisteme kako bi identifikovala funkcionalne i korisničke potrebe. U najboljem slučaju, u sklopu procesa se ostvaruje direktna interakcija sa postojećim sistemima, međutim, mogu se koristiti i snimci ekrana i druge pasivne tehnike. Cilj je naučiti što je moguće više o standardnim koracima koje korisnici obavljaju na sistemu u sklopu svakodnevnih poslovnih aktivnosti kako bi se stvorila najbolja slika o potrebnim funkcionalnostima i potencijalnim problemima. U sklopu ove metode, moguće je izvršiti analizu dokumentacije koja se odnosi na postojeće softverske alate koji imaju sličnu namenu koja se očekuje od novog sistema. U dokumentima se često nalaze podaci o industrijskim ili pravnim standardima koji moraju da budu zadovoljeni ili različite polise, procedure i priručnici koji mogu ukazati na dodatne zahteve koje treba razmotriti ili na neke koje je potrebno ukloniti zbog zastarelosti. Izveštaji o propustima ili zahtevi za poboljšanjem koji su prikupljeni od klijenata koji koriste korporativne usluge (preko službe za podršku npr.), takođe mogu pružiti sugestije za unapređenje sistema.

Na završetku faze prikupljanja zahteva, neophodno je organizovati i klasifikovati informacije kako bi se omogućilo kasnija analiza i pravilno dokumentovanje zahteva. Grupisanje se može obaviti u odnosu na tipove zahteva koji su ranije pomenuti ili se može kreirati nova kategorija u slučaju da informacija ne pripada ni jednoj od standardnih klasa. Tako, zahtev može biti poslovni, korisnički, funkcionalni ili nefunkcionalni, ili može pripadati kategoriji „ograničenja“, „ideja“ ili „eksternih zahteva“ itd.

Na primer, u poslovne zahteve spada sve ono što opisuje finansijsku, marketinšku ili reputacionu korist za organizaciju, kao što je: Povećanje prodaje proizvoda X korišćenjem novog automatizovanog sistema odabira reklama. Korisnički zahtevi objašnjavaju konkretne akcije koje *korisnici* treba da preduzmu: Kao korisnik, želim da mogu na jednostavan način da pristupim stanju svog računa i vidim nedavne finansijske transakcije. Funkcionalni zahtevi opisuju ponašanje *sistema* koje za cilj ima omogućavanje potrebnih aktivnosti korisnika: Sistem će poslati korisniku mejl sa potvrdom nakon što izvrši porudžbinu proizvoda. Nefunkcionalni zahtevi se mogu prepoznati po ključnim karakteristikama koje kvalitativno opisuju sistem: Sistem bi trebalo da bude sposoban da obradi 100 korisničkih sesija istovremeno.

2.2.2. Analiza zahteva

Zadatak ove faze jeste da analitičar izvrši razlaganje i analizu procesa koji su definisani kroz prikupljene zahteve kako bi uočio potencijalne nedostatke i predložio korake za unapređenje. U toku prikupljanja zahteva može se desiti da određeni zahtevi budu preskočeni jer ih jedna od strana podrazumeva. Ono što klijent vidi kao očigledno ne mora uvek da bude očigledno članovima projektnog tima. Na primer, klijent podrazumeva da će planirana veb aplikacija da bude prilagođena mobilnim uređajima i različitim rezolucijama ekrana. Međutim, može se desiti da u nedostatku izričitog zahteva u projektnoj dokumentaciji, a zarad uštede na vremenu, razvojni tim zanemari potrebu za korišćenjem „responsive“ tehnologija i da zato finalni proizvod ne zadovolji potrebe korisnika ili da zahteva više vremena i novca da bude završen. Kako bi se umanjila mogućnost pojave nejasnog, neprecizno definisanog ili kasnije ustanovljenog nedostajućeg zahteva mogu se koristiti sledeće metode analize:

- Proveriti da su sve kategorije korisnika učestvovala u fazi prikupljanja zahteva, odnosno, da se svaki zahtev odnosi na bar jednu klasu korisnika koja će imati korist od njega.
- Proveriti da li postoje rupe u graničnim vrednostima zahteva. Na primer, ako zahtev za obračunom cene prilikom naručivanja proizvoda podrazumeva popust od 10% na porudžbine veće od 5000 dinara, proverite da li je u okviru zahteva određeno pravilo u slučaju da je cena porudžbine tačno 5000 dinara.
- Proveriti da svaki korisnički i poslovni zahtev vodi do odgovarajućih funkcionalnih zahteva, kako bi se osiguralo da će sistem sprovesti neophodne operacije u odnosu na aktivnosti korisnika.

- Za model podataka koji će se koristiti u okviru softverskog sistema, utvrditi da li za svaki entitet postoji odgovarajuća funkcionalnost kreiranja, čitanja, izmene i brisanja (eng. *CRUD - Create Read Update Delete*) podataka.

Kako bi se dodatno unapredila implementacija poslovnih procesa obuhvaćenih prikupljenim zahtevima, analitičar može da pristupi **analizi trajanja** svakog pojedinačnog procesa. Postupak započinje utvrđivanjem ukupnog prosečnog vremena potrebnog da se izvrši odgovarajući poslovni proces u celini. Nakon toga, određuje se vreme neophodno za obavljanje pojedinačnih, konkretnih koraka (aktivnosti) koje čine sastavni deo posmatranog procesa. Na kraju, izmerena vremena pojedinačnih koraka se sabiraju i upoređuju sa ukupnim vremenom trajanja procesa. Ukoliko postoji značajna razlika između ukupnog trajanja i zbira trajanja delova procesa, to najčešće ukazuje na potrebu za preispitivanjem načina razlaganja procesa na manje gradivne celine. U ovom slučaju, poslovni proces može da se reorganizuje na način da manje ljudi radi na obradi podataka tako što će se zaposleni osposobiti za obavljanje šireg spektra zaduženja u odnosu na poslovne potrebe. Drugi način je razmatranje mogućnosti paralelizacije koraka u okviru procesa, što podrazumeva izvršavanje više koraka istovremeno. Na osnovu ove analize može se doći do identifikovanja aktivnosti iz poslovnog procesa koje je moguće (i potrebno) eliminisati iz sistema, a da se pri tome ne utiče na kvalitet funkcionalnosti za koju je vezana.

2.2.3. Dokumentovanje zahteva

Kao što je ranije rečeno, analitičar bi trebalo da se usredsredi na dobijanje odgovora na pitanje šta je to što korisnik želi da ostvari korišćenjem softvera. Način na koji je moguće prikazati način interakcije između korisnika i sistema koji dovodi do željenog ishoda su **slučajevi korišćenja** (eng. *use cases*) i **korisničke priče** (eng. *user stories*). Cilj ove tehnike jeste da se prikažu sve aktivnosti koje korisnik može da obavi na sistemu. Korisnički zahtevi su sažeto izraženi u korisničkim pričama (o kojima je ranije bilo reči). Sa druge strane, slučajevi korišćenja ulaze u detalje o tome kako korisnik zamišlja upotrebu sistema za postizanje traženog rezultata, izostavljajući pri tome konkretan dizajn interfejsa. Iz tako formulisanih izjava analitičar može da izvuče neophodne komponente koje razvojni tim treba da implementira u okviru projekta, a tim zadužen za testiranje može da identifikuje potrebne testove koje je potrebno izvršiti kako bi se proverilo da li su zahtevi pravilno implementirani. Slučajevi korišćenja se ne moraju kreirati kada je reč o jednostavnim procesima koji su objašnjeni samom definicijom korisničkog zahteva.

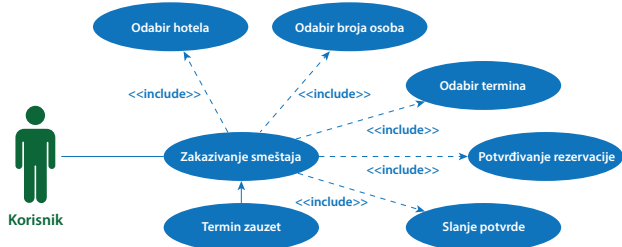
Osnovne komponente slučajeve korišćenja su naziv, opis, akter, okidač, preduslovi, osnovni tok događaja, posledice i izuzeci. Dodatno, slučaju korišćenja se mogu dodeliti prioritet, jedinstveni identifikator, alternativni tokovi, pretpostavke i relacije sa drugim slučajevima korišćenja ili poslovnim procesima. Slučaj korišćenja je u osnovi predstavljen kroz osnovni scenario i jedan ili više alternativnih scenarija. Scenario predstavlja sekvencu aktivnosti koja opisuje jednu instancu moguće interakcije između korisnika i sistema. Na primer, ako pretpostavimo da imamo slučaj korišćenja koji se bavi zakazivanjem noćenja u hotelu putem mobilne aplikacije, osnovni scenario bi mogao biti „korisnik zakazuje termin u hotelu XY“ i njegov sadržaj bi bio sličan sledećem: korisnik otvara početnu stranicu; korisnik klikće na opciju **odaberi smeštaj** u top meniju; korisnik bira željeni hotel, broj osoba i početni i krajnji datum boravka u hotelu; ukoliko je traženi termin slobodan, sistem prikazuje sažetak unetih podataka i cenu rezervacije; korisnik potvrđuje podatke o zakazanom terminu; sistem snima podatke u bazu i šalje e-mejl sa potvrdom o rezervaciji korisniku i menadžeru hotela. Međutim, može se desiti da termin koji je uneo korisnik nije dostupan za rezervaciju, u tom slučaju govorimo o alternativnom scenariju koji bi mogao biti formulisan na sledeći način: sistem prikazuje poruku korisniku da traženi termin nije slobodan; sistem predlaže korisniku tri slobodna termina koja su po početnom datumu najbliža terminu koji je korisnik tražio; sistem traži od korisnika da odabere jedan od tri predložena termina, da unese novi termin ili da odustane od procesa rezervacije; ukoliko korisnik odabere jedan od tri ponuđena termina, postupak rezervacije se nastavlja; ukoliko korisnik unese novi termin, sistem ponovo proverava dostupnost unetog termina. U ovom kontekstu, slučaj korišćenja možemo definisati kao skup scenarija koje povezuje zajednički cilj (zakazivanje noćenja u hotelu). Primer slučaja korišćenja je prikazan u nastavku.

Naziv i prioritet	SK-1: Zakazivanje noćenja u hotelu putem mobilne aplikacije
Prioritet	VISOK
Datum	01.06.2024.
Akter	Kupac koji želi da zakaže boravak u hotelu
Opis	Ovaj slučaj korišćenja uključuje rezervaciju hotelske sobe putem mobilne aplikacije. Korisnik može da obere hotel, tip sobe, broj osoba i željeni termin, unese kontakt podatke i potvrdi rezervaciju.
Okidač	Korisniku je potreban smeštaj i odlučuje da rezerviše hotelsku sobu putem mobilne aplikacije za rezervaciju hotela.
Preduslovi	1. Korisnik ima instaliranu mobilnu aplikaciju 2. Korisnik ima pristup Internetu 3. Korisnik je prijavljen na mobilnu aplikaciju
Osnovni tok	1.0 Korisnik zakazuje termin u hotelu 1. Korisnik otvara početnu stranicu 2. Korisnik klikće na opciju odaberi smeštaj u top meniju 3. Korisnik bira željeni hotel, broj osoba i početni i krajnji datum boravka u hotelu 4. Ukoliko je traženi termin slobodan, sistem prikazuje sažetak unetih podataka i cenu rezervacije (vidi 1.1) 5. Korisnik potvrđuje podatke o zakazanom terminu 6. Sistem snima podatke u bazu i šalje e-mejl sa potvrdom o rezervaciji korisniku i menadžeru hotela
Alternativni tok	1.1 Termin koji je korisnik tražio nije slobodan 1. Sistem prikazuje poruku korisniku da traženi termin nije slobodan 2. Sistem predlaže korisniku tri slobodna termina koja su po početnom datumu najbliža terminu koji je korisnik tražio 3. Sistem traži od korisnika da odabere jedan od tri predložena termina, da unese novi termin ili da odustane od procesa rezervacije 4. Ukoliko korisnik odabere jedan od tri ponuđena termina, postupak rezervacije se nastavlja 5. Ukoliko korisnik unese novi termin, sistem ponovo proverava dostupnost unetog termina
Posledice	1. Hotelska soba je uspešno rezervisana u traženom terminu 2. Korisnik i menadžer hotela primaju potvrdu o rezervaciji 3. Podaci o rezervaciji se snimaju u sistemu
Izuzeci	1.0.11 Odabrana soba je postala nedostupna nakon što ju je korisnik odabrao, a pre nego što je rezervacija potvrđena 1. Sistem obaveštava korisnika da soba više nije dostupna 2. Sistem predlaže korisniku sobu istog tipa i veličine koja je dostupna 3. Ukoliko korisnik prihvati predlog nastavlja se sa osnovnim tokom (vidi 1.0) 4. Ukoliko korisnik ne prihvati predloženu sobu slučaj korišćenja se završava
Relacije	SK-4, SK-6
Pretpostavke	1. Konekcija sa bazom podataka je uspešna 2. Korisnik ima registrovan nalog na mobilnoj aplikaciji

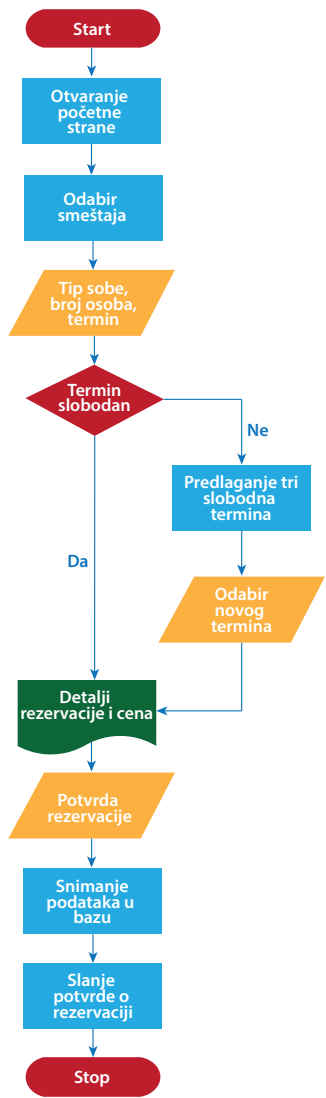
Kao što možemo videti iz navedenog primera, **preduslovi** opisuju stanje u kome se sistem mora nalaziti pre nego što slučaj korišćenja može da započne. Kada **okidač** pokaže nameru korisnika da izvrši slučaj korišćenja, sistem treba da ispita sve preduslove kako bi utvrdio da li slučaj korišćenja može da se pokrene. Proverom preduslova je moguće izbeći greške koje bi mogle nastati ako sistem pokrene tok događaja pre nego što se nalazi u potrebnom stanju da izvrši zahtevane aktivnosti. Na primer, ukoliko korisnik nije povezan na Internet u trenutku kada želi da pribavi podatke o dostupnim hotelima, aplikacija će u najboljem slučaju izbaciti grešku, a u najgorem može doći do nepredviđenog prestanka rada aplikacije. **Posledice** prikazuju stanje sistema nakon uspešno izvršenog slučaja korišćenja. **Osnovni tok** (iako slični po sadržaju, *tok događaja* kao forma obično podrazumeva numerisanu listu koraka, dok se *scenario* piše opisno, u vidu kratke priče) navodi korake koji se izvršavaju kada sve teče glatko u sistemu. **Alternativni tokovi** takođe predstavljaju uspešne instance slučaja korišćenja, koje proizvode isti krajnji rezultat kao i osnovni tok, ali se ređe javljaju u praksi. Najčešće su posledica logičkog grananja koje zavisi od izbora koje akter napravi ili stanja u kome se sistem može naći („termin je zauzet“). Događaji koji dovode do neuspešnog izvršavanja slučaja korišćenja navode se kao **izuzeci**. Izuzeci ne spadaju u standardne instance, već predstavljaju moguće očekivane greške koje sprečavaju dovođenje sistema u željeno stanje (opisano u posledicama). Alternativni tokovi obično ne predstavljaju instance slučaja korišćenja najvišeg prioriteta i ne moraju se implementirati u početnim iteracijama razvoja softvera. Za razliku od toga, izuzetke je neophodno implementirati u najranijoj mogućoj fazi, jer nedostatak njihove obrade može dovesti do pada sistema ili nedefinisanog ili nepredviđenog stanja u kome će se softver naći. Nije jednostavno pretpostaviti šta bi se desilo ukoliko bi sistem rezervisao sobu koja je već zauzeta, jer to u najvećoj meri zavisi od programske logike koja je primenjena prilikom izrade funkcionalnosti rada sa bazom podataka. U najboljem slučaju, došlo bi do dupliranja rezervacije, što bi dovelo do neminovnog nezadovoljstva korisnika i daljih finansijskih i reputacionih gubitaka. Definisanje izuzetaka u fazi prikupljanja zahteva sa korisnicima omogućava isporučivanje robusnih proizvoda (otpornih na greške). Analitičar bi korisnicima za svaki korak u okviru slučaja korišćenja trebalo da postavi pitanje da li postoje neke posebne situacije ili greške koje se mogu javiti i njih navesti kao izuzetak.

UML dijagrami i dijagrami toka (eng. *flowchart*) su dva najčešća modela vizuelnog predstavljanja slučajeva korišćenja. UML dijagramima se predstavljaju interakcije između aktera i sistema. Oni pomažu u identifikaciji i pojašnjenju glavnih funkcija sistema i odnosa između različitih slučajeva korišćenja. UML dijagrami se koriste tokom faze prikupljanja i analize zahteva kako bi se razumele funkcionalnosti sistema iz perspektive korisnika. Primer dijagrama koji se odnosi na prethodno navedeni slučaj korišćenja prikazan je u nastavku.

Primer UML dijagrama



Primer dijagrama toka



UML dijagrami slučajeva korišćenja ne ulaze u detalje samog procesa, odnosno, ne modeluju redosled kojim se koraci izvršavaju. Za razliku od toga, dijagrami toka ilustruju tok događaja korak po korak, prikazujući redosled aktivnosti i tačke logičkog grananja u okviru slučaja korišćenja. Ovakav tip vizualizacije pomaže u procesu raspodele zadataka i koordinisanja timskog rada.

Jedan od najizazovnijih zadataka poslovnog analitičara jeste da u fazi prikupljanja zahteva zainteresovanih strana pravilno identifikuje i opiše slučajeve korišćenja kako bi kasnije mogao pravilno da ih formalno definiše u projektnoj specifikaciji. Proces obično započinje izborom aktera i izradom kratkog opisa slučaja korišćenja. Nakon toga, pristupa se uspostavljanju preduslova i posledica, odnosno granica slučaja korišćenja. Na osnovu prikupljenih podataka, analitičar kroz razgovor sa korisnicima pokušava da izdvoji jedinstvene aktivnosti koje korisnik treba da izvrši kako bi se obavio određeni zadatak. Sekvenca koraka dobijenih na ovaj način predstavlja osnovni tok slučaja korišćenja.

U situacijama kada postoji više slučajeva korišćenja koje mogu da izvrše *različiti* akteri korisno je koristiti „matricu privilegija“ za prikazivanje prava pristupa. Jednostavan primer matrice naveden je nastavku.

MATRICA PRIVILEGIJA	Administrator	Profesor	Asistent	Saradnik
Logovanje na platformu	x	x	x	x
Dodavanje korisnika	x			
Unos ocena na ispitu		x		
Unos bodova		x	x	x
Pregled podataka o studentima	x	x	x	
Unos održanih časova	x	x	x	x
Aktiviranje el. testova		x	x	

Važno je naglasiti da slučajevi korišćenja opisuju interakciju sa sistemom iz perspektive *korisnika* i ne sadrže sve informacije koje su potrebne razvojnom timu za pisanje softvera. Postoje funkcionalni zahtevi koji spadaju izvan opsega dijaloga između aktera i sistema, npr. „svaka rezervacija mora da ima dodeljen jedinstveni identifikator“. Drugi funkcionalni zahtevi se ne uklapaju u standardni opis slučaja korišćenja, koji podrazumevano ne sadrži podatke o tome šta sistem treba da uradi ako, recimo, neki od preduslova nije zadovoljen. Iz ovog razloga je neophodno da ove dodatne informacije analitičar prenese razvojnom timu

(naročito programerima i testerima) kako kasnije ne bi došlo do neusaglašenosti između ponašanja softvera i onoga što klijent očekuje. Jedan od mogućih načina da se ovo postigne jeste da se funkcionalni zahtevi, ukoliko nisu očigledni, navedu uz svaku specifikaciju slučaja korišćenja. Osnovni izazov u ovom pristupu leži u činjenici da više slučajeva može zahtevati isti funkcionalni zahtev (npr. pomenuto dodeljivanje jedinstvenog identifikatora). U ovoj situaciji, analitičar ne mora da duplira tekst funkcionalnog zahteva, već može da ga navede samo jednom u specifikaciji, a onda da ga referencira na svakom mestu gde je to potrebno. Drugi način jeste navođenje funkcionalnih zahteva odvojeno od specifikacije slučajeva korišćenja, u vidu posebne sekcije u okviru dokumentacije. Ovakav pristup takođe podrazumeva kreiranje veze između slučaja i odgovarajućih funkcionalnih zahteva putem navođenja jedinstvenih identifikatora zahteva u specifikaciji slučaja korišćenja.

Povezivanje slučajeva korišćenja sa funkcionalnim zahtevima pomaže u procesu određivanja prioriteta zahteva. Prioritizacija je od posebnog značaja za razvojni tim, jer ukazuje na redosled kojim će se pristupiti implementaciji funkcionalnosti sistema. Slučajevi korišćenja su visokog prioriteta ukoliko opisuju osnovni poslovni proces organizacije, ukoliko se često koriste, ako su neophodni radi usklađenosti sa pravnom regulativom ili industrijskim standardima ili ako postoje druge funkcionalnosti sistema koje od njega zavise. Samim tim, funkcionalni zahtevi koji su povezani sa ovakvim slučajem dobijaju visok prioritet u razvojnem procesu.

Završni rezultat procesa definisanja slučajeva korišćenja, zajedno sa ranije pomenutom vizijom i opsegom projekta jeste dokument koji oslikava dogovor između zainteresovanih strana kako treba da izgleda proizvod koji je potrebno isporučiti. Za razliku od toga, funkcionalni i nefunkcionalni zahtevi se dodatno opisuju u okviru dokumenta koji se najčešće naziva Specifikacija zahteva softvera ili kraće, Softverska specifikacija i koji je namenjen članovima razvojnog tima. Ova dva dokumenta predstavljaju osnovu za uspešno upravljanje softverskim projektom.

Softverska specifikacija opisuje karakteristike, funkcionalnosti i ograničenja softverskog sistema. Osim toga, trebalo bi da detaljno opisuje predviđene nefunkcionalne karakteristike sistema, uključujući upotrebljivost, bezbednost i performanse. Dokument primarno služi kao osnova za implementaciju, testiranje i korisničku dokumentaciju, ali predstavlja i polaznu tačku za menadžera projekta prilikom određivanja vremena, cene i potrebnih resursa. Iako ne postoji propisana forma specifikacije zahteva, međunarodni standard ISO/IEC 29148:2018³ pruža

3 <https://www.iso.org/standard/72089.html>

dobre smernice na koji način treba organizovati informacije u okviru pomenutog dokumenta. Standard nosi naziv „Sistemske i softverske inženjering – Procesi životnog ciklusa – Inženjering zahteva“ i u osnovi opisuje neophodne procese u inženjerskim aktivnostima u okviru softverskih proizvoda i usluga tokom njihovog životnog ciklusa.

Pored ostalog, veliki deo teksta se odnosi na poželjan sadržaj softverskih zahteva i preporučuje format u kome se zahtevi mogu predstaviti. U nastavku je prikazan primer sadržaja dokumenta Specifikacija zahteva softvera.

1. Uvod

- 1.1. Svrha dokumenta
- 1.2. Opseg dokumenta
- 1.3. Korišćene konvencije

2. Funkcionalnosti Sistema

- 2.1. Funkcionalnost X
 - 2.1.1. Opis
 - 2.1.2. Funkcionalni zahtevi
- 2.2. Funkcionalnost Y
 - 2.2.1. Opis
 - 2.2.2. Funkcionalni zahtevi
- ...

3. Spoljni interfejsi

- 3.1. Korisnički interfejsi
- 3.2. Softverski interfejsi
- 3.3. Komunikacioni interfejsi
- 3.4. Dizajnerska ograničenja

4. Sistemske zahteve

- 4.1. Upotrebljivost
- 4.2. Dostupnost
- 4.3. Prenosivost
- 4.4. Bezbednost
- 4.5. Performanse
- ...

5. Industrijski standardi, poslovna politika i pravna regulativa

- 5.1. Upravljanje informacijama
- ...

U opisu svrhe dokumenta potrebno je navesti naziv proizvoda na koji se zahtevi iz dokumenta odnose i oznaku verzije sistema, a zatim nabrojati kategorije korisnika kojima je dokument namenjen, npr. programeri, grafički dizajneri, projekt menadžeri, testeri, kreatori dokumentacije. U okviru sekcije koja se odnosi na opseg dokumenta treba pružiti kratak opis softvera i njegovu svrhu, navođenjem poslovnih ciljeva i vrednosti koju donosi organizaciji. Ukoliko su vizija, opseg i ciljevi sadržani u posebnom dokumentu, potrebno je referencirati se na njega. Opis korisničkih interfejsa najčešće sadrži reference na fontove, ikonice, stil dugmića ili šeme boja, zatim, ograničenja u smislu veličine ekrana i rezolucije, kao i konvencije za prikazivanje dijaloga sa porukama i pravila za validaciju podataka. Softverski interfejsi identifikuju i opisuju veze između proizvoda i različitih softverskih komponenti, uključujući druge aplikacije, baze podataka, operativni sistem i eksterne resurse. U ovoj sekciji se opisuje svrha, format i sadržaj poruka koje se razmenjuju između ovih komponenti i mapiraju se ulazni i izlazni podaci koji se koriste u komunikaciji. U okviru komunikacionih interfejsa se navode zahtevi za primenom odgovarajućih mrežnih protokola, modela enkripcije podataka, sinhronizacijskih mehanizama ili formata i sadržaja elektronskih poruka koje proizvod generiše ili koristi. Nefunkcionalni zahtevi predstavljaju karakteristike koje sistem mora da poseduje, a koje nisu direktno povezane sa specifičnim funkcijama sistema.

2.2.4. Validacija zahteva

Validacija zahteva je poslednja faza procesa upravljanja zahtevima. Ovaj proces predstavlja proveru da li proizvod zadovoljava uslove koji su dogovoreni sa klijentom (za razliku od verifikacije koja određuje da li proizvod to radi na pravilan način). Proverom usaglašenosti dokumentovanih zahteva za željama klijenta se potvrđuje da zahtevi na pravi način opisuju željene mogućnosti i osobine sistema, da su zahtevi pravilno definisani na osnovu poslovnih pravila, da ne postoje suvišni (nepotrebni) zahtevi i da se na osnovu njih može pristupiti procesu implementacije projektnog rešenja. Aktivnosti potrebne za obavljanje provere se mogu sprovesti kroz formalni postupak **inspekcije**.

Formalni proces inspekcije (početno razvijen u IBM-u 70-ih godina dvadesetog veka i kasnije unapređivan) predstavlja strogo strukturiran proces koji se sastoji iz nekoliko faza. Uključuje mali tim učesnika koji ispituju kvalitet proizvoda, koji može predstavljati rezultat bilo koje faze razvojnog ciklusa projekta, npr. softverski zahtevi, izvorni kod, dokumentacija i sl. Obavezni članovi inspeksijskog tima uključuju autora proizvoda (u našem slučaju proizvod je specifikacija softverskih

zahteva, a autor je analitičar), osobe koje su bile izvor informacija za kreiranje proizvoda (predstavnici zainteresovanih strana), osobe čiji rad će se bazirati na proizvodu (članovi programerskog tima) i ljudi koji su odgovorni za interakciju proizvoda sa spoljašnjim sistemima. Prva faza inspekcije predstavlja „planiranje“. Ova aktivnost podrazumeva određivanje učesnika, pripremu materijala, procenu vremena trajanja inspekcije i određivanje termina kada će sastanak biti održan. Treba primetiti da što se više vremena odvoji analizi svakog zahteva, veći je broj nedostataka koji će se pronaći. Druga faza procesa inspekcije jeste „priprema“. Nekoliko dana pre zakazanog početka sastanka, učesnicima se prosleđuje dokument na koji se inspekcija odnosi, sa ciljem da svaki učesnik ispita dokument i samostalno identifikuje potencijalne nedostatke. Ovom vrstom individualne provere se može otkriti do $\frac{3}{4}$ postojećih nedostataka. Za vreme „inspekcijskog sastanka“ nedostaci koje su učesnici pronašli u toku pripreme se beleže i dopunjuju novim problemima koji su zajednički identifikovani u toku samog sastanka. Po završetku rada na celokupnom dokumentu, tim donosi odluku da li će prihvatiti dokument u trenutnom obliku, prihvatiti ga nakon unošenja manjih izmena ili zahtevati veće izmene dokumenta. U slučaju da postoji potreba za izmenama, autor pristupa fazi „prerade“ specifikacije. Poslednji korak inspekcijskog procesa jeste provera da li su sva otvorena pitanja rešena i greške ispravljene. Ukoliko je to slučaj, proces inspekcije se zatvara, u suprotnom, dokument se vraća na dodatnu „preradu“ od strane analitičara.

Pored formalnog procesa inspekcije koji se koristi za validaciju softverske specifikacije, naveli bismo još jednu široko rasprostranjenu metodu koja se može koristiti u ovu svrhu, naročito kada je reč o projektima nižeg prioriteta. U pitanju je tehnika pod nazivom **nezavisna recenzija**. Rezultat prikupljanja neformalnih komentara na specifikaciju softverskih zahteva nisu temeljne i strukturne informacije, već predstavlja ekspertsko mišljenje kolege ili saradnika u pogledu kvaliteta dokumenta. Identifikovani nedostaci se obeležavaju u dokumentu i ukratko obrazlažu.

2.2.4.1. Rešavanje konflikata

Konflikti mogu da nastanu u svim fazama upravljanja zahtevima, međutim, mi ćemo se procesom rešavanja konflikata pozabaviti u okviru faze validacije, jer je ona poslednji stadijum kada njihovo identifikovanje može biti obavljeno uz minimalne finansijske i vremenske posledice po razvoj projekta (za više detalja pogledati „razmeru 5:10:200“ u sekciji „Testiranje softverskog projekta“). Sistematsko upravljanje konfliktima podrazumeva **identifikovanje konflikta, analizu konflikta, rešavanje konflikta i dokumentovanje rešenja konflikta**.

Obzirom da se konflikti između zahteva zainteresovanih strana mogu javiti u toku trajanja svih aktivnosti razvoja zahteva, neophodno je da analitičar obrati posebnu pažnju na karakteristične pokazatelje potencijalnih problema. Uzmimo za primer da jedna zainteresovana strana (npr. član visokog menadžmenta) u fazi prikupljanja zahteva iskaže zahtev na sledeći način: „vreme odgovora sistema mora da bude kraće od 5 sekundi“, ovo iz njenog ugla može predstavljati prihvatljiv (kratak) vremenski period. Međutim, iz ugla tehničkog lica koje će koristiti softver i koje je suočeno sa stotinama zahteva koje prosleđuje sistemu na obradu, pet sekundi može biti neprihvatljivo dugačak period čekanja. Druga vrsta konflikata može nastati usled sukoba interesa zainteresovanih strana koje imaju različite prioritete. Dizajnerski tim može zahtevati jednostavan i intuitivan interfejs radi boljeg korisničkog iskustva, dok menadžerski tim može tražiti da na ekranu bude prikazano što više funkcija kako bi se naglasile nove sposobnosti sistema. Drugi primer jeste sukob između tehničkog tima koji se zalaže za skalabilnu arhitekturu koja se lako može prilagoditi budućem potencijalnom rastu i promenama u funkcionalnostima, što zahteva duže vreme razvoja inicijalnog proizvoda, a sa druge strane se može naći menadžerski tim koji prioritizuje brzo izbacivanje proizvoda na tržište radi ostvarivanja konkurentске prednosti. Na kraju, konflikt se može javiti usled netrpeljivosti ili nedostatka poštovanja ili komunikacije između različitih zainteresovanih strana.

Najčešće, rešavanje konflikata spada u nadležnosti menadžera projekta. U ovom procesu menadžer projekta može primeniti različite strategije, kao što su **sporazumni dogovor, kompromis, glasanje i nadjačavanje**. Sporazumni dogovor podrazumeva da se sve strane u konfliktu, kroz razmenu mišljenja i sučeljavanje stavova, saglase da prihvate jedno od predloženih rešenja. Kompromis se razlikuje od sporazumnog dogovora po tome što prihvaćeno rešenje sadrži mešavinu sukobljenih zahteva koje su ponudile različite zainteresovane strane, odnosno, sva inicijalno predložena rešenja su odbijena i napravljeno je novo (kompromisno) rešenje koje je odobreno od svih. U slučaju da sukobljene strane ne mogu da dođu do dogovora sporazumno ili putem kompromisa, može se primeniti tehnika glasanja. Sve zainteresovane strane (ne samo one koje su u konfliktu) nakon diskusije glasaju za jedno od dva ili više predloženih rešenja i rešenje koje dobije najviše glasova se usvaja. Ukoliko ni jedna od navedenih strategija ne donese rezultate (recimo, broj glasova je izjednačen), menadžer projekta može doneti odluku da primeni metod nadjačavanja. U ovoj situaciji, rešenje koje predloži zainteresovana strana koja je viša u hijerarhijskog strukturi klijentske organizacije se prihvata kao rezultat rešavanja konflikta.

MODELI RAZVOJA SOFTVERSKOG PROJEKTA

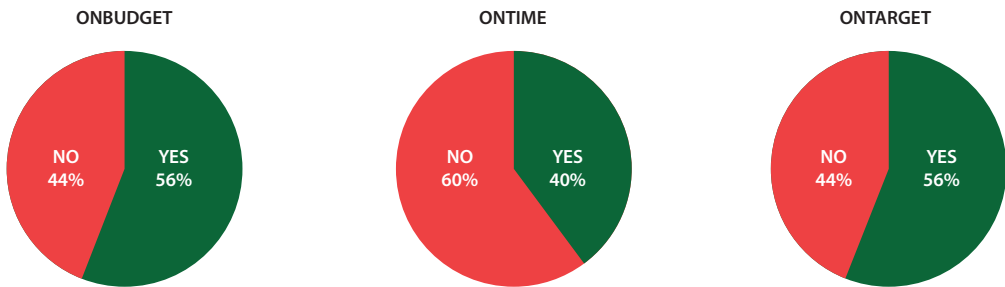


3. MODELI RAZVOJA SOFTVERSKOG PROJEKTA

Ovom prilikom se nećemo detaljno baviti mnogobrojnim modelima koji se koriste u procesu razvoja softvera. Umesto toga, prikazaćemo osnovne pristupe upravljanju softverskim projektom i objasniti najznačajnije karakteristike svakog od njih.

Da bismo prikazali značaj odabira prave strategije iskoristićemo podatke organizacije „Standish Group“⁴ koji analiziraju različite aspekte koji utiču na uspešnost razvoja softverskih projekata. Standish Group predstavlja istraživačku i savetodavnu organizaciju čiji je fokus na praćenju performansi razvoja softvera. U svom izveštaju pod nazivom „CHAOS Report 2015“ predstavili su statističke podatke koji se odnose na uspešnost razvoja softverskih projekata u periodu od 5 godina (2011-2015.).

Uspeh softverskog projekta definisan je kroz tri osnovna atributa „OnTime“ (na vreme), „OnBudget“ (u okviru budžeta) i „OnTarget“ (u skladu sa definisanim ciljem) koji odgovaraju primarnim ograničenjima projekta o kojima je ranije bilo reči.



Na prikazanim pojedinačnim grafikonima možemo videti procenete projekata koji su završeni u okviru planiranog budžeta, zatim, u toku predviđenog vremena, i na kraju, u skladu sa definisanom specifikacijom. Može se zaključiti da svega nešto više od polovine završenih projekata poseduje sve tražene funkcionalnosti i isto toliko potpada pod dogovorenu cenu, a čak 60% projekata ne uspe da se završi na vreme. Osim toga, ako se govori o stepenu zadovoljstva, korisnici izražavaju određeni stav koji nije nužno u skladu sa činjenicom da li su sve funkcionalnosti i mogućnosti koje su tražili na početku projekta ispunjene.

4 <https://www.standishgroup.com/>

Pomenuto istraživanje je otkrilo da kada se softver isporuči sa mnogo manje funkcionalnosti od originalno predviđenih, odnosno kada su ispunjeni samo osnovni zahtevi, postoji veći nivo zadovoljstva. Po dodatnim analizama, većina implementiranih mogućnosti softvera se ne koristi. Ove dodatne funkcije ne donose uvek vrednost, već umesto toga povećavaju rizik, troškove i utiču na kvalitet.

Kao najvažniji faktor koji utiče na ishod projekta izdvojena je veličina projekta. Tabela zavisnosti uspešnosti projekta od njegove veličine prikazana je u nastavku.

Size	SUCCESSFUL	CHALLENGED	FAILED	TOTAL
Grand	6%	51%	43%	100%
Large	11%	59%	30%	100%
Medium	12%	62%	26%	100%
Moderate	24%	64%	12%	100%
Small	61%	32%	7%	100%

Iz navedene tabele može se zaključiti da je veličina projekta obrnuto srazmerna stepenu uspešnosti, odnosno, veći projekti imaju manju verovatnoću da budu uspešno završeni. Dok je u slučaju najmanjih projekata uspešnost oko 60%, kada je reč o ogromnim poduhvatima, ona jedva da prelazi 5%. Ovi podaci ukazuju na potrebu razbijanja velikih softverskih projekata na više manjih projekata, što bi omogućilo brže isporuke i ostvarivanje vrednosti i dovelo do većeg zadovoljstva korisnika. U najvećem broju slučajeva veliki i kompleksni, višegodišnji projekti su nepotrebni.

Da bismo predstavili zaključke CHAOS izveštaja na koji način odabir modela razvoja softvera utiče na njegov uspešan završetak, najpre je neophodno da se upoznamo sa različitim mogućim pristupima razvoju projekta, što će biti cilj narednih poglavlja.

3.1. TPM, APM, XPM, MPX

Robert K. Wysocki u svojoj knjizi „Effective Project Management - Traditional, Agile, Extreme“ koristi dve osnovne promenljive – **cilj** i **rešenje**, koje poseduje svaki projekat kako bi izvršio njihovu kategorizaciju. Svaka od ovih promenljivih može imati samo dve vrednosti – **jasno** i **nije jasno**. Na ovaj način dobijamo matricu sa četiri kvadranta koji predstavljaju četiri najopštija tipa projekata:

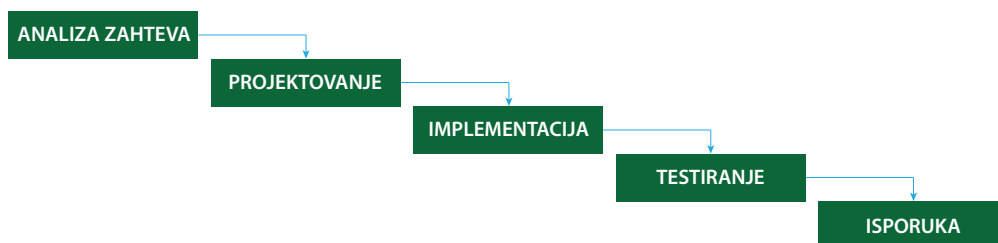
- Kada su cilj i rešenje jasni, projekat spada u kategoriju tradicionalnih projekata, odnosno Traditional Project Management (TPM).
- Kada je cilj jasan, ali rešenje nije, projekat se klasifikuje kao agilni projekat, odnosno, primenjuje se Agile Project Management (APM) strategija.
- Kada nisu ni cilj ni rešenje jasni, generiše se kategorija Extreme Project Management (xPM).
- I na kraju, kada cilj nije jasan, ali rešenje jeste, projekat se svrstava u Emertxe Project Management (MPx).

		Rešenje	
		Jasno	Nije jasno
Cilj	Nije jasan	MPx Emertxe (e-murt-see)	xPM Extreme
	Jasan	TPM Traditional	APM Agile

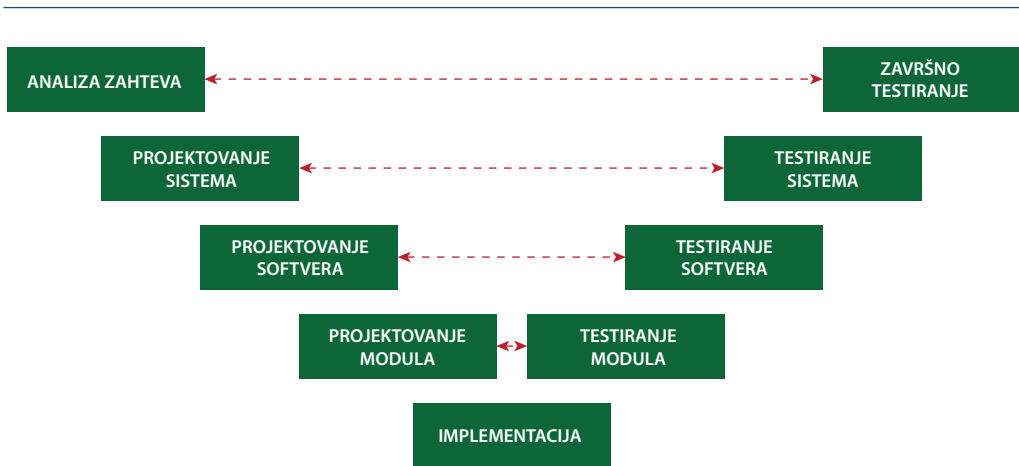
3.1.1. Tradicionalni pristup upravljanja projektom (TPM)

Tradicionalni pristup upravljanja predstavlja najjednostavniju moguću kategoriju u koju spadaju projekti čiji su cilj i rešenje poznati. U današnje vreme se sve ređe sreću, jer obuhvataju projekte koji su već više puta završeni na jedan ili drugi način. Pošto je gotovo uvek moguće preuzeti neko od postojećih rešenja i dizajn, malo stvari ostaje nepoznato ili neočekivano. Ne očekuju se promene u odnosu na početnu specifikaciju koja je dogovorena sa klijentom, pa je i sam model upravljanja takvim projektima neprilagođen mogućim izmenama u funkcionalnostima. Njihov glavni naglasak leži na ispunjenju rokova i ograničenjima budžeta, sa manje naglasaka na stvaranju nove vrednosti za kompaniju. Broj projekata koji su uspešno završeni putem kvadranta TPM postepeno opada zbog okolnosti u kojima živimo. Potpuno nov skup pristupa koji su usmereni na klijenta i generišu komercijalnu vrednost umesto tesnog pridržavanja rasporeda i budžeta zamenjuje TPM pristupe, koji postaju manje uobičajeni.

Osnovne osobine ovih projekata su mala složenost i nizak nivo rizika, poznate tehnologije. Često se karakteriše strukturisanim i linearnim procesom sa jasno definisanim fazama, poput inicijalizacije, planiranja, izvršenja, praćenja i kontrole, i zatvaranja. U ovom pristupu, detaljan projektni plan se kreira na početku projekta. Plan obuhvata zadatke, vremenske okvire, rezervisanje resursa. Promene u planu su minimalne nakon što je plan uspostavljen. Tradicionalni pristup često prati vodopadni model razvoja projekta, gde napredak teče u jednom smeru - naniže kroz faze, slično vodopadu. Svaka faza se završava pre prelaska na sledeću.



„V“ Model, odnosno model validacije i verifikacije, predstavlja proširenje konvencionalnog vodopadnog modela. Naziv je dobio po obliku slova "V" koji formira. Sastoji se od faza razvoja koje su predstavljene sa leve i faza testiranja koje se nalaze na desnoj strani slova "V". Model poseduje definisane povratne sprege između odgovarajućih faza razvoja i faza testiranja koje u slučaju problema u nekoj od faza testiranja omogućavaju ponavljanje prethodnih faza, počev od faze na koju je sprega odnosi. Prilikom izmene u nekoj faza neophodno je ažurirati sve naredne faze i odgovarajuću dokumentaciju.



Na primer, ukoliko se otkrije greška u fazi testiranja softvera ponavljaju se sve faze počev od faze projektovanja softvera. Karakteristika ovog pristupa jeste da daje kvalitetan proizvod (zbog velikog broja iteracija kroz koje razvoj prolazi), na uštrb vremena i novca.

3.1.2. Agilni pristup upravljanja projektom (APM)

U situacijama kada je jasno definisano šta je potrebno uraditi, ali nije očigledno kako postići taj cilj, primenjuje se agilni pristup upravljanja (softverskim) projektom. Neki od osnovnih principa agilne metodologije su prilagodljivost, timski rad i inkrementalni napredak. Ova metodologija je veoma korisna u projektima razvoja softvera gde postoji puno neizvesnosti ili gde se zahtevi često menjaju. Očigledno je da tradicionalna metodologija ne može da funkcioniše u slučajevima kada je rešenje nepoznato, jer ova činjenica onemogućava razvijanje detaljnog plan razvoja koji zahteva tradicionalni pristup.

Agilno upravljanje softverskim projektima stavlja snažan naglasak na korišćenje malih, samostalnih timova koji blisko sarađuju međusobno, kao i sa ostalim zainteresovanim stranama kako bi se osiguralo da njihov rad zadovoljava očekivanja klijenta. Ključna komponenta agilnog upravljanja softverskim projektima su kratki razvojni ciklusi. Tokom ovih ciklusa timovi se koncentrišu na stvaranje malih, postepenih poboljšanja softvera, što omogućava brzo testiranje i prilagođavanje. Osim toga, na ovaj način je garantovana pravovremena integracija povratnih informacija u proces razvoja.

Agilne metodologije podrazumevaju pristup prilagođen promenama, umesto pristupa prilagođenog planu, kao što je slučaj sa tradicionalnim modelima. Nivo rizika u ovim slučajevima je znatno veći, jer postoji šansa da prihvatljivo rešenje zadatog cilja ne postoji ili je teško dostižno. Ovome doprinosi i činjenica da je u ovakvim okolnostima predviđanje troškova znatno otežano, što može dovesti do gubitaka čak i slučajevima uspešnog okončanja projekta. Da bi se umanjili rizici ključno je da razvojni tim tesno sarađuje sa klijentom i da postoji deljena odgovornost za uspešnost projekta. U najčešće korišćene agilne modele razvoja spadaju SCRUM, Lean i Kanban.

3.1.2.1. *Lean pristup*

Lean predstavlja agilnu metodologiju, razvijenu sredinom prošlog veka od strane Tojote, koja je primarno namenjena oblasti proizvodnje, sa fokusom na povećanje vrednosti kroz smanjenje otpada. Polazna osnova za njenu primenu predstavlja promena u načinu razumevanja tržišta, odnosno prelaska sa tradicionalnog modela mišljenja u kome organizacija određuje CENU svog proizvoda, koja se bazira na jednačini $TROŠKOVI + \text{ŽELJENI PROFIT} = CENA$, na novo shvatanje tržišnog okruženja u kome je *kupac* (klijent) onaj koji diktira CENU, a time i PROFIT. Drugim rečima, u modernom tržišnom okruženju, način da se poveća profit nije kroz podizanje cene proizvoda, već kroz smanjenje troškova proizvodnje. Osnovni cilj jeste obezbediti najbolji kvalitet proizvoda, po najnižoj ceni, za najkraće moguće vreme, eliminišući otpad.

U kontekstu Lean metodologije „otpad“ predstavlja sve što ne doprinosi vrednosti proizvoda, pri čemu je vrednost određena kao ono za šta je kupac spreman da plati. Na primer, ako neka komponenta stoji na polici neiskorišćena ili ako proizvodni pogon napravi više proizvoda nego što je trenutno potrebno, to predstavlja otpad. U smislu fizičkog rada, premeštanje proizvoda unutar fabrike je otpad. Shodno tome, ako softverski projekat sadrži više funkcionalnosti nego što je potrebno u datom trenutku ili ako se razvoj premešta iz jednog sektora

u drugi, to predstavlja otpad. Sve što sprečava brzo zadovoljenje konkretnih, preciznih zahteva kupaca jeste otpad. Sedam tipova proizvodnog otpada iz originalne Lean metodologije se mogu prevesti u domen razvoja softverskog proizvoda tako da se definišu sledeći tipovi otpada: delimično završen posao, dodatni procesi, dodatne funkcionalnosti, promena zaduženja, čekanje, kretanje i nedostaci. Eliminisanje otpada predstavlja prvi od sedam⁵ Lean principa u agilnom pristupu razvoja softvera. **Nedovršen posao** se odnosi na delove koda koji nisu integrisani u produkciono okruženje ili nisu testirani, odnosno, nisu isporučeni klijentu. Osim što ne doprinosi vrednosti projekta, ova vrsta otpada odlaže dobijanje povratne informacije od strane klijenta, povećava složenost projekta, a samim tim i verovatnoću javljanja grešaka, koristi resurse koji se mogu iskoristiti za povećanje vrednosti i uvećava šansu da kod zastari ili postane nepotreban. **Dodatni procesi** predstavljaju aktivnosti koje troše resurse i usložnjavaju razvoj, a ne doprinose kvalitetu i funkcionalnosti završnog proizvoda. Pisanje i održavanje previše detaljne dokumentacije ili organizovanje čestih sastanaka na kojima se odobrava ili pregleda svaka mala izmena u kodu, predstavljaju primere otpada koji dodaju nepotrebne korake u razvojni ciklus, oduzimaju resurse i povećavaju mogućnost grešaka. Svaki aspekt softvera koji nije neophodan da bi se zadovoljili najaktuelniji zahtevi klijenta, spada u **dodatnu funkcionalnost** koja predstavlja otpad. Često se može videti da programeri dodaju delove koda za slučaj da nekad zatrebaju ili iz radoznalosti da provere kako određena funkcija radi. Iako ovo može delovati bezopasno, svaki deo koda povećava složenost, jer mora da se prati, integriše, testira i održava tokom celog životnog ciklusa proizvoda. Kada govorimo o Lean pristupu u razvoju softvera, **promena zaduženja** se odnosi na često premeštanje članova tima sa jednog na drugi zadatak ili sa jednog na drugi projekat. Iako se čini da „multi-tasking“ može da poveća produktivnost, on zapravo smanjuje efikasnost i time predstavlja otpad. Najbrži način da se završe dva projekta koja koriste iste resurse jeste da se završe jedan za drugim. Prelazak sa jednog posla na drugi podrazumeva određeni psihički napor. Potrebno je zapamtiti gde se stalo na prethodnom poslu i fokusirati se na nove parametre preuzetog posla, što dovodi do grešaka ili previđanja detalja i povećava stres i zamor. **Čekanje** kao vrsta otpada se odnosi na bilo koje vreme kada posao ne napreduje, odnosno, kada je rad odložen. Čekanje da drugi tim završi njihov deo posla, čekanje na odobrenje ili odluke zainteresovanih strana, čekanje da se oslobode potrebni resursi, odlaganja prouzrokovana čekanjem na tehničku podršku ili razjašnjenje zahteva, predstavljaju primere otpada. Kada se pojavi kritičan zahtev od strane klijenta, brzina kojom će se na njega odgovoriti je direktno povezana sa čekanjima u razvojnem ciklusu koja sprečavaju klijenta da realizuje vrednost u najbržem roku. **Kretanje** predstavlja

5 M. Poppendieck and T. Poppendieck, Lean Software Development: An Agile Toolkit. 2003.

nepotrebne aktivne radnje ljudi koje ne dodaju vrednost proizvodu. Najbolji primeri su stalni prelazak sa jednog alata ili okruženja na drugo, kretanje kroz neefikasno organizovan fizički radni prostor, ručno isporučivanje nadogradnji sistema i sl. Ovo je razlog zašto se u slučaju agilnog razvoja predlaže da tim radi u istoj prostoriji gde je komunikacija i pristup informacijama olakšan. Osim toga, premeštanje dokumentacije ili baze podataka od jednog do drugog člana tima (npr. od programera do testera ili dizajnera) takođe predstavlja kretanje. Uticaj otpada koji se odnosi na **nedostatke** je srazmeran uticaju samog nedostatka na projekat i vremenu proteklom do njegovog otkrivanja. Mnogo veći problem predstavlja manji nedostatak koji je neotkriven više nedelja, nego kritičan koji je rešen za nekoliko sati. Dakle, način da se umanje ova vrsta otpada jeste često testiranje, integracija i brzo puštanje u produkciju.

U najvećem broju slučajeva, uključivanje predstavnika klijenta i korisnika koji najbolje razumeju problem koji projekat treba da reši, predstavlja ključnu pretpostavku za uspešno okončanje projekta. Česte iteracije tokom kojih se odvija unapređenje dizajna sistema predstavlja jedan od najefikasnijih načina za generisanje znanja, rano otkrivanje rešenja i izgradnju sistema sa integritetom. Ako se ima u vidu da tokom razvoja softvera mnogo stvari može poći u nepredviđenom pravcu, pribavljanje povratnih informacija od korisnika jeste najbolji način kako se uticaj nepredviđenih okolnosti može ublažiti. Akcenat treba staviti na ispunjenju zahteva klijenata, umesto na poštovanju zacrtanih rokova. Ovo se postiže isporučivanjem prototipova funkcionalnosti u okvirno definisanim vremenskim terminima i pribavljanjem mišljenja od strane klijenta. Dakle, *iteracija* predstavlja inkrement (dodatak) softvera koji se implementira, testira i isporučuje u kratkim periodima, čime se obezbeđuje rano otkrivanje problema i njihovo rešavanje što umanjuje rizik po uspešnost projekta. Inkrement treba da sadrži smislen, logički zaokružen skup funkcionalnosti koji doprinosi krajnjem cilju projekta, ali je istovremeno dovoljno mali da omogućava preciznu procenu truda koji je potrebno uložiti u njegovu realizaciju. Dobra strategija je da se najprioritetniji zahtevi rešavaju na početku kako bi se što pre ostvarila poslovna vrednost za klijenta. Prilikom prikupljanja projektnih zahteva, najpre se od klijenta traži da navede najvažnije funkcije koje sistem treba da obavlja, te funkcije se što je pre moguće implementiraju i isporučuju klijentu, a tek nakon toga se od klijenta zahteva da definiše funkcije nižeg prioriteta i tako dok se zahtevi svih nivoa važnosti ne prikupe.

Lean pristup razvoju softvera predlaže **odlaganje donošenja odluka** što je duže moguće, jer je lakše promeniti odluku koja još nije doneta. Na ovaj način se naglašava potreba za razvijanjem robusnog i prilagodljivog dizajna sistema koji ukazuje na neizbežnost promena i koji je sposoban na brzo prilagođavanje. Na početku razvojnog ciklusa neophodno je doneti kritične odluke u pogledu arhitekture sistema koje će odrediti njegova primarna ograničenja.

U te odluke spadaju: odabir strukture softverskog sistema - mikro servisi ili monolitna struktura, izbor tipa baze podataka (SQL ili NoSql), odabir alata za upravljanje verzijama, izbor bezbednosnih okvira i protokola itd. Izmene ovih ključnih odluka u kasnijim fazama razvoja su skupe, jer zahtevaju fundamentalne promene u svih komponentama projekta, pa njihov broj treba svesti na minimum. Agilni razvoj softvera se može posmatrati kao kreiranje različitih opcija dizajna, čime se odlaže donošenje konačne odluke dok želje klijenta ne postanu jasnije. Kako bi se umanjio uticaj nepredvidivosti, razvojni tim učenjem kroz iteracije i eksperimentisanjem prilagođava plan realnom scenariju. Ovaj pristup podrazumeva započinjanje razvoja čim su delimični zahtevi klijenta poznati i korišćenje povratnih informacija iz kratkih iteracija za nastavak razvojnog procesa. Na ovaj način, donošenje odluka se odlaže do poslednjeg trenutka, odnosno sve do kad je moguće zadržati sve opcije dizajna otvorene. Neke od praksi koje se mogu primeniti kako bi se donošenje odluka odložilo uključuju⁶: korišćenje modularnog dizajna, razdvajanje interfejsa od implementacije (interfejsi treba da ostanu isti čak i ukoliko se implementacija funkcionalnosti promeni), korišćenje objektno-orijentisanog pristupa pisanju koda, korišćenje parametara umesto konstanti.

Kako bi agilni pristup doneo željene rezultate, proces razvoja se mora kontinuirano unapređivati. Dizajniranje proizvoda predstavlja iterativni proces koji započinje tako što inženjeri naprave nešto što radi, nauče iz njegovih slabosti i zatim unaprede dizajn. Kako bi proces unapređenja bio efikasan, potrebno je da arhitektura softverskog sistema od samog početka podržava glatko dodavanje novog skupa funkcionalnosti. Ukoliko se desi da programeri u nekom trenutku naiđu na problem u osnovnoj arhitekturi sistema, zaustavlja se proces dodavanja novih funkcionalnosti i pristupa se rešavanju uzroka problema pre nego što se nastavi sa razvojem.

Operativna strategija koja organizaciji donosi značajne konkurentske prednosti na tržištu je **brza isporuka**. Ova strategija je uklapa sa prethodno navedenim pristupom odlaganja odluka, jer što je isporuka brža, odluke se mogu duže odložiti.

Brza isporuka omogućava organizaciji da u kraćim intervalima izbacuje manje inkremente softvera. Na ovaj način, razvojni tim dobija pravovremene povratne informacije od klijenta koje pomažu u ranom identifikovanju i rešavanju problema. Organizacioni sistem koji se može primeniti kako bi se postiglo efikasno upravljanje resursima u razvojnem ciklusu projekta jeste korišćenje kartica sa zahtevima. Na početku svake iteracije klijent dostavi razvojnog timu funkcionalnosti koje su mu potrebne tako što ih kratko opiše na karticama. Na osnovu

6 H. Thimbleby, "Delaying Commitment" in IEEE Software, vol. 5, no. 03, pp. 78-86, 1988. doi: 10.1109/52.2027

prioriteta koji su određeni sa klijentom i informacija dobijenih od programera koji rade na projektu, menadžer tima procenjuje koliko vremena je potrebno za implementaciju svake kartice i u skladu sa tim određuje koje kartice će ući u trenutnu iteraciju. Nakon toga, kartice se postavljaju na tablu u „To Do“ sekciju, odakle programeri sami biraju karticu na kojoj žele da rade. Kada odaberu željenu karticu, dopisuju svoje ime na njoj i premeštaju je u sekciju „Selected“. Kada je posao na funkcionalnosti završen, kartica se premešta u „Finished“. Na ovaj način se obezbeđuje da u svakom trenutku programeri znaju na čemu treba da rade što osigurava efikasno korišćenje vremena, bez potrebe za predefinisanim planom. Optimizacija u okviru tima za testiranje se može postići podelom na manje funkcionalne jedinice koje je potrebno testirati, kako bi se sa jedne strane obezbedilo da članovi tima uvek imaju na čemu da rade i, sa druge strane, da ne budu prenatrpani poslom.

Na kraju, ako se ima u vidu da Lean metodologija podrazumeva agilni razvoj baziran na kratkim iteracijama u kojima članovi tima sami upravljaju i prate proces razvoja, postavlja se pitanje gde je tu uloga menadžera projekta. Početni zadatak koji predstavlja odgovornost menadžer projekta jeste da identifikuje otpad i kreira strategiju za njegovo uklanjanje. Ovo uključuje detaljnu analizu svih faza projekta kako bi se prepoznali procesi, resursi ili aktivnosti koje ne donose vrednost, a troše resurse. Menadžer projekta mora sarađivati sa timom i zainteresovanim stranama kako bi razumeo sve aspekte rada, od planiranja do implementacije. Nakon identifikacije otpada, menadžer kreira strategiju za njegovo eliminisanje koja može uključivati optimizaciju procesa, unapređenje komunikacije, uvođenje automatizacije ili drugih alata za poboljšanje efikasnosti. Ova strategija ne samo da povećava produktivnost i smanjuje troškove, već i omogućava timu da se fokusira na aktivnosti koje direktno doprinose ciljevima projekta i kreiraju stvarnu vrednost za klijente i korisnike. Osim toga, menadžeri projekta vode sastanke tima, staraju se da zacrtane aktivnosti budu realizovane, pružaju informacije o napretku i obezbeđuju neophodne resurse da bi tim uspešno izvršio svoje zadatke.

3.1.2.2. SCRUM

Zvanični Scrum Vodič⁷ je dokument koji opisuje principe i prakse Scrum metodologije, odnosno, razvojnog okvira koji pomaže timovima i organizacijama da rešavaju složene probleme putem prilagodljivih rešenja. Razvijen je početkom 1990-ih godina i od tada je evoluirao, tako da se u današnje vreme

⁷ (<https://scrumguides.org/scrum-guide.html>)

koristi u mnogim domenima koji su izvan sfere razvoja softverskih proizvoda iz koje Scrum vuče svoje korene. Frejmwork je namerno napravljen kao nepotpun, definišući samo delove potrebne za implementaciju Scrum teorije, kako bi timovima pružio dovoljno slobode da se prilagode specifičnim zahtevima konkretnog projekta. Scrum Vodič se sastoji od nekoliko sekcija, uključujući definiciju Scrum-a, njegovu teoriju, principe, pravila, članove tima, događaje i artefakte (proizvode). Zasnovan je na eksperimentisanju i „lean“ razmišljanju, po kome znanje dolazi iz iskustva i donošenja odluka na osnovu onoga što se posmatra i zaključuje. Scrum angažuje grupe ljudi koji zajedno poseduju sve veštine i stručnost potrebne za obavljanje posla ili stižu te veštine prema potrebi.

Scrum tim je mala grupa ljudi koja se sastoji od tri odgovornosti ili uloge: jednog **Scrum Master**-a, jednog **Product Owner**-a (vlasnika proizvoda) i više **Developer**-a (razvojni inženjera). Scrum timovi su samostalni u smislu da sami odlučuju ko šta radi, kada i kako. Timovi bi trebalo da budu dovoljno mali da ostanu agilni i dovoljno veliki da završe dovoljan posao unutar Srinta (obično do 10 ljudi). Ako Scrum timovi postanu preveliki, trebalo bi razmotriti njihovu reorganizaciju u više manjih Scrum timova, pri čemu je svaki fokusiran na isti proizvod, odnosno, dele isti cilj proizvoda, Backlog proizvoda i Product Ownera. Scrum tim je odgovoran za sve aktivnosti vezane za proizvod, uključujući saradnju sa zainteresovanim stranama, verifikaciju, održavanje, istraživanje i razvoj, i sve ostalo što bi moglo biti potrebno za razvoj projekta.

Razvojni inženjeri su odgovorni za kreiranje svih aspekata svakog Sprint inkrementa, uključujući kreiranje Sprint plana i njegovo svakodnevno prilagođavanje u skladu sa ciljem Srinta. Takođe su odgovorni za efikasno upravljanje listom stavki (funkcionalnosti) proizvoda (Product Backlog). Scrum Master je odgovoran za uspostavljanje Scrum okvira kako je definisano u Scrum vodiču i za pomoć svima da razumeju Scrum teoriju i praksu kroz poučavanje članova tima, obezbeđivanje produktivnih Scrum događaja, pomaganje u definisanju ciljeva i upravljanju listom stavki proizvoda i olakšavanje saradnje sa ostalim učesnicima u projektu. Iako nema ovlašćenja da određuje zadatke članovima tima, njegova uloga je da nadgleda proces razvoja, štiti tim od spoljašnjeg mešanja i vodi sastanke tima.

Zadatak vlasnika proizvoda je da obezbedi efikasno upravljanje Backlog-om, putem razvoja ciljeva proizvoda, kreiranja stavki u okviru Backlog liste i obezbeđivanja njene transparentnosti i jasnoće. Osim toga, donosi odluku o tome kada je proizvod spreman za isporuku i jedini ima ovlašćenje da skрати ili prekine Sprint u slučaju potrebe.

Scrum događaji su formalni procesi koji omogućavaju proveru i prilagođavanje Scrum proizvoda. **Sprint** predstavlja osnovni događaj u okviru Scrum-a. To je događaj fiksne dužine u trajanju od jednog meseca ili manje, koji započinje odmah nakon završetka prethodnog Sprints. U okviru Sprint-ova se obavljaju sve aktivnosti potrebne za postizanje cilja proizvoda, uključujući: Planiranje Sprint-a (Sprint Planning), Dnevni Scrum (Daily Scrum), Pregled Sprint-a (Sprint Review) i Retrospektivu Sprint-a (Sprint Retrospective). Svaki Sprint započinje **Planiranjem Sprint-a**, odnosno, definisanjem posla koji treba da se obavi tokom Sprints. Glavna pitanja kroz koja se prolazi u ovoj aktivnosti uključuju određivanje cilja Sprints i njegove vrednosti za zainteresovane strane, odabir stavki iz Backlog-a koje će biti uključene u konkretan Sprint i na koji način će konkretni zadaci biti izvršeni. Razvojni inženjeri dele stavke Backlog-a na manje jedinice koje se mogu završiti za jedan dan. Cilj Sprints, odabrane stavke iz Backlog-a i plan njihove realizacije zajedno čine *Sprint Backlog*. Proces planiranja Sprints ne bi trebalo da traje duže od osam sati. **Dnevni Scrum** je petnaestominutni događaj u kome učestvuju razvojni inženjeri, sa ciljem potencijalnog prilagođavanja planirane dinamike rada. Pregled Sprints je pretposlednji događaj Sprints, vremenski ograničen na najviše četiri sata za jednomesečni Sprint. Pregled Sprints ima za svrhu ispitivanje izlaza Sprints i određivanje budućih izmena. Scrum tim predstavlja svoje rezultate i postignuti napredak u razvoju proizvoda zainteresovanim stranama na osnovu čega se predlažu naredni koraci. Na kraju, **Sprint retrospektiva** služi da se odrede načini za poboljšanje kvaliteta i efikasnosti procesa razvoja. Ispituju se uzroci odstupanja od plana i dogovaraju se aktivnosti za unapređenje. Ovaj događaj najčešće traje manje od tri sata i njime se zaključuje Sprint.

Backlog proizvoda je lista funkcionalnosti izvedenih iz projektnih zahteva i prikazanih u vidu korisničkih priča. Vlasnik proizvoda, u saradnji sa razvojnim timom, prioritizuje korisničke priče na osnovu poslovnih prioriteta i eventualnih zavisnosti između priča. Upravljanje Backlog-om je odgovornost vlasnika proizvoda. Kada je neka stavka iz Backlog-a završena i temeljno testirana, kreira se novi **Inkrement**. Unutar jednog Sprint-a može biti kreirano više inkremenata i oni se predstavljaju na Pregledu Sprint-a. Međutim, inkrement se može isporučiti i pre kraja Sprints. Kada su završene sve stavke iz Backlog-a proizvoda i kada vlasnik proizvoda zaključi da je cilj proizvoda postignut i da proizvod ispunjava potrebe i očekivanja zainteresovanih strana, projekat se završava.

PROJEKAT	ZAHTEV (EPIC)	KORISNIČKE PRIČE
Izrada mobilne aplikacije za rezervaciju smeštaja u hotelima	Zakazivanje noćenja u hotelu putem odabira termina, sobe, broja osoba i potvrde podataka	• Kao korisnik, moram da budem u mogućnosti da pretražujem dostupne hotele prema lokaciji, ceni, oceni i sadržajima • Kao korisnik, moram da budem u mogućnosti da biram željeni termin, sobu i broj osoba • Kao korisnik, moram da budem u mogućnosti da proverim podatke koje sam odabrao pre plaćanja • ...
	Upravljanje i otkazivanje rezervacija bez potrebe kontaktiranja korisničke podrške	• Kao korisnik, želim da mogu da vidim sve svoje trenutne rezervacije • Kao korisnik, želim da mogu da otkazem svoju rezervaciju bez kontaktiranja korisničke podrške • Kao korisnik, moram da budem u mogućnosti da pratim istoriju svojih rezervacija • ...

U prethodnoj tabeli je prikazan primer razlaganja većih zahteva (u Scrum metodologiji se koristi termin *EPIC*) koje se ne mogu završiti u okviru jednog Sprint-a na korisničke priče. EPIC se deli na korisničke priče kako bi se olakšalo Scrum timu da se koncentriše na specifičnosti projekta i jednostavnije planira i upravlja realizacijom Sprint-ova. Na prvom mestu, podela na manje celine omogućava brže identifikovanje završenih i preostalih aktivnosti čime se pojednostavljuje praćenje napretka razvoja projekta. Prilikom kreiranja korisničkih priča neophodno je definisati uslove (kriterijume prihvatanja) koji moraju biti ispunjeni pre nego što se korisnička priča smatra završenom.

Na primer, u slučaju otkazivanja rezervacije, kriterijumi prihvatanja (eng. *acceptance criteria*) bi mogli biti: korisnici se mogu informisati o uslovima i troškovima u slučaju otkazivanja; korisnici mogu otkazati svoje rezervacije direktno iz aplikacije; korisnici dobijaju potvrdu o otkazivanju.

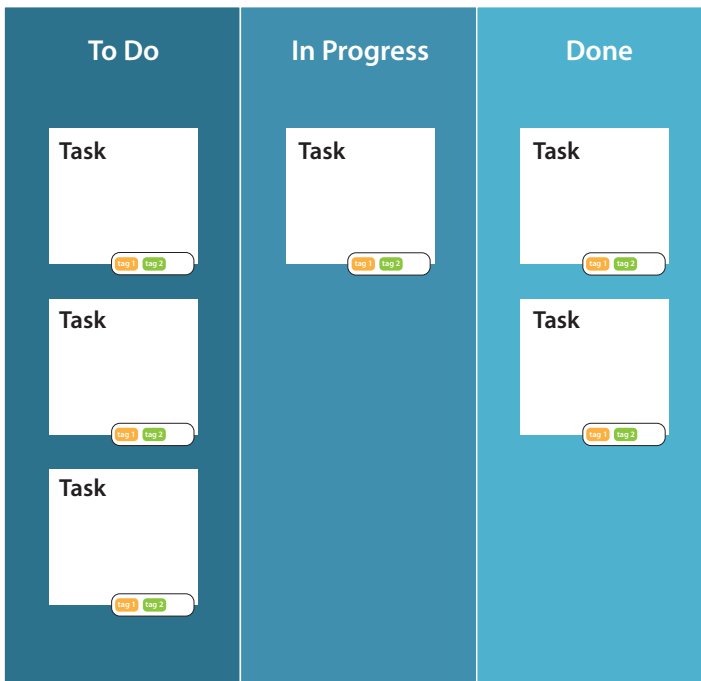
Najčešći način za vizuelno predstavljanje toka upravljanja i praćenja zadataka u tekućem Sprint-u je Scrum tabla. Tabla se sastoji od kolona koje označavaju različite faze u kojima se zadaci nalaze, (npr. To Do, In Progress, Complete) i od redova u kojima su izlistani zadaci koji su planirani za trenutni Sprint. Na ovaj način se obezbeđuje centralizovano upravljanje svim zadacima, što omogućava menadžerima da vizualizuju napredak i rešavaju probleme u realnom vremenu. Tradicionalna Scrum tabla se sastoji od fizičkih elemenata kao što su bela tabla, lepljivi papirići ili kartice. Može se podeliti u tri sekcije: zadaci koje tek treba započeti, zadaci čiji je rad u toku i zadaci koji su završeni. Lepljivi papirići se mogu koristiti za označavanje zadataka i njihovo pomeranje kroz različite sekcije. Uprkos sve većoj popularnosti digitalnih Scrum tabli, fizičke table i dalje imaju svoje prednosti u vidu podsticanja kreativnosti i diskusije, lakše vizualizacije i niskih troškova korišćenja.



Osim tradicionalnog načina, putem fizičkih tabli, Sprint-ovima se može upravljati korišćenjem onlajn Scrum alata. Prednosti onlajn tabli se ogledaju u povećanoj prilagodljivosti, jer pružaju mogućnost kaćenja dokumenata na tablu, što olakšava prikupljanje svih potrebnih podataka o projektu na jednom mestu. Osim toga, omogućavaju jednostavan rad timovima koji rade sa udaljenih lokacija ili u slučajevima kada osoba koja nije član Scrum tima (klijent ili menadžment organizacije) treba da bude informisana o napretku koji je ostvaren.

Većina ovih alata poseduje mogućnost automatskog obaveštavanja članova o promenama ili približavanju roka. Naročito korisna opcija predstavlja kontrola privilegija za dodavanje, izmenu i deljenje stavki na tabli čime se mogu definisati specifične dozvole za različite članove Scrum tima. Na kraju, praćenje istorije promena obezbeđuje dodatni nivo integriteta u proces upravljanja Sprint-om.

Online Scrum Board



3.1.2.2. Kanban metod

Kanban je strategija optimizacije toka razvoja projekta korišćenjem tehnike vizualizacije. Čine ga tri osnovne komponente: definisanje i vizualizacija toka, upravljanje stavkama i unapređenje toka. U proizvodnom okruženju Kan-ban (signalna kartica) se koristi da informiše tim koji upravlja prethodnim korakom proizvodnje da je potrebno napraviti novu komponentu. Radnicima nije dozvoljeno da rade ukoliko nisu dobili signalnu karticu sa nižeg proizvodnog nivoa. Druga primena Kanban sistema jeste sprečavanje preopterećenja kapaciteta proizvodnje kroz ograničavanje broja kartica konkretnih segmenata proizvodnje. Svakom delu procesa se dodeljuje posebna kartica. Broj kartica je ograničen. Kada se započne sa radom u nekom segmentu posla uzima se jedna kartica koja se odnosi na taj deo procesa. Ukoliko nema slobodnih kartica, posao ne može da se započne već se stavlja u red na čekanje. Kartica putuje od jednog dela procesa do drugog. Kada se ceo proces završi kartica se vraća i postaje dostupna za početak posla koji je na čekanju (ovaj mehanizam se naziva *pull system*). Na ovaj način se sprečava preopterećenje pojedinačnih proizvodnih segmenata ukoliko je kapacitet (broj kartica u sistemu) pravilno određen.

U kontekstu softverskog projekta, Kanban sistem ne koristi kartice u smislu signala za započinjanje novog posao, već da bi se njima predstavile pojedinačne stavke u okviru posla. Slično kao u proizvodnji, u razvoju softverskog projekta, kanban se koristi da bi ograničio količinu tekućeg posla postavljajući meru kapaciteta i da bi balansirao zahteve koji se timu nameću sa izlazom koji tim proizvodi. Putem vizuelizacije koja se koristi moguće je jasno utvrditi uticaj nedostataka i uskih grla na tok razvoja. Za razliku od Scrum-a, Kanban po sebi ne predstavlja metodologiju razvojnog ciklusa projekta, već opisuje principe koji se prate kako bi se postojeća metodologija razvoja prilagodila Lean pristupu. Samim tim, Kanban pruža veći stepen slobode u kreiranju i razvoju procesa koji će se koristiti prilikom razvoja softverskog proizvoda. Prvi zadaci u primeni Kanban sistema jesu definisanje početka i kraja toka koji će se posmatrati i određivanje načina komunikacije sa klijentom, menadžmentom i drugim timovima. Nakon toga, potrebno je odrediti tipove ulaza koji dolaze na početnu tačku toka i one stavke koje postoje u samom toku. Na primer, ulazi mogu biti korisničke priče, korisnički zahtevi ili funkcionalnosti (ili Epic iz Scrum terminologije). Primeri posla koji se javlja u okviru samog toka razvoja su zahtevi za promenom, nedostaci, održavanje sistema, ažuriranje alata i sl. Kada su sve stavke koje cirkulišu u okviru posmatranog toka definisane, može se pristupiti izradi Kanban table sa karticama. Tablu je potrebno podeliti po aktivnostima koje se obavljaju u okviru procesa na koji se posmatrani tok odnosi. Stavke u okviru aktivnosti dodatno treba kategorisati na one koji su trenutno aktivne (u radu) i na one koji su završene.

U nastavku se može videti primer Kanban table.

Ulazni red	Analiza		Spremno za razv.	Razvoj		Spremno za test.	Testiranje		Spremno za prod.	Produkcija
	u radu	završeno		u radu	završeno		u radu	završeno		



U navedenom primeru je značajno zapaziti, da se, osim aktivnosti, nalaze i takozvane „buffer“ sekcije – Spremno za razvoj, Spremno za testiranje i Spremno za produkciju, koje će kasnije poslužiti da se odrede uska grla u procesu (ukoliko je broj kartica u datoj sekciji veliki, to ukazuje na usko grlo).

Dizajn kartica na tabli igra važnu ulogu u Kanban sistemu. Informacije koje se na njoj nalaze treba da omoguće članovima tima donošenje odluka u okviru „pull“ sistema, bez potrebe za intervencijom menadžera projekta. Osnovni podaci bi trebalo da uključuju jedinstveni identifikator kartice, odnosno posla, zatim, datum ulaska kartice u određenu aktivnost, naziv posla (funkcionalnosti, zahteva) i polje sa imenom osobe koja je na sebe preuzela rad na kartici. U slučaju potrebe, kartica može da sadrži kratak opis posla i datum koji je predviđen za njegov završetak, kada se radi o poslovima koji zahtevaju precizne rokove. Osim toga, neki timovi praktikuju da koriste različite nalepnice ili magnete koje stavljaju na karticu kako bi ukazali na prioritet posla ili eventualno kašnjenje u odnosu na planirani rok.

KABAN KARTICA

ID: _____

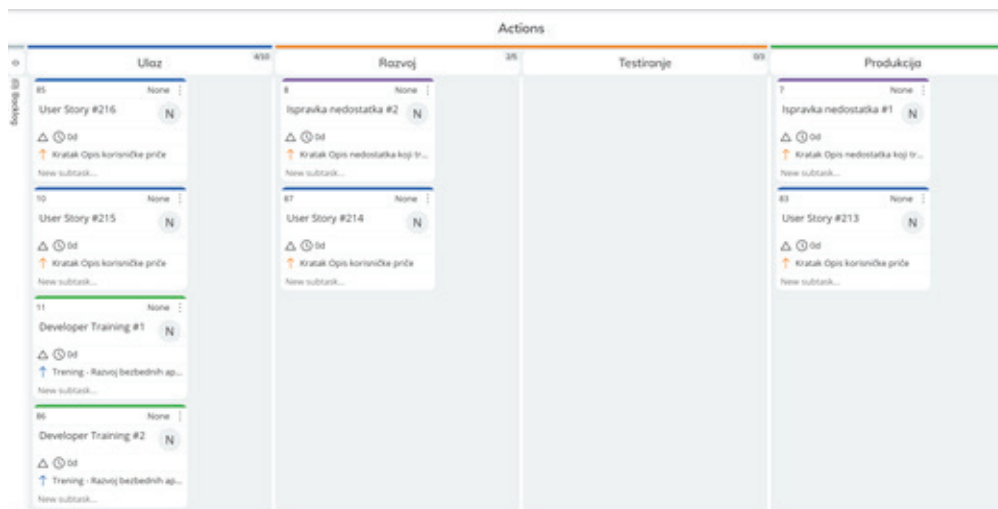
Ulaz: _____

Naziv: _____

Osoba: _____



Kapacitet (maksimalni broj kartica) svake kolone se najčešće navodi pored njenog naziva. Kretanje kartica kroz tablu se obavlja u skladu sa naznačenim ograničenjem. Na jednostavnom primeru elektronske Kanban table⁸ prikazanom na slici ispod, može se videti da je kapacitet sekcije „Razvoj“ pet stavki, a da se u njoj trenutno nalaze dve. To signalizira članovima tima da mogu prebaciti tri kartice iz „Ulaza“ u sekciju „Razvoj“. Odluka o tome koje kartice će biti premeštene se donosi na osnovu informacija o tipu posla, trajanju i rokovima za realizaciju. S obzirom da se nakon premeštanja stavki u sekciji „Ulaz“ nalazi samo jedna kartica, menadžer projekta u nju može ubaciti dodatnih devet najprioritetnijih funkcionalnosti.

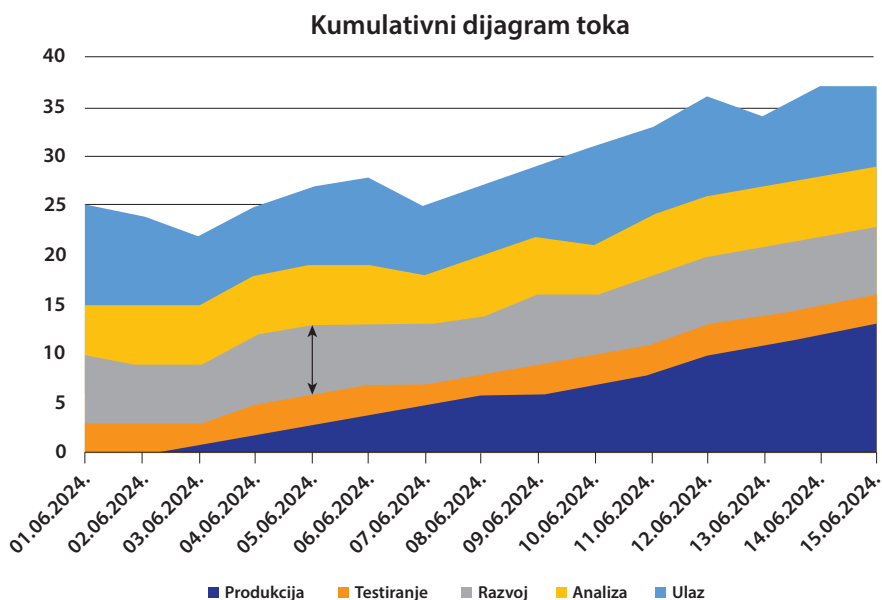


⁸ businessmap.io

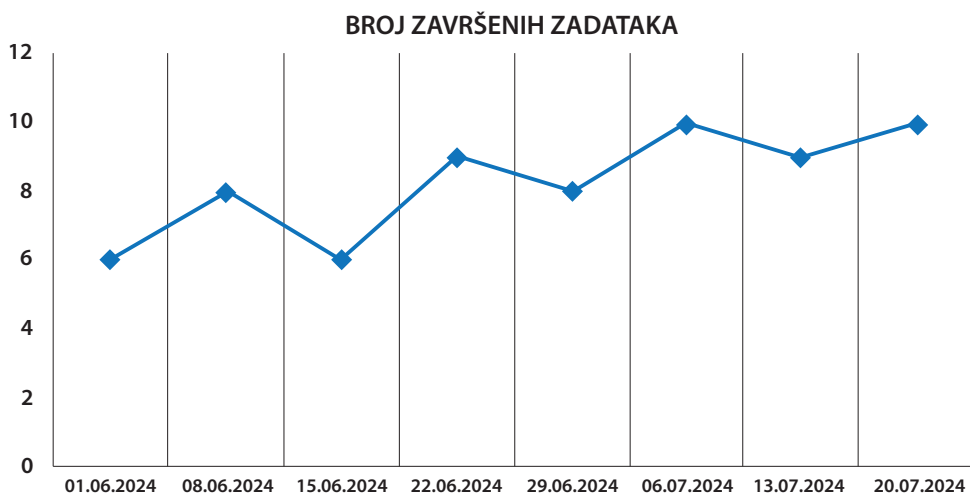
Jedan od najvažnijih i najsloženijih zadataka u upravljanju zadacima korišćenjem Kanban table jeste određivanje kapaciteta *aktivnih* sekcija table (sekcije „u radu“). Najjednostavniji slučaj jeste kada jedan član tima može biti angažovan samo na jednom poslu. U toj situaciji, broj članova tima istovremeno predstavlja i kapacitet aktivnih sekcija. Na primer, ukoliko tim čini pet programera i tri testera, kolona „u radu“ aktivnosti „Razvoj“ ima kapacitet pet, a „Testiranje“ tri kartice. Ukoliko se želi dozvoliti da članovi tima rade na više zadataka istovremeno, dobra praksa jeste da kapacitet aktivnih sekcija predstavlja dupliran broj članova tima koji su angažovani u okviru date aktivnosti (ako ima pet članova tima, kapacitet je deset). Kada je reč o kapacitetu „buffer“ sekcija, opšte pravilo je da maksimalan broj stavki treba da bude što je moguće manji. Ove sekcije se koriste da umanje preopterećenje sistema tako što predstavljaju zonu u kojoj se nalaze poslovi koji su spremni za narednu aktivnost, ali čekaju da se za njih oslobodi mesto.

Za razliku od tradicionalnih agilnih metoda (npr. Scrum) kod kojih su intervali isporuke, odnosno trajanja iteracija strogo vremenski definisani, u Kanban sistemu rokovi se prilagođavaju u odnosu na karakteristike konkretnog posla. Isporučivanje novih verzija proizvoda u redovnim intervalima ima mnogobrojne prednosti od kojih su najznačajnije pružanje osećaja sigurnosti i uspostavljanje odnosa poverenja sa klijentom. Međutim, često je izazovno razložiti posao koji treba uraditi na dovoljno male gradivne celine koje se mogu završiti u okviru veštački definisanih rokova, naročito ako je česta isporuka cilj kome organizacija teži. U određivanju termina isporuke Kanban metod preporučuje da se, osim obima posla, u obzir uzme prioritet zahteva koji su obrađeni u datoj verziji, troškovi isporuke i vrednost koju isporučeni proizvod donosi klijentu, time pružajući timu veću slobodu u samostalnom odlučivanju o trajanju konkretne iteracije.

Fokusiranost Kanban pristupa na prilagodljivost toka razvoja podrazumeva korišćenje posebne metrike za merenje uspešnosti implementirane metode. Najbolji pregled performansi sistema se može videti na kumulativnom dijagramu toka (eng. *Cumulative flow diagram*). Kumulativni dijagram toka predstavlja vizuelni prikaz količine posla u aktivnim sekcijama pojedinačnih aktivnosti u datim vremenskim intervalima. Konkretno, na dijagramu koji je dat kao primer, može se videti da se 05.06. u aktivnim poslovima razvoja nalazilo sedam stavki (kartica), što je označeno crnom strelicom. U dobro optimizovanim Kanban sistemima, granice traka (gornje ivice) aktivnosti treba da budu glatke, a njihova visina da bude konstantna. Ukoliko je visina trake u nekom trenutku suviše mala u odnosu na kapacitet aktivnosti, to ukazuje na nedovoljno iskorišćenje članova tima u okviru konkretne aktivnosti, odnosno da je kapacitet sekcije veći nego što je potrebno. Sa druge strane, naglo povećanje visine trake ukazuje na znatno veći broj poslova koji pristižu u odnosu na one koji iz sekcije izlaze, što potencijalno može predstavljati usko grlo procesa.



Najbolji pokazatelj mere uspešnosti kontinuiranog *unapređenja* Kanban sistema jeste dijagram koji prikazuje broj stavki koje su u potpunosti završene (koje su prešle u produkciju) u određenim vremenskim perioda (npr. na nedelju dana).



U Kanban sistemu koji se stalno unapređuje grafik će pokazati konstantni trend rasta broja završenih poslova.

Na kraju bismo napomenuli da Kanban nije jedini način primene „pull“ sistema iz Lean pristupa. Alternativni način predstavlja Goldratova Teorija ograničenja⁹. Osnovna postavka teorije jeste da se unapređenje procesa postiže kroz eliminisanje uskih grla. Zadatak je identifikovati usko grlo u procesu, analizirati ga i pronaći način da se ono ublaži sve dok ne prestane da ograničava tok procesa. Kada se ovo postigne, pojaviće se novo usko grlo, koje je potrebno analizirati i eliminisati. Kontinuiranim uklanjanjem uskih grla se postiže sistematsko unapređenje performansi procesa. Ako se proces razvoja softvera vizuelno predstavi kao tok, odnosno prelazak funkcionalnosti iz jednog statusa u drugi, pronalaženje uskih grla se svodi na nalaženje statusa sa najvećim brojem stavki. Na primer, ako pogledamo Scrum tablu o kojoj je bilo reči u prethodnom poglavlju i ustanovimo da se najviše papirića nalazi u „To Do“ sekciji, to nam ukazuje da usko grlo predstavlja broj programera u timu ili nedovoljno optimizovana aktivnost razvoja.

3.1.3. Ekstremni pristup upravljanja projektom (xPM)

Kao što mu samo ime kaže, ekstremni pristup razvoju softvera se primenjuje u situacijama koje proizilaze iz specifičnih slučajeva projekata kod kojih cilj i rešenje nisu poznati ili ih nije moguće precizno odrediti. Ovi projekti najčešće spadaju u grupu istraživačko-razvojnih projekata koje karakteriše visok stepen neuspešnosti. Iz ovog razloga, projekti iz ove kategorije su visokorizični projekti.

U slučaju istraživačko-razvojnih projekata krajnje željeno stanje do koga projekat treba da dovede nije uvek jasno. Uzimo za primer istraživanje i procenu načina integracije veštačke inteligencije u okviru različitih poslovnih procesa organizacije. Organizacija je svesna potencijala primene veštačke inteligencije u poboljšanju efikasnosti i korisničkog iskustva, ali konkretni primeri primene i optimalni načini integracije u postojeće tokove rada još uvek nisu utvrđeni. Ovaj projekat poseduje uopštenu svrhu istraživanja, ali specifični primeri upotrebe ili metrike performansi, nisu jasno definisani na početku i treba ih otkriti u toku trajanja projekta. Osim toga, razvojni tim nema unapred definisano rešenje za implementaciju veštačke inteligencije, već mora eksperimentisati sa različitim modelima i algoritmima kako bi se identifikovalo najefikasnije rešenje za upotrebu veštačke inteligencije u organizaciji.

Drugi primer može predstavljati projekat koji istražuje mogućnosti primene i izvodljivost integracije kvantnog računarstva u različitim sferama industrije. Razvojni tim projekta ima zadatak da istraži i otkrije potencijalne primene i koristi kvantnog računarstva, pri čemu konačno rešenje nije unapred određeno.

9 E. M. Goldratt, Theory of Constraints. North River Press. 1999.

Ekstremne modele razvoja projekata karakteriše potreba da se istraživanje odvija i na strani klijenta i na strani projektnog tima. Za razliku od drugih modela, opseg (scope) projekta se može menjati nakon svake razvojne faze. Ako je klijent nezadovoljan napretkom, projekat se može pokrenuti od početka u drugom smeru.

3.1.4. Emurci (Emertxe) pristup upravljanja projektom (MPx)

Kada je rešenje poznato, a cilj nije definisan, radi se o projektima koji spadaju u emurci razvojne projekte. Iako na prvi pogled može delovati da je ovakva situacija u praksi neostvariva, učestalost projekata čiji je zadatak pronaći cilj na osnovu postojećih saznanja je sve veća. Moglo bi se reći da je metodološki pristup MPx projektima obrnut od pristupa projektima koji pripadaju ekstremnoj grupi. (I sam naziv „Emertxe“ zapravo predstavlja „extreme“ napisano unazad). Kada je reč o „ekstremnim“ projektima, najčešće postoji predstava o nekom željenom krajnjem stanju (iako ono nije jasno određeno), a projekat ima za cilj da pronađe odgovor na pitanje da li i na koji način je moguće ostvariti željeno stanje. Pri tome, treba imati u vidu da je, u slučaju potrebe, moguće promeniti traženo krajnje stanje ukoliko se pokaže da je njegovo dostizanje nepraktično. Sa druge strane, u slučaju MPx razvojnog pristupa, tim koji radi na projektu već poseduje neko rešenje, ali još uvek nije otkrivena primena tog rešenja u praktičnim situacijama. U ovom kontekstu se može reći da je cilj MPx projekata da pronađu problem koji se rešava primenom već postojećih otkrića ili saznanja do kojih se došlo (namerno ili slučajno). Dakle, projekat se može smatrati uspešnim ukoliko se pronađe primena rešenja koja ima neku poslovnu vrednost.

Pretpostavimo da organizacija radi na projektu istraživanja načina primene veštačke inteligencije u različitim sferama društva. Sam po sebi, ovaj projekat spada u grupu ekstremnih projekata. Međutim, zamislimo situaciju da je razvojni tim u toku svog rada na projektu i istraživanja različitih modela i algoritama veštačke inteligencije došao do ideje kako da razvije potpuno novi model mašinskog učenja. U ovom slučaju, tim poseduje neko stečeno znanje, ali se postavlja pitanje kako i gde se ono može primeniti. Na ovaj način se zapravo dolazi do potrebe formiranja novog projekta, ovog puta iz emurci kategorije, koji treba da pronađe odgovor na ovo pitanje.

U praksi, najčešći izvor MPx projekata predstavljaju akademski članci. Članci predstavljaju pouzdane izvore koji pružaju temeljne opise tehnoloških dostignuća. Integracijom najnovijih istraživačkih rezultata, projekti stiču legitimitet i rigoroznost koje proističu iz naučnog diskursa. Štaviše, naučne publikacije garantuju da su članovi tima koji rade na ovim projektima upoznati sa najnovijim i relevantnim saznanjima i dostignućima iz oblasti koju projekat obuhvata.

3.2. RAZVOJNI TIM SOFTVERSKOG PROJEKTA

U zavisnosti od izabranog pristupa upravljanju projektom, sastav i funkcionisanje tima za razvoj projekta može se značajno razlikovati.

Kada je reč o tradicionalnim projektima, tim je obično organizovan u hijerarhijsku strukturu sa jasno određenim ulogama i odgovornostima. Naglasak je na striktno specijalizovanim ulogama, kao što su menadžer projekta, analitičar, programer ili tester. Kanali komunikacije su formalni i strogo definisani i često se obavljaju putem dokumentacije ili u pisanoj formi.

Agilni pristup podrazumeva timove u kojima član može da ima više različitih funkcija. Često su manji po veličini, ali imaju mogućnost samoupravljanja. Očekuje se da članovi tima imaju raznovrsne veštine, podstičući saradnju i fleksibilnost. Naglasak je na čestoj i neformalnoj komunikaciji, putem ličnih, neposrednih interakcija. Primarni akcenat je na iterativnom planiranju u svakoj fazi projekta, sa fokusom na prilagođavanje promenama i isporuku malih, inkrementalnih verzija.

Visoko fleksibilni timovi, sposobni za brze odgovore na promene, neophodni su za uspešno realizovanje ekstremnih modela razvoja projekata. Karakteriše ih promenljiva struktura članova projektnog tima u zavisnosti od faze u kojoj se projekat nalazi. Tako, na primer, u početnoj fazi, tim se može sastojati od teoretičara iz oblasti veštačke inteligencije i modela mašinskog učenja koji će kasnije biti zamenjeni timom programera čiji je zadatak konkretna implementacija rešenja do kojih je prethodna grupa zaposlenih došla. Kanali komunikacije moraju biti visoko razvijeni i prilagodljivi, sa akcentom na rešavanju prioriternih izazova koji su česti u ovom tipu projektnog razvoja. Planiranje se odvija u okviru svake faze i bazira se na naučenim lekcijama iz prethodnih koraka. Na taj način, putem iterativnog učenja i planiranja stiže se do željenog rešenja ili se projekat preusmerava na drugi pravac razvoja.

Karakteristike emurci projekata diktiraju specifičnost strukture timova koji učestvuju u njihovoj realizaciji. Najčešće su uključeni članovi akademske zajednice koji formulišu teorijske osnove predloženog rešenja i tehnička lica koja rešenje treba da sprovedu u praksi. Akcenat tima je na sposobnosti analitičkog razmišljanja i pronalaženja novih primena koje donese poslovnu vrednost organizaciji. Planiranje je iterativnog tipa, pri čemu se krajnji cilj projekta menja, sve dok ne bude u potpunosti postignut ili dok budžet ne bude iscrpljen.

3.2.1. Članovi projektnog tima

Menadžer projekta je osoba koju je organizacija odredila kao odgovorno lice za postizanje ciljeva projekta. Njegova osnovna uloga jeste da obezbedi viziju i smernice projektnom timu i usklađuje njihove napore s opštim ciljevima i zadacima projekta.

Menadžeri projekta odgovorni su za temeljno planiranje projekta. To uključuje definisanje opsega projekta, pravljenje rasporeda, dodeljivanje resursa i kreiranje budžeta. Efikasno planiranje navedenih komponenti postavlja osnovu za uspešnost projekta, obezbeđujući da su zadaci jasno definisani i izvodljivi u okviru zadatih ograničenja. Menadžeri projekta moraju imati razvijene sposobnosti rešavanja problema i fleksibilnost koja je neophodna za promenu pravca razvoja u slučaju potrebe, pri čemu su odgovorni da projekat ostane u okviru planiranog rasporeda.

Jedan od ključnih zadataka menadžera predstavlja identifikacija i upravljanje rizicima. On procenjuju potencijalne izazove koji mogu nastati tokom projekta, razvija strategije za umanjeње rizika i kontinuirano prati i upravlja rizicima tokom životnog ciklusa projekta. Ovaj proces podrazumeva efikasnu komunikaciju kao jednu od osobina koje menadžer projekta mora da poseduje. Od njega se očekuje da odgovara na upite, upravlja očekivanjima zainteresovanih strana i obaveštava ih o statusu projekta.

Još jedna ključna komponenta posla menadžera projekta je obezbeđivanje kvaliteta isporučenih rezultata. Njegov zadatak je da uspostavljanja i održava standarde kvaliteta, sprovodi česte evaluacije i uspostavlja pravila, kako bi se garantovalo da projekat ostane unutar unapred definisanih parametara.

Menadžer projekta igra ključnu ulogu u sastavljanju i upravljanju timom projekta. Ovo podrazumeva dodeljivanje zadataka na osnovu veština članova tima, podsticanje na saradnju i stvaranje pozitivnog okruženja za rad. Osim toga, njegov zadatak je da rešava potencijalne konflikte, kako u samom timu, tako i u odnosu sa zainteresovanim stranama.

Pored menadžera projekta, koji predstavlja integralni deo svakog razvojnog tima, u zavisnosti od konkretnih potreba, u proces razvoja projekta mogu biti uključene sledeće uloge:

- **Analitičari** - prikupljaju i dokumentuju zahteve projekta u saradnji sa zainteresovanim stranama. Najčešće su angažovani u početnim fazama projekta sa zadatkom da zajedno sa menadžerom projekta analiziraju i razlože prikupljene zahteve na manje, razumljive komponente. To uključuje

dokumentovanje funkcionalnih i nefunkcionalnih zahteva, kreiranje slučajeva korišćenja i razvoj detaljne specifikacije. Osim toga, njihov posao je da, kroz efikasnu komunikaciju, premoste jaz između poslovnih potreba i tehničkih rešenja, prenoseći složene, tehničke informacije na jeziku koji je razumljiv zainteresovanim stranama. U kasnijim fazama projekta, ostvarivanjem koordinacije između projektnog tima i zainteresovanih strana, pomažu u upravljanju zahtevima za promenama, kako bi se promene efikasno implementirale i bile u skladu sa ciljevima projekta. Na kraju, analitičari mogu biti zaduženi za održavanje ažurne projektne dokumentacije.

- **Članovi programerskog tima** – implementiraju i razvijaju rezultate projekta. Dok se menadžeri projekta fokusiraju na ukupno planiranje, koordinaciju i kontrolu projekta, programeri imaju ključnu ulogu u aspektima tehničke implementacije i razvoja. U početnim fazama, njihov zadatak je da pruže uvid u tehničku izvodljivost ciljeva projekta i učestvuju u proceni neophodnih resursa i vremenskih okvira. Primarna odgovornost programera je pisanje koda i implementacija projektovanih rešenja prema specifikacijama projekta. U toku razvoja, imaju ključnu ulogu u identifikaciji i rešavanju tehničkih izazova koji se mogu javiti tokom razvojnog procesa. Na kraju, njihov posao je da učestvuju u aktivnostima testiranja i osiguravanja kvaliteta proizvoda.
- **Inženjeri za testiranje** (QA inženjeri, inženjeri za osiguranje kvaliteta ili tester) – osiguravaju da softver ispunjava zadate standarde kvaliteta pre nego što bude isporučen klijentu i pušten u produkciju. Pre početka testiranja odgovoran je za pravljenje plana testiranja na osnovu specifikacija projekta tako što identifikuje ključne funkcionalnosti koje zahtevaju testiranje, i pravi strategiju testiranja. Na osnovu plana testiranja, inženjer za testiranje razvija testne scenarije i slučajeve korišćenja koji pokrivaju različite aspekte softverskog proizvoda, a zatim definiše očekivane rezultate i kriterijume za uspeh za svaki testni scenario. Kada otpočne faza testiranja, tim za testiranje izvršava predviđene testne slučajeve koristeći ručno i automatizovano testiranje. U toku trajanja procesa testiranja neophodno je praćenje i dokumentovanje rezultata testiranja i prijavljivanje otkrivenih grešaka razvojnom timu. Primarni zadatak inženjera za testiranje je da doprinose procesu osiguranja kvaliteta softvera tako što će se postarati da softver bude stabilan, funkcionalan i da odgovara postavljenim standardima pre nego što bude dostupan korisnicima.

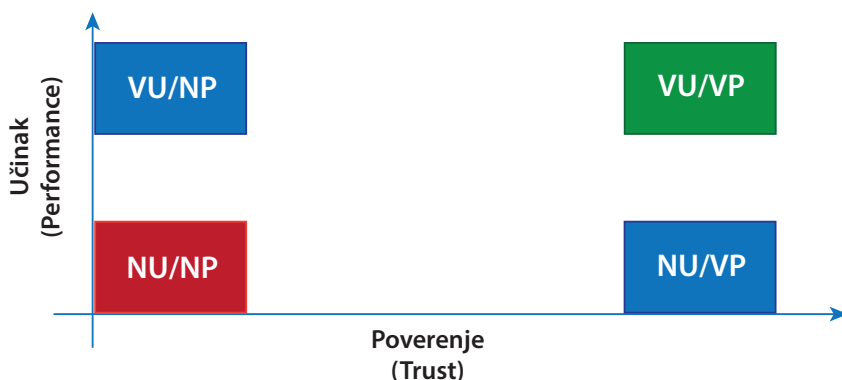
- **Spoljni eksperti** – obavljaju dopunske aktivnosti potrebne za razvoj ili izvođenje plana upravljanja projektom. Poslovi spoljnih eksperata mogu uključivati finansijsko upravljanje, logistiku, pravne poslove, bezbednost, inženjering, ispitivanje ili kontrolu kvaliteta. Zavisno o veličini projekta i nivou podrške koji je potreban, podrška eksperata može biti angažovana puno radno vreme ili se mogu samo pridružiti timu kada su određene veštine potrebne.
- **Prodavci** – eksterne kompanije koje ulaze u ugovorni sporazum kako bi obezbedile komponente ili usluge neophodne za projekat.
- **Poslovni partneri** – eksterne kompanije koje pružaju specijalizovano znanje ili obavljaju određenu ulogu kao što je instalacija, prilagođavanje, obuka ili podrška.

Kvalitet predstavlja stepen usaglašenosti završenog projekta sa eksplicitno iskazanim zahtevima u projektnoj specifikaciji. Iz tog razloga, osnovni cilj projektnog tima jeste da zajedničkim naporima obezbedi kvalitetan proizvod. Nivo kvaliteta softvera je glavno merilo uspešnosti projektnog tima i osnovni kriterijum koji klijenti uzimaju u obzir prilikom ponovnog odabira softverske kompanije.

3.2.2. Izbor članova projektnog tima

U jednom od svojih predavanja, Simon Sinek, pisac knjiga o poslovnom upravljanju, govori o načinu odabira članova tima koji će biti angažovani na projektima u okviru organizacije.

Radeći sa američkim mornaričkim fokama (US Navy Seals), koje predstavljaju jednu od najefikasnijih organizacija na svetu, upitao ih je kako od hiljadu kandidata koji su im na raspolaganju biraju šest članova tima koji će učestvovati u nekoj konkretnoj operaciji. Akcije mornaričkih foka se tiču ljudskih života, odnosno, u pitanju su visokorizični „projekti“ sa velikim ulogom. Kako bi odgovorio na postavljeno pitanje, predstavnik foka je nacrtao dijagram (čija je reprezentacija prikazana ispod).



Na prikazanom grafiku, iks (x) osa označava nivo poverenja u određenu osobu, a ipsilon (y) osa predstavlja njen učinak. Učinak je definisan kao stepen ostvarenja ciljeva na bojnopolju (na radnom mestu), koji zavisi od veština i sposobnosti individue. Poverenje označava kakva je neko osoba izvan bojnog polja (izvan radnog prostora, kancelarije). Simbolički, poverenje se može odrediti rečenicom: „I may trust you with my life, but do I trust you with my money and my wife“ (Mogu ti verovati životom, ali da li ti verujem i sa svojim novcem i ženom).

Najnepoželjniji članovi tima su osobe sa niskim učinkom i niskim poverenjem (NU/NP). U idealnim slučajevima, svi članovi tima će biti osobe sa visokim učinkom i visokim poverenjem (VU/VP). Međutim, borbeno iskustvo američkih foka ih je naučilo da osobe koje imaju visok učinak, a nisko poverenje (VU/NP) predstavljaju toksične vođe i toksične članove tima. Zbog toga, radije u tim primaju ljude koji imaju nizak učinak i visoko poverenje (NU/VP), nego pomenute toksične članove.

Problem u poslovnom svetu jeste taj što je u najvećem broju slučajeva ova logika izvrnuta. Postoji ogroman broj metrika za određivanje performansi nekog zaposlenog, a gotovo ni jedna koja meri nečiju pouzdanost. Iz tog razloga, zaposleni koji imaju visok učinak bivaju unapređeni, čime se nagrađuje toksičnost, što na duge staze predstavlja loš izbor jer će na kraju uništiti organizaciju. Zapravo, prirodno nadarene vođe su one osobe koje će uvek biti tu kada je teško i koje prave zdravo okruženje za rad i za uspeh svih ostalih članova tima.

Sada, kada su prikazane karakteristike četiri osnovna pristupa razvoju softverskog projekta, možemo pogledati kako odabir modela utiče na uspešnost završetka projekta. U nastavku je data tabela iz pomenutog izveštaja Standish Group organizacije koja sadrži podatke o procentu uspešnosti u zavisnosti od korišćene Agile ili Waterfall (vodopad) metode i veličine projekta.

Size	METHOD	SUCCESSFUL	CHALLENGED	FAILED
All Size Project	Agile	39%	52%	9%
	Waterfall	11%	60%	29%
Large Size Project	Agile	18%	59%	23%
	Waterfall	3%	55%	42%
Medium Size Project	Agile	27%	62%	11%
	Waterfall	7%	68%	25%
Small Size Project	Agile	58%	38%	4%
	Waterfall	44%	45%	11%

U prvom delu tabele se nalaze rezultati za sve projekte, iz kojih se može videti da agilni projekti imaju gotovo četiri puta veću uspešnost ($\approx 40\%$) u odnosu na projekte u kojima su korišćeni tradicionalni modeli razvoja ($\approx 10\%$), odnosno, „vodopad“ model pokazuje trostruko veći neuspeh u odnosu na agilni pristup. Rezultati su takođe razvrstani prema veličini projekta: veliki, srednji i mali. Interesantno je zapaziti da što je projekat manji, to je manja razlika u uspešnosti između agilnog i vodopadnog modela. Takođe, vredi napomenuti da kombinacija malih projekata sa agilnom metodom daje najbolje rezultate, jer je procenat neuspešnih projekata svega 4%.

SOFTVERSKI ALATI ZA UPRAVLJANJE PROJEKTOM



4. SOFTVERSKI ALATI ZA UPRAVLJANJE PROJEKTOM

Kako bi se omogućilo praćenje, kontrola i ispunjenje ciljeva, prvi korak u pravljenju plana razvoja projekta jeste razlaganje projekta na manje delove: faze, aktivnosti i zadatke. Iako faze nisu neophodna gradivna jedinica projekta, često se koriste da pojednostave proces upravljanja. Svaka faza može imati svoj skup aktivnosti i zadatak koji doprinose postizanju specifičnih ciljeva te faze.

Aktivnosti predstavljaju konkretne korake koje je potrebno izvršiti kako bi se postigli ciljevi projekta. Proces kreiranja aktivnosti nužno podrazumeva identifikaciju i razumevanje zavisnosti između različitih aktivnosti. Zavisnost između dve aktivnosti bi se mogla definisati kao neka vrsta ograničenja u njihovoj izvodljivosti, odnosno situacija u kojoj je završetak jedne aktivnosti uslovljen početkom ili završetkom druge. Aktivnosti se dalje mogu razložiti na konkretne zadatke, čime se olakšava dodeljivanje konkretnih poslova i odgovornosti i praćenje napretka u kratkim vremenskim intervalima. Svaka aktivnost treba biti vremenski ograničena, što omogućava uspostavljanje realnih rokova za završetak projekta i efikasnije upravljanje vremenom u toku razvoja.

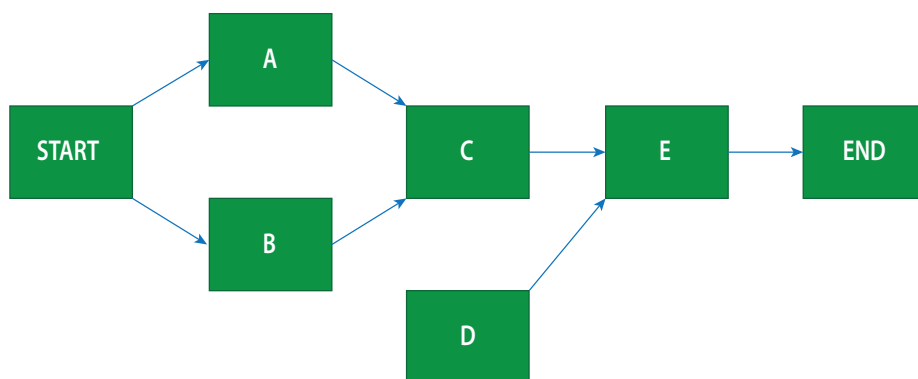
Između dve aktivnosti može postojati četiri tipa zavisnosti u odnosu na to da li se ograničenja odnose na početak ili završetak aktivnosti:

- **Finish-to-Start (FS)** – Ova vrsta zavisnosti je najčešća i podrazumeva da aktivnost B ne može početi dok se aktivnost A ne završi. Na primer, pre nego što počne sastavljanje izveštaja (aktivnost B), mora se završiti prikupljanje podataka (aktivnost A), ili, početak podizanja zidova (A) ne može početi dok se ne završi izgradnja temelja (B).
- **Start-to-Start (SS)** – U ovom slučaju, aktivnost B može početi samo kada počne aktivnost A. Na primer, početak obuke u novoj tehnologiji (aktivnost A) je neophodan za započinjanje razvoja softvera (aktivnost B). Drugi primer je aktivnost monitoringa ili praćenja (B) koja može početi tek kada počne aktivnost koja se nadgleda (aktivnost A).
- **Finish-to-Finish (FF)** – Ova vrsta zavisnosti implicira da se aktivnost B ne može završiti dok se aktivnost A ne završi. Recimo, pisanje dokumentacije (aktivnost B) može se završiti samo kada se završi testiranje funkcionalnosti softvera (aktivnost A).
- **Start-to-Finish (SF)** – U ovoj vrsti zavisnosti, aktivnost B se može završiti tek kada počne aktivnost A. Ovaj tip zavisnosti je manje uobičajen, ali može se primeniti u određenim situacijama. Na primer, pretpostavimo da se ne može završiti priprema izveštaja (aktivnost B) dok se ne počne istraživanje (aktivnost A).

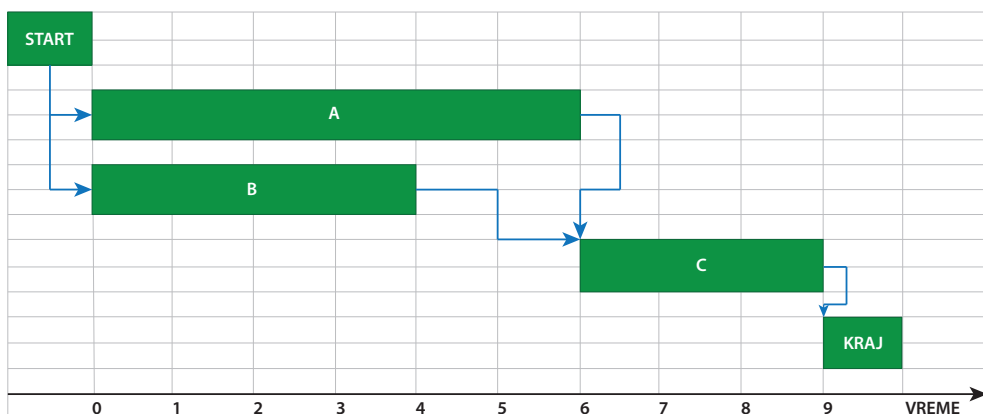
4.1. MREŽNI DIJAGRAM

Mrežni dijagram je grafički alat koji ilustruje međuzavisnosti između projektnih aktivnosti i redosled kojim bi one trebalo da budu završene. Uz njegovu pomoć menadžeri projekata jednostavnije planiraju, organizuju i kontrolišu tok rada unutar projekta. Mrežni dijagrami se često koriste u kombinaciji sa drugim alatima za upravljanje projektima, kao što je Gantov grafikon (Gantt chart), kako bi se pružio detaljniji prikaz toka razvoja projekta. Za razliku od Gantovog grafikona (Gantogram) koji prikazuje samo vremensku liniju projekta, mrežni dijagram prikazuje zavisnosti između različitih aktivnosti u projektu (FS, SS, FF, SF). Iako ovi dijagrami mogu da se kreiraju ručno, najčešće se generišu automatski korišćenjem specijalizovanih softvera.

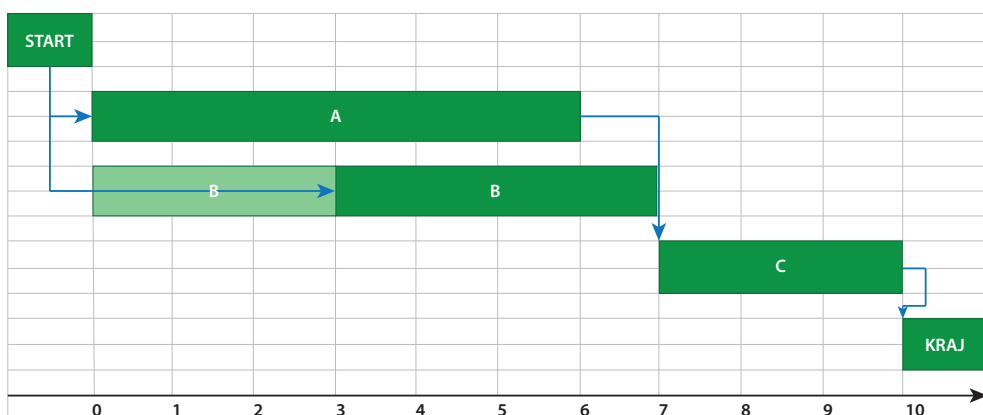
Postoji više vrsta mrežnih dijagrama. Kao reprezentativni primer može se uzeti „activity on node“ (AON) mrežni dijagram gde su aktivnosti predstavljene kao čvorovi (pravougaonici ili krugovi), a odnosi među njima u vidu strelica.



Jedna od prednosti korišćenja mrežnog dijagrama jeste da pomaže menadžerima projekta da identifikuju kritične putanje u projektu. **Kritična putanja** je sekvenca aktivnosti koje moraju biti završene na vreme kako bi projekat bio završen prema planu. Osim toga, vizualizacijom toka realizacije projekta, uticaj promena na vremenski raspored postaje očigledniji. Na primer, ukoliko dođe do odlaganja jedne aktivnosti, mrežni dijagram može biti ažuriran kako bi se prilagodio novonastalim okolnostima i prikazivao uticaj promene na ukupno vreme trajanja projekta. Ukoliko za primer uzmemo aktivnosti A, B i C sa prikazanog dijagrama i pretpostavimo da postoji FS zavisnost između A i C, odnosno B i C (C ne može da počne dok se A i B ne završe), proširivanje mrežnog dijagrama dodavanjem informacija o trajanju aktivnosti omogućava nam identifikaciju kritičnih sekcija.



Na predstavljenom grafiku može se videti da aktivnost A i B započinju u isto vreme, pri čemu A traje šest jedinica vremena (dana, nedelja ...), dok je trajanje aktivnosti B četiri vremenske jedinice. Očigledno je da aktivnost B može da počne dve jedinice vremena kasnije u odnosu na aktivnost A, a da to ne utiče na trajanje projekta, jer će zavisna aktivnost C moći da otpočne u planirano vreme. Međutim, ukoliko bi se početak aktivnosti B pomerio za više od dve jedinice, to bi značilo da projekat mora da se produži za jednu vremensku jedinicu. Vremenski period za koji se može odložiti (najraniji) početak neke aktivnosti bez uticaja na naredne zavisne aktivnosti naziva se **slobodno odstupanje**. U slučaju aktivnosti B, slobodno odstupanje je dve jedinice vremena, dok aktivnost A nema slobodno odstupanje, odnosno, odstupanje je nula.



Ukupno odstupanje je vremenski period za koji se može pomeriti početak neke aktivnosti, a da to ne utiče na vreme završetka projekta. Iako to nije slučaj na prikazanim dijagramima, hajde da pretpostavimo situaciju da aktivnost C ima slobodno odstupanje od jedne vremenske jedinice, tj. završetak aktivnosti C bi bio jednu jedinicu ranije od završetka projekta. U tom slučaju, čak i ako aktivnost A nema slobodno odstupanje (jer ne možemo pomerati njen početak bez uticaja na početak zavisne aktivnosti C), obzirom da A može da „apsorbuje“ odlaganje od jedne jedinice koje potiče od aktivnosti C, to znači da aktivnost A ima ukupno odstupanje koje je jednako jednoj vremenskoj jedinici. U toj situaciji, aktivnost B bi imala ukupno odstupanje od tri vremenske jedinice (dve jedinice svog slobodnog odstupanja i jedna jedinica „apsorbovanog“ odstupanja od aktivnosti C).

Na osnovu rečenog, kritična putanja se može identifikovati kao skup svih aktivnosti kod kojih je ukupno odstupanje jednako nuli. Drugim rečima, svako odlaganje početka bilo koje aktivnosti bi dovelo do pomeranja vremena završetka projekta. Razumevanje šta je kritična putanja u planu omogućava menadžeru projekta da se fokusira na aktivnosti koje su najvažnije za održavanje projekta u skladu sa rasporedom. Važno je napomenuti da svaki plan ima kritičnu putanju.

4.2. GANTOV DIJAGRAM (GANTOGRAM)

Gantov dijagram je popularan alat za upravljanje projektima koji se koristi za praćenje i vizualizaciju napretka projekta tokom vremena. U osnovnoj verziji prikazuje datume početka i završetka svakog zadatka (task-a) u okviru aktivnosti, kao i zavisnosti između zadataka. Gantogrami se često koriste u upravljanju projektima kako bi se planirali, pratili i kontrolisali vremenski okviri projekta.

Gantogram se sastoji od horizontalne ose koja predstavlja vremenski tok projekta i vertikalne ose koja nabraja zadatke koji čine aktivnosti ili projekat. Svaki zadatak je predstavljen obojenom horizontalnom trakom čija dužina označava trajanje zadatka, od datuma početka do datuma završetka. Zavisnosti između zadataka označene su strelicama koje povezuju trake zadataka. Na ovaj način menadžeri projekta mogu da vide kako su zadaci međusobno povezani i identifikuju potencijalna kašnjenja u toku trajanja projekta.

Osim praćenja napretka pojedinačnih zadatak, Gantogrami se mogu koristiti i za dodelu resursa zadacima i za procenu troškova projekta. Kombinovanje ovih dodatnih informacija sa trajanjem i statusom zadatka (često označenog određenom bojom) omogućava da se ustanovi šta predstavljaju različite aktivnosti, koliko će koštati svaka aktivnost i ceo projekat, koje aktivnosti zavise od drugih i koje su kritične putanje projekta.

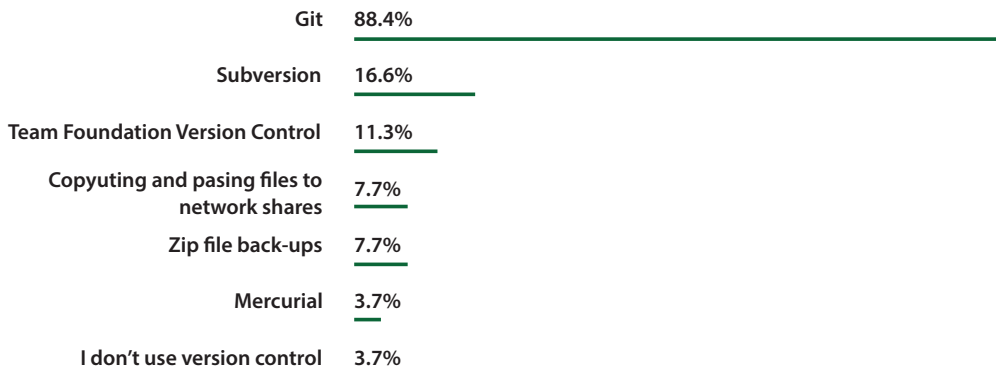
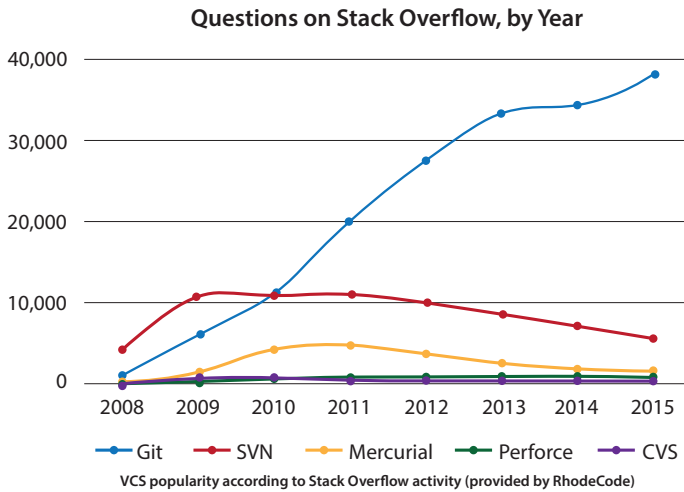
Na prikazanom primeru Gantovog dijagrama možemo videti tri aktivnosti koje počinju u različito vreme, pri čemu postoji FS zavisnost između aktivnosti A1 i A2 takva da A2 ne može da počne dok se aktivnost A1 ne završi. Prvobitno je planirano da realizacija aktivnosti A1 započne 29. decembra i da traje dva dana, zaključno sa 30. decembrom, što je predstavljeno trakom sa oznakom „A1 planirano“. Po završetku rada na aktivnosti A1 započinje aktivnost A2 koja traje tri dana, do 2. januara, kada je po planu trebalo da obe aktivnosti budu završene, što prikazuje traka „A2 planirano“. Ukupno vreme predviđeno za obe aktivnosti je pet dana, od 29. decembra do 1. januara. Međutim, zbog nepredviđenih okolnosti, rad na aktivnosti A1 je produžen na četiri dana (traka A1), što zbog postojanja FS zavisnosti nužno utiče na početak realizacije aktivnosti A2 (traka A2). Usled neplaniranog kašnjenja, razvojni tim se dodatno angažovao (radio prekovremeno, uposlio dodatne članove) i aktivnost A2 završio u kraćem intervalu od planiranog, za dva dana, umesto za tri. Na ovaj način je delimično kompenzovano kašnjenje završetka aktivnosti A1 i ukupno vreme rada na aktivnostima A1 i A2 umesto planiranih pet, iznosi šest dana. Na kraju, aktivnost A3 je nezavisna od A1 i A2 tako da njihovo pomeranje ne utiče na vreme njene realizacije.

Gantogrami se često generišu korišćenjem specijalizovanih softvera za upravljanje projektima. Osim dizajniranja grafikona, ovi alati omogućavaju dodavanje drugih članova tima i definisanje njihovih zaduženja u okviru projekta, kao i jednostavniju komunikaciju i informisanje.

4.3. GIT

Git predstavlja najpopularniji distribuirani sistem za kontrolu verzija (**Distributed Version Control System**) koji se koristi za praćenje promena u izvornom kodu tokom razvoja softverskih projekata. Kreirao ga je Linus Torvalds (tvorac Linuksa) i od tada je postao standard za kontrolu verzija u softverskoj industriji. Git je besplatan i otvorenog koda, a dizajniran je za efikasan rad na svim projektima bez obzira na njihovu veličinu i složenost. Git je ključan alat u timskom razvoju softvera, jer omogućava da više članova tima efikasno radi na istom projektu.

Iako jeste najpopularniji, Git nije jedini sistem za upravljanje verzijama. Kao drugi primeri se mogu navesti Apache-ov Subversion, Mercurial i Perforce, čije je ukupno učešće na tržištu upravljanja izvornim kodom značajno manje od Git-ovog.



Stack Overflow's 2018 Developer Survey

Uopšteno govoreći, svi sistemi za upravljanje verzijama omogućavaju korisnicima da posećuju ranije verzije fajlova, upoređuju promene između verzija, vraćaju raniji sadržaj fajlova i dele promene sa drugim članovima tima. Korisnici koji ne koriste VCS-e su primorani da primenjuju druge metode za praćenje verzija. Jedan od načina jeste arhiviranje fajlova projekta i imenovanje arhiva u odnosu na verziju i vreme pravljenja, npr. `project1_01012023.zip` ili `project1_v1.1.zip` i sl. Ovakav sistem nije pogodan u slučaju saradnje više članova tima na istom projektu, kada se deli veliki broj fajlova i kada su izmene koje se vrše u njima paralelne (istovremene).

4.3.1. Instalacija Git-a

Git je softver za kontrolu verzija koji se pokreće na lokalnom računaru. Ne postoji potreba za otvaranjem naloga i nije neophodna konekcija sa Internetom. Za razliku od Git-a, Github (o kome će biti reči kasnije) je servis koji hostuje Git repozitorijume u distribuiranom okruženju (u kladu) i time olakšava saradnju sa drugim ljudima na projektu. Za korišćenje Github-a se mora otvoriti nalog. Git je primarno terminalni alat, ali postoje i grafička okruženja za Git koje vredi pomenuti, kao što su GitKraken, SourceTree ili Github Desktop. Prednosti korišćenja terminalnog okruženja uključuju dostupnost onlajn dokumentacije i resursa, brzinu izvršavanja komandi, mogućnost upotrebe naprednijih opcija koje nisu dostupne kroz klasično vizualno okruženje i uniformnost korišćenja bez obzira na operativni sistem. Sa druge strane, GUI alati su jednostavniji za početnike i omogućavaju vizualizaciju strukture projekta i promena u verzijama.

Pošto je Git dizajniran da radi na Linuks operativnim sistemima, a Windows operativni sistemi imaju sopstveni komandni interfejs, potrebno je instalirati **Git Bash** pre nego što Git može da se pokrene na Windows okruženju. Git Bash emulira Unix bash shell na Windows-u i omogućava korišćenje standardnih operacija Git alata. Iz **Downloads** sekcije zvaničnog Git sajta¹⁰ mogu se skinuti instalacioni paketi za Linux/Unix, Windows i macOS. Prilikom instalacije Windows verzije preporučljivo je odabrati „Notepad++“ ili „Visual Studio Code“ kao podrazumevani editor za Git, a zatim i instalirati odabrani softver. Ostale opcije se mogu ostaviti sa podrazumevanim vrednostima. Na Debian baziranim Linuks operativnim sistemima, Git se najjednostavnije instalira korišćenjem APT menadžera paketa (RPM na Redhat Linuks distribucijama):

```
# apt install git
```

Iako Git ne zahteva registraciju naloga, za uspešno korišćenje neophodno je konfigurisati korisničko ime i email adresu kako bi alat mogao da razlikuje korisnike koji unose promene u fajlovima i verzijama:

```
# git config --global user.name "Milos Antonijevic"
# git config --global user.email "mantonijevic@singidunum.ac.rs"
```

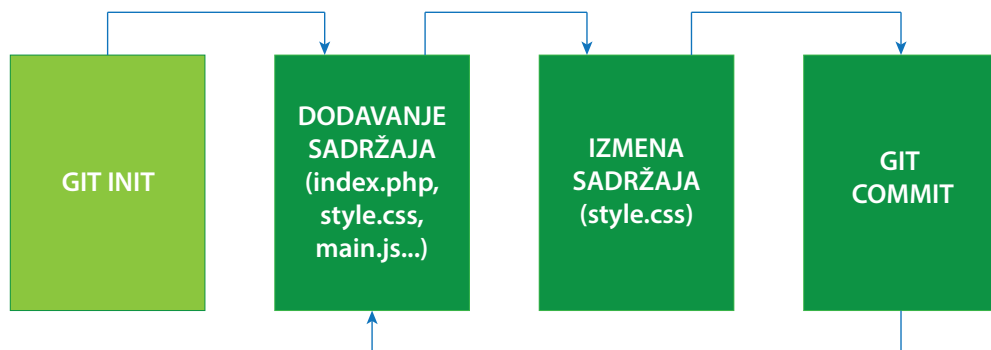
Poželjno je da korisničko ime i email adresa budu konzistentni u svim projektima gde se Git koristi i da odgovaraju podacima koji će biti uneti prilikom otvaranja GitHub onlajn naloga.

¹⁰ (<https://git-scm.com/>)

4.3.2. Osnovni pojmovi i koncepti

Git repozitorijum je radni prostor koji prati promene i upravlja fajlovima u okviru direktorijuma. Svaki projekat zahteva kreiranje novog, nezavisnog Git repozitorijuma sa sopstvenim sadržajem. Repozitorijum prati promene u fajlovima tokom vremena održavanjem istorije fajlova koja omogućava korisnicima da vide ko je izvršio izmene i kad. Fajlovi koje Git može da prati podrazumevaju izvorni kod, konfiguracione fajlove, dokumentaciju i bilo koju drugu komponentu koja je vezana za projekat, odnosno koja se nalazi u okviru direktorijuma Git repozitorijuma.

Git „komitovanje“ (committing) je proces pravljenja preseka Git repozitorijuma u datom vremenskom trenutku (slično snimanju statusa igrice). Git snima stanje svih fajlova i foldera u okviru repozitorijuma. Komitovanje predstavlja fundamentalnu operaciju u okviru Git alata i ima ključnu ulogu u praćenju istorije razvoja projekta. Standardni tok rada u Git-u podrazumeva kreiranje sadržaja, dodavanje izmena i komitovanje.



Pre korišćenja Git komandi potrebno je inicijalizovati projektni Git repozitorijum. Komanda za inicijalizovanje repozitorijuma je **git init**. Izvršava se samo jednom, nakon kreiranja projekta, unutar korenog direktorijuma projekta. Ova komanda će kreirati skriveni direktorijum **.git** koji sadrži sve što je neophodno kako bi Git alat čuvao stanje i istoriju razvoja projekta.

```
$ cd .git
$ ls -l
total 32
drwxr-xr-x 2 debian debian 4096 Jan 3 15:22 branches
-rw-r--r-- 1 debian debian  92 Jan 3 15:22 config
-rw-r--r-- 1 debian debian  73 Jan 3 15:22 description
-rw-r--r-- 1 debian debian  23 Jan 3 15:22 HEAD
drwxr-xr-x 2 debian debian 4096 Jan 3 15:22 hooks
drwxr-xr-x 2 debian debian 4096 Jan 3 15:22 info
drwxr-xr-x 4 debian debian 4096 Jan 3 15:22 objects
drwxr-xr-x 4 debian debian 4096 Jan 3 15:22 refs
```

Git će pratiti sve promene, uključujući kreiranje, izmenu i brisanje fajlova u projektnom direktorijumu i svim njegovim poddirektorijumima. Komanda **git status** pruža informacije o trenutnom stanju Git repozitorijuma. U zavisnosti od sadržaja radnog direktorijuma (**working directory**), pripremnog prostora (**staging area**) i samog repozitorijuma komanda će vratiti različite izlaze:

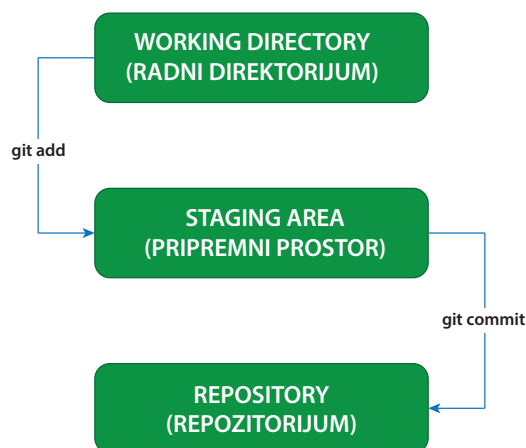
```
$ git status
```

```
On branch master
```

```
No commits yet
```

```
nothing to commit (create/copy files and use "git add" to track)
```

Git alat upravlja projektom koristeći koncept tri različita „prostora“ (područja). Prvi prostor se naziva „radni direktorijum“ i obuhvata celinu u okviru fajl sistema, odnosno direktorijum projekta, gde se odvijaju promene u fajlovima, kreiranje, izmena, snimanje i brisanje sadržaja direktorijuma ili konkretnih fajlova. Drugi prostor predstavlja „pripremní prostor“ (ili područje za pripremu), u okviru koga se može izvršiti grupisanje napravljenih izmena kao priprema za komitovanje ili pravljenje *ček pointa* projekta u vremenu. Ne moraju sve napravljene izmene u radnom direktorijumu da uđu u pripremní prostor ili komit, kao što ni svaka pojedinačna izmena ne mora da se komituje.



Treći prostor predstavlja sam „komit“ ili trenutni presek stanja projekta u vremenu (checkpoint, snapshot, save) koji u repozitorijum (.git direktorijum) upisuje izmene koje se u datom trenutku nalaze u pripremnom prostoru. Komitovi predstavljaju osnovne gradivne jedinice Git repozitorijuma koje se mogu međusobno upoređivati i na koje se korisnik može po potrebi vraćati.

4.3.3. Dodavanje izmena u pripremni prostor

Kao što je prethodno objašnjeno, Git omogućava grupisanje izmena u okviru projekta sa ciljem pravljenja preseka stanja projekta ili komita. Ovaj proces započinje dodavanjem željenih promena u pripremni prostor. Korišćenjem pomenute Git **status** komande, u svakom trenutku možemo dobiti informacije o promenama u okviru radnog direktorijuma i pripremnog prostora. U odnosu na prvobitno pokretanje komande, neposredno nakon inicijalizacije repozitorijuma, kada smo dobili informaciju da ne postoje ništa što je podložno komitovanju, ukoliko napravimo neku izmenu, npr. kreiramo dva fajla, izlaz će izgledati drugačije:

```
$ echo "Ovo je naslov novog rada u casopisu" > manuscript.txt
$ echo "Ovo je spisak relevantne literature" > references.txt
$ git status
On branch master
```

```

No commits yet
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    manuscript.txt
    references.txt
nothing added to commit but untracked files present (use "git
add" to track)

```

Kao što se može videti, Git je „primetio“ da postoje dva nova fajla (`manuscript.txt` i `references.txt`), koja se do ovog trenutka ne prate (Untracked files) i o tome nas obavestio. Dodavanje ove dve izmene u pripremni prostor se obavlja upotrebom **git add** komande:

```

$ git add references.txt manuscript.txt
ili
$ git add .

```

, u slučaju da u pripremni prostor želimo da dodamo sve izmene koje su do datog trenutka napravljene u radnom direktorijumu.

4.3.4. Komitovanje (potvrđivanje) izmena (u repozitorijum)

Svaki put kada se izmene iz pripremnog prostora potvrđuju u repozitorijum, odnosno kada se kreira novi komit, Git zahteva od korisnika da unese poruku koja taj komit opisuje. Komitovanje se obavlja korišćenjem **git commit** komande koja otvara podrazumevani editor teksta i očekuje unosenje komit poruke. Umesto toga, može se koristiti opcija `-m` uz `git commit`, nakon koje se pod navodnicima navodi tekst komit poruke.

```

$ git commit -m "kreirani fajlovi manuscript.txt i references.txt"
[master (root-commit) dd20d20] kreirani fajlovi manuscript.txt
i references.txt
2 files changed, 2 insertions(+)
create mode 100644 references.txt
create mode 100644 manuscript.txt

```

Prilikom pravljenja komita treba voditi računa da izmene koje ulaze u komit čine smislenu celinu, u slučaju softvera jednu funkcionalnost ili ispravku propusta. Komit poruke treba da budu koncizne i po konvenciji se pišu u sadašnjem vremenu, npr. „kreiram fajlove ...“, „menjam stil stranice ...“. Ovo pravilo nije

striktno, ali je poželjno da stil poruka bude konzistentan kroz sve komitove, odnosno ukoliko je opredeljenje da se koristi sadašnje vreme, sve poruke bi trebalo pisati na taj način. Stil komit poruka treba da bude jedinstven na nivou celog projekta i ispoštovan od strane svih članova razvojnog tima. Ažuriranje **poslednjeg** komita, što podrazumeva dodavanje ili brisanje izmena koje će se nalaziti u okviru komita i promenu komit poruke, može se izvršiti korišćenjem **--amend** opcije (git commit --amend).

Git omogućava da se određeni fajlovi (ili direktorijumi) isključe iz repozitorijuma vezanog za konkretan projekat. U ovom slučaju Git neće pratiti izmene sadržaja i statusa fajla, niti će pratiti promene u direktorijumima koji su izuzeti. Najčešći primeri kada se primenjuje navedena opcija jesu slučajevi fajlova sa kredencijalima i API ključevima koji su privatne prirode i nisu vezani za projekat, već za konkretnog korisnika. Sistemski fajlovi operativnog sistema i log fajlovi u najvećem broju slučajeva takođe mogu biti izuzeti iz repozitorijuma čak i ako se nalaze u projektnom direktorijumu. Na kraju, eksterni paketi koji ne predstavljaju integralni deo samog projekta i koji se mogu nezavisno instalirati i naknadno pridružiti projektu ne moraju da se nalaze u projektnom repozitorijumu. Lista fajlova i direktorijuma koji predstavljaju izuzetke iz repozitorijuma i koji će biti ignorisani može se navesti u okviru skrivenog fajla **.gitignore**.

4.3.5. Upravljanje granama repozitorijuma

Git komanda koja se koristi za prikazivanje istorije komitova, zajedno sa relevantnim informacijama kao što su autor, datum, i poruka komita je **git log**. U svom osnovnom obliku, komanda daje izlaz sličan prikazanom:

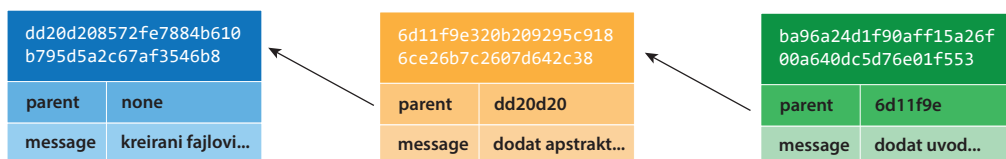
```
$ git log
commit 6d11f9e320b209295c9186ce26b7c2607d642c38 (HEAD -> master)
Author: Milos Antonijevic <mantonijevic@singidunum.ac.rs>
Date:   Fri Jan 6 10:18:08 2024 -0600

    dodat apstrakt u fajl manuscript.txt

commit dd20d208572fe7884b610b795d5a2c67af3546b8
Author: Debian <debian@debian10.linuxvmimages.local>
Date:   Fri Jan 5 09:35:16 2024 -0600

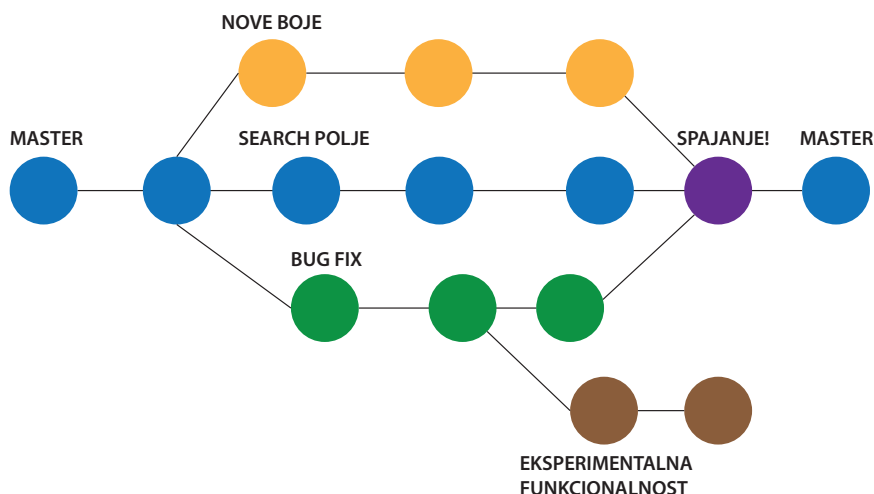
    kreirani fajlovi manuscript.txt i references.txt
```

Iz navedenog primera se može videti da u okviru repozitorijuma postoje dva komita: jedan kreiran 5. januara od strane „Debian“ korisnika i drugi od 6. januara od strane korisnika „Milos Antonijevic“. U oba slučaja, Pored reči „commit“ se nalazi jedinstveni heš kod (eng. *hash code*) koji odgovara konkretnom komitu i koji predstavlja njegovu jedinstvenu odrednicu. Ispod osnovnih informacija se nalazi prikazana komit poruka koja je uneta prilikom njegovog kreiranja.



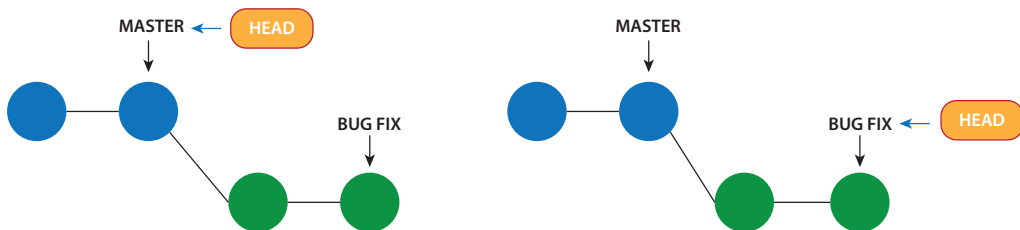
Na prikazanom dijagramu predstavljen je model upravljanja sa komitovima koji Git alat koristi kako bi pratio istoriju promena. Svaki komit, osim prvog, referencira na najmanje jedan komit koji predstavlja njegovog roditelja čime se formira linearna istorija promena. Nedostatak koji se javlja u slučaju linearne istorije je nemogućnost istovremenog rada na različitim sadržajima projekta koji potencijalno utiču jedni na druge. Na primer, ukoliko je dizajner dobio zadatak da promeni oblik i boju dugmića na glavnoj stranici veb aplikacije, jedan od programera radi na ispravljanju pronađenog propusta, a drugi želi da implementira polje za pretragu u okviru glavne stranice, pomenute izmene u kodu mogu dovesti do konfliktne situacije kada dvoje članova tima želi da promeni iste fajlove ili funkcionalnosti. Kako bi se rešio pomenuti problem, u Git alat je inkorporiran koncept **grana**.

Grane (eng. *branches*) predstavljaju paralelne sadržajne celine repozitorijuma projekta, koje se mogu kreirati, nezavisno razvijati i međusobno spajati u nekom trenutku. Može se napraviti paralela između grana i vremenskih linija projekta. Grane su nezavisne jedna od druge tako da promene koje se naprave u okviru jedne grane ne utiču na druge grane sve do trenutka njihovog potencijalnog spajanja.

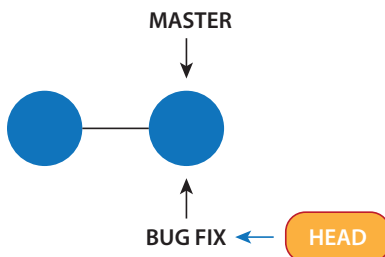


Početna, glavna grana repozitorijuma podrazumevano nosi naziv „master“ (raniji izlaz git status komande je sadržao odrednicu On branch **master**). Iako predstavlja početnu granu projekta koja se inicijalizuje izvršavanjem git init komande, „master“ grana se ni po čemu ne razlikuje od ostalih grana u okviru projekta. Uprkos tome, veliki broj programera tretira „master“ granu kao „zvaničnu granu“ koja sadrži glavnu verziju projekta, u koju se u nekom momentu stapaju sve ostale grane za koje je odlučeno da treba da pripadaju projektu. Osnovna logika iz ovakvog pristupa jeste da se na „master“ grani ne rade nikakve izmene, osim da se sa njom (u nju) spajaju sve ostale grane na kojima su radili članovi razvojnog tima. GitHub je 2020. godine zabranio da glavna grana nosi naziv „master“, pa se u situacijama kada projekat treba distribuirati putem GitHub platforme koristi neko od alternativnih imena, npr. „main“ ili „trunk“.

Poslednji koncept koji je neophodno razmotriti pre početka rada sa granama jeste koncept HEAD-a ili glave repozitorijuma. Odrednicu HEAD smo ranije mogli videti prilikom korišćenja git log komande kod prikaza podataka o poslednjem komitu: `commit 6d11f9e320b209295c9186ce26b7c2607d642c38 (HEAD -> master)`. Glava (HEAD) repozitorijuma predstavlja pokazivač (eng. *pointer*) na trenutnu lokaciju u repozitorijumu. Najčešće, HEAD upućuje na poslednji komit koji je napravljen u trenutno aktivnoj grani. Kada se kreira nova grana, njen početak se nalazi na mestu na koje HEAD upućuje. Precizna terminologija bi glasila, HEAD podrazumevano upućuje na referencu trenutno aktivne grane, a referenca ili pokazivač grane upućuje na poslednji komit konkretne grane. Kao što ćemo kasnije pokazati, HEAD možemo pomerati napred i nazad po tačkama vremenske linije grane, tj. može „skočiti“ na bilo koji odabrani komit.



Na prikazanom dijagramu mogu se videti dve situacije: prva, kada HEAD upućuje na referencu „master” grane, tj. na poslednji komit „master” grane i druga, kada HEAD upućuje na referencu grane „bug fix”. Kada se vrši prebacivanje aktivnosti sa jedne grane na drugu, HEAD se automatski pozicionira na pokazivač novoizabrane (aktivne) grane. Neposredno nakon kreiranja nove grane imamo situaciju da pokazivači obe grane, „roditeljske” grane i grane „deteta”, upućuju na isti komit, dok HEAD može referencirati samo na jedan od dva pokazivača grane u zavisnosti od toga koja grana je trenutno odabrana (aktivna). Razlog ovome je što po kreiranju nove grane poslednji komit roditeljske grane istovremeno predstavlja poslednji (i jedini) komit grane deteta, pa oba pointera grana upućuju na njega. Sa druge strane, u svakom datom trenutku samo jedna od dve grane može biti aktivna i HEAD sadrži referencu na njen pokazivač.



Pregled postojećih grana u repozitorijumu se obavlja izvršavanjem **git branch** komande. Karakter zvezdica * u izlazu komande ukazuje na trenutno aktivnu granu:

```
$ git branch
  bug_fix
  nove_boje
* master
```

U prikazanom izlazu komande se mogu videti tri postojeće grane: `bug_fix`, `nove_boje` i `master`, pri čemu je „`master`” grana trenutno aktivna, tj. na nju upućuje HEAD. Ukoliko se `git branch` komandi doda argument koji predstavlja naziv nove grane, Git će kreirati navedenu granu i dodati je repozitorijumu. Početak nove grane je komit na koji HEAD upućuje u trenutku njenog kreiranja.

U verzijama Git-a počev od 2.23.0 promena aktivne grane se vrši korišćenjem komande `git switch <naziv-grane>`, dok se u ranijim verzijama upotrebljavala `git checkout` naredba:

```
$ git branch nova_grana
$ git branch
* master
  nova_grana
$ git switch nova_grana
Switched to branch 'nova_grana'
$ git branch
  master
* nova_grana
$ git log --oneline
ba96a24 (HEAD -> nova_grana, master) dodat uvod u fajl manuscript.
txt
6d11f9e dodat apstrakt u fajl manuscript.txt
dd20d20 kreirani fajlovi manuscript.txt i references.txt
```

Izvršavanjem navedenih komandi kreirana je grana pod nazivom „`nova_grana`”, prebačen je HEAD pokazivač na novokreiranu granu i prikazana skraćena istorija komitova korišćenjem `git log` komande sa opcijom **--one-line**. Iz prikazanog primera se može videti da se kreiranjem nove grane ne obavlja automatsko prebacivanje na nju, tj. HEAD ostaje na roditeljskoj grani (* stoji uz „`master`” granu). Osim toga, treba uočiti pomenutu situaciju (prikazanu na prethodnom dijagramu) kada (poslednji) komit koji počinje hešom **ba96a24** predstavlja pokazivač (referencu) za obe grane - „`master`” i „`nova_grana`”, pri čemu HEAD upućuje na pointer grane „`nova_grana`” (`ba96a24 (HEAD -> nova_grana, master)`).

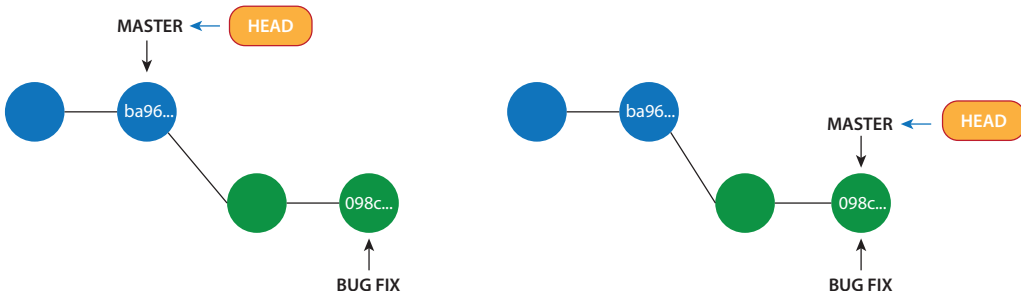
Brisanje grane iz repozitorijuma se obavlja izvršavanjem `git branch` komande sa opcijom **-d**. Grana se može izbrisati samo iz neke druge grane i to podrazumevano samo ukoliko je grana spojena sa drugom granom:

```
$ git branch -d stara_grana
error: The branch 'stara_grana' is not fully merged.
If you are sure you want to delete it, run 'git branch -D
eksperimentalna'.
```

Koncept grana u osnovi omogućava nezavistan rad na različitim aspektima projekta. Međutim, na kraju razvojnog ciklusa se javlja potreba za kombinovanjem svih implementiranih funkcionalnosti, sadržaja i dizajna u jedinstvenu celinu, što se postiže mehanizmom spajanja (**git merge**) grana. Na ovaj način, u toku trajanja projekta svako od članova razvojnog tima može raditi na svom delu posla, pri čemu će se kao finalni rezultat dobiti jedinstveni projekat.

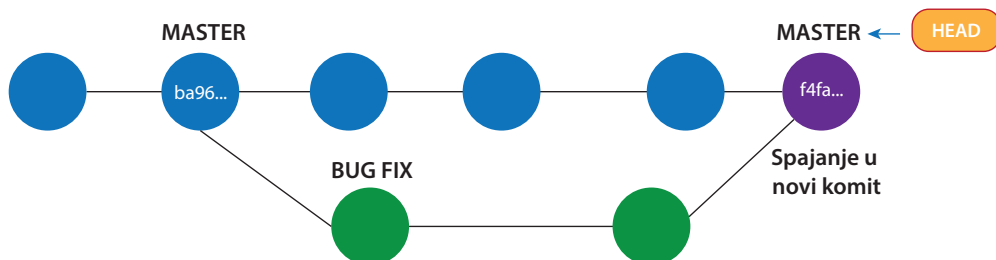
Spajanje grana zapravo predstavlja spajanje izmena iz odabrane grane sa izmenama u trenutno aktivnoj grani, tj. grani na koju upućuje HEAD. Iz ovoga proizilazi da je, pre započinjanja procesa spajanja, neophodno prebaciti se na granu kojoj se žele pridružiti izmene. U zavisnosti od vrste izmena koje su u granama napravljene, postoje tri moguća tipa spajanja: **fast-forward** spajanje, automatsko spajanje i „konfliktno“ (ručno) spajanje.

U slučaju da grana sa kojom se obavlja spajanje nema komite koju su napravljeni u međuvremenu, primarna grana jednostavno „sustigne“ sekundarnu granu (pokazivači na obe grane postanu isti, odnosno ukazuju na poslednji komit sekundarne grane) kao što je prikazano na narednom dijagramu.



Na levoj strani dijagrama se može videti da je od komita čiji heš počinje sa ba96 napravljena nova grana „bug_fix“. Nakon toga su u grani „bug_fix“ napravljena dva komita koja sadrže izmene koda koje za cilj imaju ispravljanje pronađenog propusta iz „master“ grane projekta. S obzirom da u primarnoj grani u međuvremenu nisu napravljeni novi komitovi, prilikom spajanja grana, Git će jednostavno pomeriti pokazivač grane „master“ na isto mesto na koje upućuje pokazivač grane „bug_fix“, čime će grana „master“ obuhvatiti sve izmene koje su napravljene u grani „bug_fix“.

U praksi je češća situacija kada se paralelno radi na više grana od strane različitih članova tima. U tom slučaju primarna grana sadrži komitove koji se ne nalaze u sekundarnoj grani, a sekundarna grana sadrži nove izmene koje želimo da pridružimo primarnoj grani, kao što je prikazano na sledećem dijagramu.



U zavisnosti od toga da li su izmene koje su napravljene u jednoj grani u konfliktu sa izmenama koje postoje u drugoj, Git može automatski izvršiti spajanje ili spajanje može prepustiti korisniku. Na primer, ukoliko su nakon kreiranja nove grane (posle komita ba96...), u okviru grane „master“ napravljena dva nova fajla, izmene u grani „bug_fix“ se ne mogu odnositi na ta dva nova fajla, s obzirom da oni u njoj ne postoje. Dakle, izmene u vidu dva fajla u grani „master“, pod uslovom da su to jedine izmene, nisu u sukobu sa izmenama u grani „bug_fix“ i u tom slučaju Git može automatski da obavi spajanje grana tako što će kreirati novi komit u primarnoj grani i taj komit će sadržati sve izmene iz sekundarne grane:

```
$ git switch master
Switched to branch 'master'
$ git merge bug_fix
Merge made by the 'recursive' strategy.
 manuscript.txt | 2 ++
 1 file changed, 2 insertions(+)
$ git log --oneline
128294a (HEAD -> master) Merge branch 'bug_fix'
3a32fed dodata dva fajla
43bcefe (bug_fix) ispravljen propust
ba96a24 dodata tamna tema
6d11f9e dodato search polje
dd20d20 kreirana login stranica
```

Međutim, ako pretpostavimo da je član tima koji radi u okviru grane „master“ u *index.php* fajl dodao novi kod kako bi implemetirao mogućnost sortiranja rezultata pretrage, a programer koji radi na ispravci propusta u istom fajlu izmenio više linija teksta, u slučaju spajanja grana, postavlja se pitanje koja verzija fajla je odgovarajuća da bude zadržana u novom komitu. Ovo je situacija kada Git ne može samostalno da donese odluku i prepušta korisniku koji radi spajanje da odluči o sadržaju fajla tako što će ručno odabrati željene izmene koje će se nalaziti u novoj verziji fajla. Proces ručnog rešavanja konflikta se sastoji iz nekoliko koraka: otvaranje i izmena svakog fajla kod koga postoji konflikt tako što će se zadržati željeni sadržaj uz uklanjanje generisanih markera (<<<<<<, =====, >>>>>>), a zatim dodavanje izmena u pripremni prostor i njihovo komitovanje.

4.3.6. Vraćanje prethodnih verzija i putovanje kroz vreme

Kao što je ranije pomenuto, istorija komitova u okviru projektnog Git repozitorijuma se može videti upotrebom `git log` komande. Izlaz koji komanda `log` prikazuje između ostalog sadrži jedinstveni identifikator svakog komita u obliku heš koda (npr. 6d11f9e320b209295c9186ce26b7c2607d642c38). Iako heš komita sadrži 40 karaktera, kako bi se identifikovao određeni komit dovoljno je koristiti prvih 7 karaktera (npr. 6d11f9e). U slučaju potrebe za prelaskom na neki određeni komit koristi se komanda **`git checkout <heš komita>`**.

```
$ git checkout ba96a24
```

```
Note: checking out 'ba96a24'.
```

```
You are in 'detached HEAD' state. You can look around, make
experimental changes and commit them, and you can discard any
commits you make in this state without impacting any branches
by switching back to a branch.
```

```
If you want to create a new branch to retain commits you cre-
ate, you may do so (now or later) by using -b with the switch
command. Example:
```

```
git switch -b <new-branch-name>
```

```
Or to undo this operation with:
```

```
git switch -
```

```
Turn off this advice by setting config variable advice.deattachedHead
to false
```

```
HEAD is now at ba96a24 dodat uvod u fajl manuscript.txt
```

Iako sadržaj poruke koju komanda prikazuje može izgledati kao greška, 'detached HEAD' state zapravo predstavlja potpuno legitimno stanje u kome se repozitorijum može nalaziti. Pre svega treba napomenuti da je komanda uspešno izvršena, Git je skočio na željeni komit, što se može videti iz poslednje linije izlaza HEAD is now at ba96a24 dodat uvod u fajl manuscript.txt ili korišćenjem git log komande pri čemu će biti prikazana istorija samo onih komitova koji su nastali pre odabranog komita. Naime, kao što je objašnjeno u prethodnom poglavlju, HEAD pokazivač obično upućuje na referencu grane, a ne neki određen komit. Ako bismo pogledali sadržaj fajla **HEAD** koji se nalazi u **.git** direktorijumu, u zavisnosti od grane u kojoj se nalazimo, u najvećem broju slučajeva bismo videli sledeće:

```
$ cat .git/HEAD
ref: refs/heads/master
```

Sa druge strane, referenca (pokazivač) grane upućuje na poslednji komit koji je u toj konkretnoj grani napravljen.

```
$ cat .git/refs/heads/master
128294a0c280404b0287b5ad663cf78dacac6ea5
```

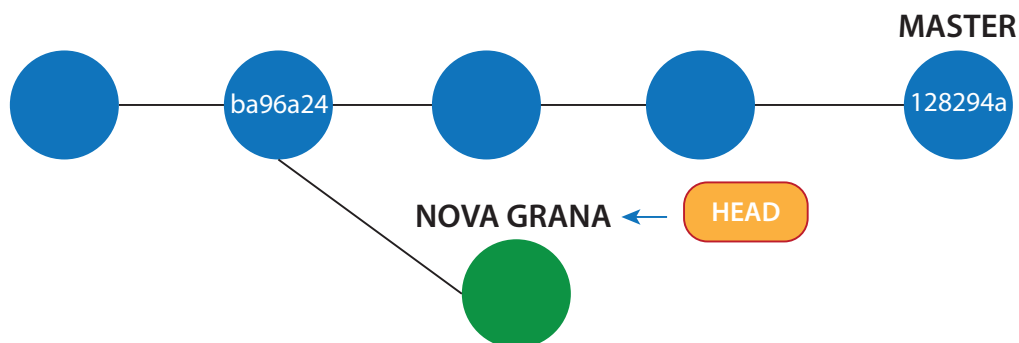
Razlog prikazivanja prethodnog obaveštenja prilikom korišćenja checkout komande koje sadrži 'detached HEAD' state frazu jeste taj što u situaciji nakon prelaska na neki određen komit, HEAD pokazivač zapravo upućuje na taj komit (umesto na referencu grane). U ovom slučaju HEAD je u „otkačenom“ (eng. *detached*) stanju u odnosu na referencu grane.

```
$ git checkout ba96a24
...
$ cat .git/HEAD
ba96a24d1f90aff15a26f00a640dc5d76e01f553
```

Kada se repozitorijum nalazi u „detached HEAD“ stanju, korisniku je na raspolaganju nekoliko mogućih operacija:

1. Pregledanje (ispitivanje) sadržaja odabranog komita,
2. Vraćanje HEAD pokazivača na referencu grane,
3. Kreiranje nove grane i prebacivanje na nju.

Važno je napomenuti da git checkout komanda omogućava pregledanje sadržaja komitova, odnosno verzija projekta, iz uloge **posmatrača**, jer kad je HEAD otkaćen od reference grane, izmene sadržaja nisu moguće. Jedini način za vršenje izmena jeste kreiranje nove grane i nastavak rada u okviru nje, jer u tom slučaju HEAD upućuje na referencu grane koja je kreirana.



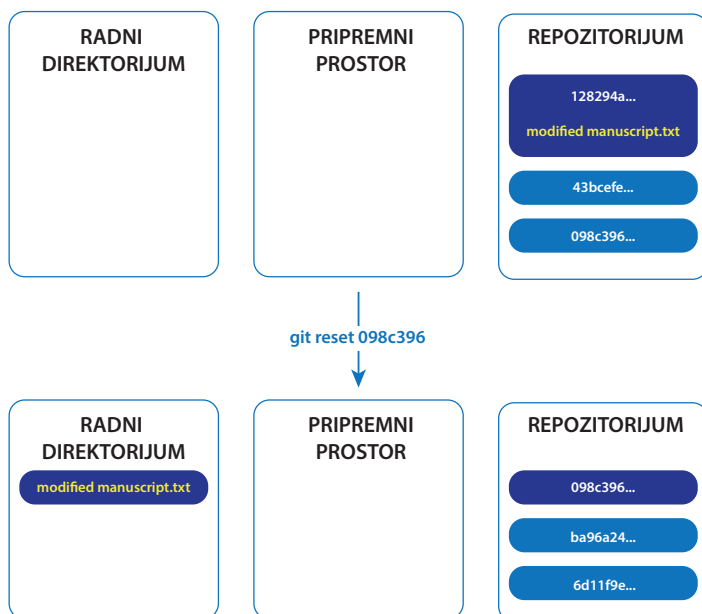
U slučaju druge opcije, ukoliko se želi vraćanje HEAD pokazivača na podrazumevanu poziciju gde upućuje na referencu grane, dovoljno je prebaciti se na željenu granu korišćenjem `git switch` komande.

Osim pregledanja sadržaja prethodnih komitova, `git checkout` komanda se može koristiti za vraćanje sadržaja fajlova na stanje koje se nalazi u poslednjem napravljenom komitu. Na primer, pretpostavimo da smo kreirali fajl `style.css`, da smo uneli neke css stilove u fajl i napravili komit. Nakon toga, nastavili smo da menjamo `style.css` fajl kako bismo implementirali stilove za nove HTML elemente na stranici i promenili izgled nekih postojećih elemenata. Međutim, rukovodiocu projekta se nisu dopale napravljene izmene i želi da se izgled elemenata vrati na prethodno stanje. Pošto novi komit još uvek nije napravljen, izmene koje smo uneli u fajl možemo poništiti vraćanjem sadržaja fajla na verziju koja postoji u poslednjem komitu korišćenjem `git checkout` komande na sledeći način:

```
$ git checkout HEAD style.css
```

Uopšteno govoreći, za vraćanje sadržaja fajla na stanje iz proizvoljnog komita, sintaksa koja se koristi je `git checkout <heš komita> <fajl(ovi)>`.

Za razliku od prethodne primene naredbe `checkout`, koja omogućava vraćanje sadržaja pojedinačnih fajlova na stanje iz prethodnog komita, vraćanje stanja celog repozitorijuma na određeni komit se obavlja uz pomoć `git reset` komande. Izvršavanje ove komande pomera pokazivač grane unazad do navedenog komita, pri čemu briše sve komitove koji se nalaze nakon njega. Ukoliko se `git reset` koristi bez dodatnih argumenata (osim obaveznog navođenja heša komita na koji se želi povratak), može se govoriti o osnovnoj primeni komande. U ovom slučaju, iako su prateći komitovi obrisani, trenutne izmene (trenutni sadržaj) koje su napravljene u fajlovima ostaju u radnom direktorijumu.



Ovo je korisno u situacijama kada su komitovi bili napravljeni u pogrešnoj grani ili ako se želi postići sinteza izmena iz više komitova u jedinstveni novi komit.

```
$ git log --oneline
```

```
128294a (HEAD -> master) dodati opis eksperimenta
```

```
43bcefe dodati zaključak
```

```
098c396 dodate 2 revizije teksta
```

```
ba96a24 dodati uvod u fajl manuscript.txt
```

```
6d11f9e dodati apstrakt u fajl manuscript.txt
```

```
dd20d20 kreirani fajlovi manuscript.txt i references.txt
```

```
$ git reset 098c396
```

```
Unstaged changes after reset:
```

```
M manuscript.txt
```

```
$ git log --oneline
```

```
098c396 (HEAD -> master) dodate 2 revizije teksta
```

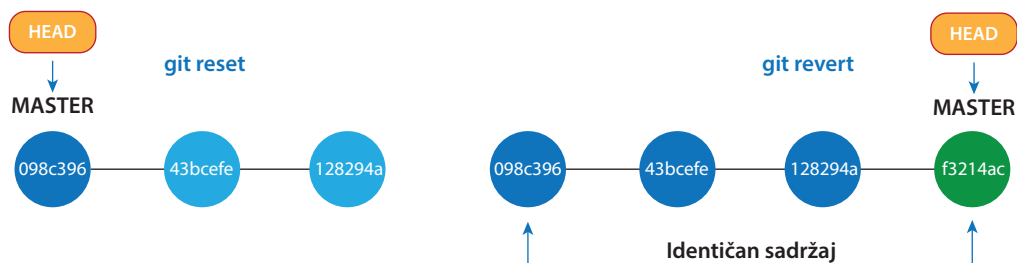
```
ba96a24 dodati uvod u fajl manuscript.txt
```

```
6d11f9e dodati apstrakt u fajl manuscript.txt
```

```
dd20d20 kreirani fajlovi manuscript.txt i references.txt
```

Treba imati u vidu da `git reset` utiče samo na komitove iz trenutno aktivne grane, ostale grane i njihov sadržaj ostaju netaknuti. Ukoliko smo sigurni da želimo u potpunosti da obrišemo izmene u fajlovima, komanda se može izvršiti dodavanjem argumenta `--hard`.

Drugi način za vraćanje sadržaja repozitorijuma na stanje iz prethodnog komita jeste korišćenje **git revert** komande. Za razliku od **reset** komande koja pomera pokazivač grane unazad pri tome brišući usputne komitove, naredba **revert** kreira potpuno novi komit koji sadrži stanje u kome su fajlovi bili u nekom ranijem komitu. Oba navedena procesa su prikazana na narednom dijagramu.



Iz prikazanog se može zaključiti da **reset** briše istoriju izmena i vraća stanje repozitorijuma unazad, a **revert** čuva sve ranije izmene i postavlja stanje repozitorijuma na raniji sadržaj. Važno je naglasiti značajnu razliku u načinu korišćenja prethodne dve komande. Naime, kao što samo ime **revert** komande ukazuje, obavezni argument u vidu heša koji se prosleđuje komandi predstavlja komit koji želimo da „preobratimo“, odnosno „obrišemo“ (za razliku od **reset** gde navodimo komit na koji želimo da se vratimo). Ako bismo napravili paralelu sa prethodnim primerom upotrebe **reset** komande: `git reset 098c396`, u slučaju da želimo da koristimo **revert** izvršili bismo sledeću liniju: `git revert 43bcefe`, odnosno naveli bismo heš komita koji se nalazi nakon komita koji predstavlja argument **reset** naredbe, jer je to komit koji želimo da „preokrenemo“.

Razlika u konceptu **reset** i **revert** komande posebno dolazi do izražaja u slučaju kolaboracije na projektu. U slučaju kada različiti članovi tima imaju iste kopije repozitorijuma na svojim lokalnim računarima, brisanje istorije izmena može dovesti do ozbiljnih problema, naročito imajući u vidu da je svaki član tima u međuvremenu mogao da pravi sopstvene izmene u sadržaju. Iz tog razloga je preporučljivo da se u takvim situacijama koristi **revert** komanda koja čuva istoriju repozitorijuma.

4.3.7. Kolaboracija (saradnja) na Git projektima

Najjednostavniji način kolaboracije na Git projektu predstavlja zajednički rad na jednoj, najčešće „master“, grani. Jednostavnost ovog pristupa se ogleda u odsustvu potrebe za kreiranjem, upravljanjem i spajanjem različitih grana, pa predstavlja dobar izbor za timove koji su novi u korišćenju Git-a. Svaki član tima ima pristup lokalnoj kopiji celog projekta. Međutim, paralelan rad na različitim aspektima projekta je značajno otežan zbog nemogućnosti izolovanja izmena koje se odnose na konkretne projekte celine na kojima rade različiti članovi tima. Na primer, da pretpostavimo da dva člana tima koji rade na zajedničkom projektu: Programer A i Programer B, u nekom trenutku imaju identične kopije repozitorijuma projekta na svom lokalnom računaru. Neka Programer A ima zadatak da implementira novu funkcionalnost dodavanja klijenata u Veb aplikaciju i u tom procesu je izmenio fajl koji sadrži kod za rad sa bazom podataka, pri čemu je napravio nekoliko novih komitova. Sa druge strane, Programer B ispravlja propust koji je pronađen u interfejsu za izlistavanje postojećih korisnika tako što vrši izmene u istom fajlu sa funkcijama za rad sa bazom kao i u slučaju Programera A. Kada Programer B pokuša da sinhronizuje svoj lokalni repozitorijum sa zajedničkim repozitorijumom projekta (na GitHub-u ili GitLab-u npr.) doći će do konflikta. Git će zahtevati od Programera B da najpre spoji postojeće izmene koje je napravio Programer A sa svojim lokalnim repozitorijumom, pa tek onda može da komituje svoj rad i obavi sinhronizaciju sa zajedničkim repozitorijumom. Pošto su oba programera radila na istom fajlu, konflikt koji je nastao se najverovatnije mora razrešiti ručno od strane Programera B na njegovom računaru. Kao glavni nedostatak centralizovane kolaboracije na jednoj grani može se izdvojiti nemogućnost rada na nezavisnim celinama projekta bez uticaja na glavni, integralni kod projekta.

Drugi pristup saradnji na Git projektima predstavlja koncept rada sa granama pojedinačnih funkcionalnosti (eng. *feature branches*). Ako princip kolaboracije na jednoj grani podrazumeva da svi članovi tima vrše izmene u sadržaju iste grane, koncept sa granama funkcionalnosti zahteva da niko od članova koji rade na projektu ne menja glavnu granu. Posao se uvek obavlja samo na sporednim granama funkcionalnosti, pri čemu glavna grana predstavlja zvaničnu (stabilnu) verziju projekta i sadrži istoriju njegovog razvoja. U ovom slučaju, sporedne grane se spajaju u glavnu granu kada se završi određeni stepen razvoja definisane funkcionalnosti, tako da glavna grana nikad se sadrži nepotpun kod, već predstavlja neku zaokruženu celinu. Prednost ovog modela je u tome što je sadržaj koda svih funkcionalnosti na kojima se trenutno radi dostupan u svakom trenutku svim članovima razvojnog tima i stepen razvoja pojedinačne komponente ne utiče niti zavisi od stanja drugih komponenti projekta.

Potencijalni nedostatak ovog pristupa jeste izazov sinhronizacije velikog broja grana sa glavnom granom, naročito kod velikih projekata sa brзом dinamikom razvoja. Mnoge zastarele ili napuštene grane mogu zatrpати projektni repozitorijum ako se njima neprikladno upravlja.

Slično prethodnom modelu, „Gitflow“ pristup koristi pojedinačne grane za razvoj različitih funkcionalnosti, pri čemu umesto jedne sadrži više primarnih ili glavnih grana. Članovi tima kreiraju granu funkcionalnosti za određenu komponentu projekta i odlažu njeno spajanje u primarnu granu sve dok posao na toj komponenti nije završen. Ovaj model je pogodan za projekte koji koriste ciklusne modele razvoja kao što je SCRUM. Gitflow model podrazumeva korišćenje pojedinačnih grana za pripremu, održavanje i beleženje verzija projekta čime se obezbeđuje okruženje za izolovane eksperimente na kodu i efikasniju saradnju, unapređujući ukupni razvojni tok. Na primer, mogu se koristiti dve grane za dokumentovanje istorije projekta umesto samo jedne glavne grane. Zvanična istorija projekta čuva se u glavnoj „master“ grani gde se svakom komitu dodeljuje broj verzije, dok se funkcionalnosti integrišu u drugu primarnu granu „development“. Celokupna istorija projekta biće sadržana u „development“ grani, dok će „master“ grana pružiti samo sažetak istorije i obavljati ulogu zvanične, stabilne verzije projekta.

5. TESTIRANJE SOFTVERSKOG PROJEKTA

Testiranje softvera je nezaobilazna komponenta životnog ciklusa razvoja softvera koja pomaže da se garantuje da program ispunjava svoje zahteve, izvršava se kako je planirano i da je visokog kvaliteta. Testiranje nosi ključnu ulogu u pravovremenom identifikovanju i otklanjanju softverskih grešaka, sprečavajući skupocene ispravke sistema, kako u toku razvoja, tako i nakon isporuke.

Razmera 5:10:200 često se koristi kao smernica u upravljanju softverskim projektima, prikazujući uporedni trošak ispravljanja softverskih grešaka u različitim fazama razvojnog procesa. Ovo pravilo sugerise da će ispravljanje greške koja je identifikovana tokom faze dizajna softverskog projekta biti 5 puta skuplje nego da je otkrivena tokom faze prikupljanja zahteva. Ako se greška otkrije tokom faze implementacije, trošak za njeno ispravljanje biće 10 puta veći, dok bi se trošak za ispravku greške koja je pronađena tokom održavanja povećao na 200 puta.

Analiza zahteva	Projektovanje	Implementacija	Održavanje
	x5	x10	x200

Osnovni princip na kome se zasniva navedena ideja jeste da što je greška kasnije otkrivena u razvojnem procesu, to će biti skuplje njeno ispravljanje zbog povećane kompleksnosti i troškova povezanih sa modifikacijom softvera. Široko se koristi kao pomoćno sredstvo prilikom donošenja odluka o određivanju količine truda i resursa koje treba uložiti u osiguravanje kvaliteta i testiranje, kao i za naglašavanje važnosti rane detekcije i ispravljanja grešaka u razvojnem procesu. Važno je napomenuti da ova smernica nije apsolutno pravilo i da se stvarni odnos troškova može menjati u zavisnosti od više faktora, kao što su veličina i kompleksnost softverskog projekta, stručnost i iskustvo razvojnog tima, kao i alati i metodologije razvoja i testiranja koji se koriste. Ipak, osnovni princip uvećavanja troškova za ispravku grešaka kako softverski projekat napreduje ostaje univerzalno priznat princip u upravljanju softverskim projektima.

Testiranje softvera je takođe ključno za osiguravanje da program ispunjava potrebe i očekivanja svojih korisnika. Razvojni timovi mogu dobiti korisne komentare i informacije koje mogu koristiti za poboljšanje programa i korisničkog iskustva uključivanjem korisnika ili njihovih predstavnika u proces testiranja.

Dodatno, testiranje potvrđuje da softver ispunjava relevantne standarde i zakone, što je ključno u sektorima gde je bezbednost prioritet i gde postoji jasno definisana regulativa.

5.1. VERIFIKACIJA I VALIDACIJA

Dva važna procesa koja obezbeđuju da softver ispunjava postavljene zahteve i funkcioniše na predviđeni način jesu **verifikacija** i **validacija**. Verifikacija je skup aktivnosti koje se izvode na sistemu kako bi se osiguralo da sistem pravilno **implementira** projektne zahteve. To podrazumeva analizu različitih softverskih komponenti, uključujući kod, dizajnersku specifikaciju i dokumentaciju. Provera da dizajn softvera tačno odražava zahteve i poštuje utvrđene principe dobrog interfejsa naziva se verifikacija dizajna. S druge strane, verifikacija koda podrazumeva pregled izvornog koda kako bi se osiguralo da sledi principe dobre prakse kodiranja i na pravilan način implementira ponašanje softvera.

Za razliku od verifikacije, validacija uključuje procenu projekta sa ciljem da se osigura da napravljeni sistem zadovoljava specifikaciju koja je dogovorena sa klijentom. Drugim rečima, validacija proverava da li se pravi odgovarajući sistem, a verifikacija obezbeđuje da se sistem pravi na odgovarajući način. Proces validacije može uključivati testiranje nefunkcionalnih zahteva poput performansi, pouzdanosti, prenosivosti i bezbednosti, pri čemu se rezultati najčešće se izražavaju kvantitativno i egzaktno su proverljivi kroz proces testiranja. Na kraju, postoje situacije kada je neophodno sprovesti proveru usklađenosti za zahtevima okruženja, što uključuje pravnu regulativu ili industrijske standarde. Kroz čestu uključenost korisnika u proces validacije, razvojni tim obezbeđuje da softver ispunjava njihove zahteve i da je jednostavan za upotrebu.

5.2. VRSTE TESTIRANJA

Jedan od načina sprovođenja verifikacije i validacije jeste testiranje. Na osnovu opsega koje obuhvataju, mogu se razlikovati nekoliko tipova testiranja:

- **Jedinično testiranje**, koje za cilj ima da se proveri ponašanje najmanjih nedeljivih celina koda. Jedinično testiranje proverava da li se kod ponaša onako kako je navedeno prilikom projektovanja sistema. Izvršavanje ovih testova je najčešće automatizovano.
- **Integrisano testiranje**, kada je cilj testiranja osigurati da se različite komponente sistema ponašaju na očekivani način kada su međusobno povezane. Akcenat integrisanog testiranja je na greškama u načinu razmene podataka između komponenti. Najčešći problemi koji se javljaju u ovoj fazi testiranja vezani su nesklad između tipova ili formata podataka koji se prosleđuju kao izlaz jedne komponente i strukture koju druga komponenta očekuje kao svoj ulaz.
- **Sistemska testiranje**, gde je cilj testiranja osigurati da sistem implementira sve zahteve i ponaša se u skladu sa njima. Tokom sistemskog testiranja, softver se testira u okruženju koje oponaša produkciono okruženje kako bi se simulirali uslovi stvarne upotrebe. Cilj sistemskog testiranja je da se identifikuju problemi koji mogu nastati kada se softver primeni u realnom okruženju.
- **Testiranje upotrebljivosti**, ima za cilj da proveri da li je korisničko iskustvo intuitivno, efikasno i jednostavno. Testiranje upotrebljivosti je posebno važno u dizajniranju korisničkih interfejsa kako bi se smanjila verovatnoća ljudskih grešaka.

Dva osnovna pristupa upravljanju procesom testiranja softverskog projekta su **razvoj rukovođen testovima** (Test-Driven development) i **razvoj rukovođen ponašanjem** (Behavior-Driven development). Standardni proces Test-Driven razvoja testova započinje kreiranjem testa, zatim implementiranjem funkcionalnosti, i na kraju potencijalnim ispravljanjem testova, formirajući kružni obrazac. Ali šta se dešava kada su promene u toj meri značajne da učine postojeće testove beskorisnim jer je izmenjeno ponašanje softvera različito od početnog stanja? U tim situacijama, prisiljeni smo da izvršimo sveobuhvatnu ponovnu evaluaciju i ponovno pisanje testova. Ova strategija testiranja se fokusira na potvrdu funkcionalnosti koda, stvarajući gotovo direktnu vezu između koda i testova. Kao što će kasnije biti prikazano, TDD pristup podrazumeva postojanje konkretnih metoda (funkcija) sa jasno definisanim argumentima i tipom vrednosti koje vraćaju,

pri čemu su testovi prilagođeni toj konkretnoj implementaciji. Ova stroga zavisnost može predstavljati problem u slučaju agilnog razvoja kada ne postoji jasno predefinisana specifikacija proizvoda koja detaljno opisuje svaku funkciju koja treba da bude implementirana i testirana. Umesto toga, podrazumeva brzo pravljenje prototipa, iterativno usavršavanje na osnovu povratnih informacija i fleksibilno prilagođavanje rešenjima. Ponekad je potrebno napraviti strateški korak unazad, promeniti smer razvoja određene komponente projekta, kako bi se krenulo napred i projekat uspešno završio.

Kako bi se izbegla zavisnost testova od konkretnog koda koji je korišćen koristi se model razvoja testova baziran na ponašanju. U prethodno pomenutom TD razvojnom modelu definisanje testova u potpunosti obavljaju programeri koji su upoznati sa tehnologijom u kojoj je softver razvijan. Za razliku od toga, BD pristup je vođen poslovnim zahtevima, odnosno, naglasak je na karakteristikama proizvoda umesto na unutrašnjem funkcionisanju koda. Ponašanje softvera se najčešće opisuje kroz korisničke **priče**. Priča je nešto što opisuje poslovne zahteve, poput onoga šta se traži, kako bi trebalo da funkcioniše, i kakav bi trebao biti rezultat. Artikuliše se govornim jezikom koji je lako razumljiv poslovnim ljudima i usklađen je sa poslovnim potrebama. Primeri korišćenja se dokumentuju na način koji omogućava automatizovano testiranje koje kasnije pomaže razvojnom timu u procesu pisanja koda. Na ovaj način se obezbeđuje da se resursi za razvoj softvera efikasno raspoređuju koordinirajući prioritete investitora, programera i korisnika.

5.3. PHPUNIT TEST

Pisanje testova i postavljanje automatizovanog testiranja trebalo bi da budu deo svakog projekta i budžeta. Brza izrada projekta bez testiranja kasnije može prouzrokovati više grešaka, skupo restrukturiranje koda i nepredvidive troškove. Investiranje u automatizovano testiranje i pisanje koda koji se jednostavno testira je ključno za uspeh projekta. PHP je programski jezik prvobitno napisan za proceduralno programiranje, međutim, u današnjim verzijama je u prilagođen objektno orijentisanim principima. PHPUnit je testno okruženje za testiranje pojedinačnih programskih jedinica (units) pisanih u PHP programskom jeziku, što znači da se primarno koristi u procesu TD testiranja. Uslov za korišćenje PHPUnit alata je instaliran PHP programski jezik sa dodatnim proširenjima (eng. *extensions*) - php-cli, php-json, php-mbstring, php-xml i php-xdebug. Instalacija navedenih komponenti u Linuks operativnom sistemu se iz terminala može obaviti korišćenjem sledeće komande:

```
$ sudo apt install php php-cli php-json php-mbstring php-xml
php-xdebug
```

Po završetku instalacije neophodnih modula, PHPUnit se može preuzeti sa zvanične veb adrese <https://phar.phpunit.de/phpunit.phar> na sledeći način:

```
$ wget https://phar.phpunit.de/phpunit.phar
```

Navedeni proces će instalirati najnoviju verziju PHPUnit alata, koja u trenutku pisanja knjige zahteva verziju PHP 8.1. Ukoliko je potrebno raditi sa starijom izdanjem PHP-a, moguće je preuzeti raniju stabilnu verziju PHPUnit-a tako što će se u prethodnoj komandi putanja do PHPUnit fajla zameniti sa konkretnom željenom verzijom alata. Tako, npr. `wget https://phar.phpunit.de/phpunit-9.phar` komanda preuzima PHPUnit 9, koji uspešno radi sa PHP okruženjem verzije 7.3 i kasnijim.

Kako bi se alat mogao pokrenuti neophodno je da se preuzetom fajlu dodeli dozvola za izvršavanje:

```
$ chmod +x phpunit.phar
```

Radi lakšeg pristupanja alatu, poželjno je iskopirati ga u neku od lokacija koje se nalaze na listi podrazumevanih putanji u kojima Linux operativni sistem traži izvršne fajlove. To se može postići korišćenjem komande:

```
$ sudo mv phpunit.phar /usr/local/bin/phpunit
```

Za potvrdu uspešne instalacije dovoljno je pokrenuti alat sa argumentom koji prikazuje verziju alata:

```
$ phpunit --version
```

```
PHPUnit 10.1.2 by Sebastian Bergmann and contributors.
```

Alternativan način instalacije PHPUnit-a podrazumeva korišćenje **composer** alata i obavlja se na sledeći način:

```
$ composer require --dev phpunit/phpunit
```

Navedena komanda će instalirati PHPUnit u trenutno aktivni direktorijum. U ovom slučaju alat se pokreće korišćenjem pune putanje do izvršnog fajla, odnosno, iz aktivnog direktorijuma gde je obavljena instalacija, pokreće se komandom `./vendor/bin/phpunit`.

Kako bismo demonstrirali koncept testiranja PHP aplikacije korišćenjem PHPUnit alata kreiraćemo fajl Student.php koji će definisati klasu **Student** na sledeći način:

```
<?php
class Student {
    public $broj_indeksa;
    public $predmeti = [];
    public $ime;
    public function __construct(int $broj_indeksa, string $ime) {
        $this->broj_indeksa = $broj_indeksa;
        $this->ime = $ime;
    }
    public function get_ime(): string {
        return "Ime studenta je " . $this->ime . ".";
    }
    public function get_broj_indeksa(): string {
        return $this->broj_indeksa;
    }
    public function dodaj_predmet(string $predmet): bool {
        $this->predmeti[] = $predmet;
        return true;
    }
    public function ukloni_predmet(string $predmet): bool {
        if (!in_array($predmet, $this->predmeti)) return false;
        unset($this->predmeti[array_search($predmet, $this->predmeti)]);
        return true;
    }
}
```

Klasa Student, osim konstruktora, sadrži četiri metode:

- `get_ime()` koja prikazuje ime studenta
- `get_broj_indeksa()` koja prikazuje broj indeksa studenta,

i dve klase koje upravljaju predmetima koje student pohađa:

- `dodaj_predmet()` koja dodaje predmet u niz predmeta koje student sluša
- `ukloni_predmet()` koja uklanja predmet iz spiska predmeta.

Testiranje funkcionisanja komponenti programa obavlja se korišćenjem ugrađenog mehanizma u okviru PHPUnit alata pod nazivom „potvrde“ (eng. *assertions*). „Potvrda“ predstavlja programski kod koji izvršava drugi deo koda (funkciju ili metodu) u poznatom kontekstu, sa poznatim ulazima, i upoređuje izlaz sa očekivanim vrednostima. Najčešće korišćene potvrde uključuju:

- `assertContains()` - koja proverava da li se element naveden u prvom argumentu nalazi u okviru drugog elementa
- `assertCount()` - koja proverava da li je broj elementa niza jednak vrednosti prvog argumenta
- `assertEmpty()` - koja proverava da li je navedeni element prazan
- `assertEquals()` - koja proverava da li su vrednosti dva elementa jednake
- `assertTrue()` i `assertFalse()` - koje proveravaju da li je vrednost izraza **tačno**, odnosno **netačno**
- `assertInt()` - koja proverava da li je tip navedene promenljive ceo broj
- `assertNull()` - koja proverava da li je vrednost elementa NULL

Metode potvrde vraćaju grešku ukoliko provera koju obavljaju nije uspešna. Test koji bi proveravao da li se metode koje smo definisali u okviru klase **Student** ponašaju na očekivani način smestićemo u fajl `TestStudent.php`:

```
<?php
use PHPUnit\Framework\TestCase;
require('Student.php');
final class TestStudent extends TestCase {
    // Testiranje konstruktora
    public function test_constructor() {
        $student = new Student(2022200200, 'Petar');

        $this->assertSame('Petar', $student->ime);
        $this->assertSame(2022200200, $student->broj_indeksa);
        $this->assertSame(2021100100, $student->broj_indeksa); //
        vraca failure
        $this->assertEmpty($student->predmeti);
    }
    // Testiranje promenjive $ime
    public function test_get_ime() {
        $student = new Student(2022200200, 'Petar');

        $this->assertIsString($student->get_ime());
        $this->assertStringContainsStringIgnoringCase('eta',
        $student->get_ime());
    }
    // Testiranje promenjive $broj_indeksa
    public function test_get_broj_indeksa() {
        $student = new Student(2022200200, 'Petar');

        $this->assertIsNumeric($student->get_broj_indeksa());
    }
}
```

Kao što se može videti iz prethodnog primera, svaki test slučaja je posebna klasa (TestStudent). Fajl koji sadrži kod samog testa bi trebalo da ima identičan naziv kao klasa koju testira uz dodatak prefiksa „Test“. Osim toga, klasa testa mora da proširuje ugrađenu klasu PHPUnit\Framework\TestCase. Ime svake testne metode bi trebalo da počinje rečju „test“. Konvencija je da se nakon toga navede naziv metode iz klase koja se testira kako bi značenje imena testne klase bilo razumljivo iz samog njenog naziva. U prikazanom primeru su definisane tri metode: `test_constructor()`, `test_get_ime()` i `test_get_broj_indeksa()`.

Prva metoda demonstrira osnovni koncept pisanja testova u PHPUnit alatu. Najpre se kreira objekat klase **Student** tako što se konstruktoru klase prosleđuju dva argumenta koja predstavljaju broj indeksa i ime studenta. Nakon toga se pozivaju različite metode potvrda kako bi se proverilo očekivano stanje polja kreiranog objekta. Na primer, `$this->assertSame('Petar', $student->ime)` upoređuje vrednost znakovnog niza (String) „Petar“ i polja **ime** objekta **\$student** koji sadrži podatke o kreiranom studentu. Obzirom da je prilikom kreiranja objekta u konstruktoru kao prvi argument naveden String „Petar“, pomenuta metoda potvrde proverava da li je trenutna vrednost polja **ime** jednaka očekivanom Stringu „Petar“. Slično je i u slučaju provere vrednosti polja **broj_indeksa**, pri čemu se očekuje da njegova vrednost bude „2022200200“. Kao što će se videti iz rezultata testiranja, linija `$this->assertSame(2021100100, $student->broj_indeksa)` vraća grešku upravo iz razloga što se vrednosti koje se upoređuju ne poklapaju. Poslednja naredba u okviru `test_constructor()` metode, korišćenjem `assertEmpty()` potvrde, proverava da li je polje **predmeti** (koje je definisano u klasi **Student** kao niz elemenata) prazno.

Kao što se može videti iz samog naziva, `test_get_ime()` metoda za cilj ima testiranje vrednosti polja **ime** upotrebom metode `get_ime()` koja je definisana u okviru klase **Student**. U ovom slučaju, primenjene su dve provere: da li je vrednost koju vraća metoda `get_ime()` tipa String, korišćenjem potvrde `assertIsString()` i da li vraćena vrednost sadrži u sebi podstring „eta“, putem `assertStringContainsStringIgnoringCase()` potvrde. Pošto je za vrednost argumenta koji u konstruktoru predstavlja ime studenta korišćen String „Petar“, očekivano je da se obe metode koje obavljaju testiranje izvrše bez greške.

Poslednja testna metoda je korišćena za proveru ispravnosti vrednosti koju vraća klasna metoda `get_broj_indeksa()`. Metoda za potvrdu `assertIsNumeric()` vraća grešku ukoliko joj nije prosleđen brojčani tip argumenta.

Sam proces testiranja se pokreće korišćenjem komande **phpunit** <naziv testa>:

```
$ phpunit TestStudent.php
```

```
PHPUnit 9.6.7 by Sebastian Bergmann and contributors.
```

```
F.. 3 / 3 (100%)
```

```
Time: 00:00.002, Memory: 22.00 MB
```

```
There was 1 failure:
```

```
1) TestStudent::test_constructor
```

```
Failed asserting that 2021100100 is identical to 2021100200.
```

```
/var/www/html/phpunit/TestStudent.php:11
```

```
FAILURES!
```

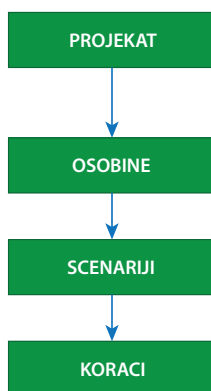
```
Tests: 3, Assertions: 6, Failures: 1.
```

Izlaz komande nam prikazuje rezultat obavljenog testiranja. Najpre, u sažetku rezultata se može videti da su u procesu testiranja korišćena tri testa (tri testne metode) koji ukupno sadrže šest provera (šest metoda potvrde). U jednom slučaju, potvrda je vratila grešku (Failures: 1). Detaljne informacije o lokaciji i uzroku greške su ispisane nakon podataka o vremenu trajanja testa i iskorišćenoj memoriji. U ovom primeru metoda `test_constructor()`, klase **TestStudent**, na jedanaestoj liniji koda je vratila grešku koja ukazuje da je došlo do neslaganja između vrednosti koja joj je prosleđena kao prvi argument i vrednosti polja **broj_indeksa**, koje je navedeno kao drugi argument. Kao očekivano stanje je naveden String „202100100“, dok je vrednost koja je sadržana u polju **broj_indeksa** jednaka „2021100200“, kao što je navedeno prilikom konstruisanja objekta.

5.4. BEHAT

Behat je alat za testiranje ponašanja softvera koje je opisano na jeziku razumljivom ljudima, upotrebom **Gherkin** jezika. Za razliku od PHPUnit-a kojim se testiraju konkretne funkcionalnosti softvera, Behat koristi Gherkin sintaksu da bi kreirao testne scenarije koji opisuju željeno ponašanje sistema u određenoj situaciji. Pošto pripada grupi alata namenjenih slučajevima testiranja putem BD modela, Behat predstavlja moćan alat da se osigura da razvoj softvera bude usklađen sa poslovnim očekivanjima.

Pre nego što se započne korišćenje Behat alata u procesu testiranja softvera, neophodno je upoznati se sa osnovama Gherkin notacije. Standardna struktura koja se upotrebljava prilikom opisivanja željenog ponašanja softvera u Gherkin jeziku prikazana je na sledećem dijagramu.



Na vrhu strukture se nalaze osobine (eng. *features*) projekta koje opisuju različite funkcije koje softver može da obavlja i stanja u kojima može da se nalazi. Na primer, ukoliko se želi implementirati mogućnost logovanja na sistem, to bi predstavljalo jednu osobinu projekta koju je potrebno testirati. Sledeći nivo su scenariji koji predstavljaju slučajeve korišćenja neke konkretne osobine. Tako, jedan od mogućih slučajeva prilikom logovanja predstavlja „uspešno logovanje“, drugi slučaj bi mogao biti „neuspešno logovanje“, pri čemu svaki od njih podrazumeva zasebno ponašanje sistema. Svaki scenario se sastoji od niza koraka koje softver treba da obavi i željenog rezultata koji treba da ostvari u odnosu na scenario koji je izvršen. Ako bismo opet kao primer uzeli osobinu logovanja korisnika na sistem i slučaj (scenario) uspešnog logovanja, mogući koraci koje bismo definisali mogli bi da budu: u situaciji kada se korisnik nalazi

na „login“ stranici, kada unese korisničko ime i lozinku, i kada pritisne dugme „Login“, prikaži mu poruku dobrodošlice. Kako bi se navedeno ponašanje zapisalo u Gherkin notaciji koriste se ključne reči koje jezik definiše (npr. Feature, Scenario, Given, When, Then).

Feature: Admin Login operation

In order to manage users

System admin

Needs to be able to log in to the system

Scenario: Login Successful

Given I am on the login page

When I enter my username and password

And I click the Login button

Then I should see a welcome message

Scenario: Login Failed

Given I am on the login page

When I enter an incorrect username or password

And I click the Login button

Then I should see an error message

Naziv i opis osobina može biti proizvoljan, pri čemu je dobra praksa da se koristi sažeta i jasna terminologija kako bi sve zainteresovane strane mogle da razumeju koje ponašanje softvera se testira. U prikazanom kodu su definisana dva ranije pomenuta scenarija, zatim, radnje koje obavlja korisnik u svakom od njih i očekivano ponašanje softvera u odnosu na akcije korisnika. Kada Behat pokrene određeni scenario, ako se sistem ponaša kako je opisano u scenariju, tada će scenario proći, u suprotnom je neuspešan. Svaki scenario sadrži opis koraka koji prati definisani obrazac: sistem se dovodi u neko stanje (Given I am on the login page), obavljaju se neke operacije sa sistemom (When I enter my username and password And I click the Login button) i na kraju se izvršava željeni ishod operacija (Then I should see a welcome message).

Najjednostavniji način instaliranja Behat alata je korišćenjem Composer menadžera paketa:

```
$ composer require --dev behat/behat
```

Navedena komanda će instalirati Behat u direktorijum **vendor**. Proveru uspešnosti instalacije možemo proveriti korišćenjem **vendor/bin/behat -V** komande. Inicijalizaciju novog projekta obavljam o izvršavanjem komande **vendor/bin/behat --init**.

Kao primer najjednostavnijeg korišćenja Behat-a navešćemo scenario testiranja upotrebe Linuks komandi echo i cat. Prvi korak predstavlja opisivanje osobine koju želimo da proverimo, scenarija koji očekujemo i koraka koji treba da budu sprovedeni u tom procesu. Komanda **echo** se može koristiti za upisivanje sadržaja u fajl, a komanda **cat** za njegovo isčitavanje. Dakle, željeni slučaj korišćenja bi mogao da glasi „Upisivanje i čitanje sadržaja iz fajla“, a koraci koji su sadržani u njemu: ukoliko se nalazim u direktorijumu XYZ, kada pokrenem komandu echo 'Pozdrav, svete!' > test_file.txt, i kada pokrenem komandu cat test_file.txt, očekujem da na ekranu vidim poruku 'Pozdrav, svete!'. U Gherkin jeziku zapis bi izgledao na sledeći način:

```
Feature: Combining echo and cat commands
  In order to be able to write and read to and from a file
  As a Linux user
  I need to be able to use echo and cat commands

Scenario: Writing and reading a String to/from a file
  Given I am in the directory "XYZ"
  When I run the "echo 'Hello, world!' > test_file.txt" command
  And I run the "cat test_file.txt" command
  Then I should see the message "Hello, world!" in the output
```

Behat očekuje da se zapisi osobina koje se testiraju nalaze u okviru **features** direktorijuma koji je automatski kreiran prilikom inicijalizacije projekta. Navedeni primer možemo snimiti u fajl po nazivom write_read.feature. Ako pokrenemo Behat, kao izlaz ćemo dobiti nešto slično sledećem prikazu:

```
$ vendor/bin/behat
Feature: Combining echo and cat commands
  In order to be able to write and read to and from a file
  As a Linux user
  I need to be able to use echo and cat commands

Scenario: Writing and reading a String to/from a file
  Given I am in the directory "XYZ"
  When I run the "echo 'Hello, world!' > test_file.txt" command
  And I run the "cat test_file.txt" command
  Then I should see the message "Hello, world!" in the output

1 scenario (1 undefined)
4 steps (4 undefined)
0m0.01s (9.58Mb)

>> <snippet_undefined><snippet_keyword>default</snippet_keyword>
suite has undefined steps. Please choose the context to generate
snippets:</snippet_undefined>
```

```
[0] None
[1] FeatureContext
```

Ono što se najpre može zaključiti jeste da je Behat prepoznao jedan scenario i četiri definisana koraka. Osim toga, obaveštava nas da **FeatureContext** klasa ima nedostajuće korake i predlaže nam da ih automatski generišemo. FeatureContext je testna klasa kreirana izvršavanjem behat komande sa argumentom `--init` i njen sadržaj se nalazi na putanji `features/bootstrap/FeatureContext.php`. Ukoliko izaberemo opciju "[1] FeatureContext" iz interfejsa Behat alata, na ekranu ćemo dobiti generički sadržaj klase FeatureContext u odnosu na zapis osobine koji smo napravili u `write_read.feature` fajlu:

```
> 1
--- FeatureContext has missing steps. Define them with these snippets:
/**
 * @Given I am in the directory :arg1
 */
public function iAmInTheDirectory($arg1)
{
    throw new PendingException();
}
/**
 * @When I run the :arg1 command
 */
public function iRunTheCommand($arg1)
{
    throw new PendingException();
}
/**
 * @Then I should see the message :arg1 in the output
 */
public function iShouldSeeTheMessageInTheOutput($arg1)
{
    throw new PendingException();
}
```

Prikazani tekst možemo kopirati u `FeatureContext.php` fajl ili možemo zatražiti od Behata da ga automatski generiše pokretanjem komande **vendor/bin/behat --append-snippets**, a zatim odabirom opcije "[1] FeatureContext". Nakon ovog koraka, klasa FeatureContext bi trebalo da sadrži konstruktor i tri metode, po jednu za svaki korak koji smo definisali (koraci 2 i 3 predstavljaju izvršavanje komandi pa se zapravo testiraju kroz istu metodu `iRunTheCommand()`). „Komentari“ iznad metoda (`/** */`) predstavljaju PHP sintaksu koja označava

blokove dokumentacije koju mogu da koriste različiti alati za upravljanje kodom. Konkretno, @When I run the :arg1 command govori Behat-u da izvrši odgovarajuću metodu svaki put kad u "feature" fajlu naide na korak koji glasi: ... I run the ... command. Ono što na kraju preostaje da se uradi jeste da se definišu tela metoda na sledeći način:

```
<?php
use Behat\Behat\Tester\Exception\PendingException;
use Behat\Behat\Context\Context;
use Behat\Gherkin\Node\PyStringNode;
use Behat\Gherkin\Node\TableNode;

/**
 * Defines application features from the specific context.
 */
class FeatureContext implements Context
{
    /**
     * Initializes context.
     *
     * Every scenario gets its own context instance.
     * You can also pass arbitrary arguments to the
     * context constructor through behat.yml.
     */
    public function __construct()
    {
    }

    /**
     * @Given I am in the directory :arg1
     */
    public function iAmInTheDirectory($arg1)
    {
        if (!file_exists($arg1)) {
            mkdir($arg1);
        }
        chdir($arg1);
    }

    /**
     * @When I run the :arg1 command
     */
    public function iRunTheCommand($arg1)
    {
    }
}
```

```

        exec($arg1, $output);
        $this->output = $output;
    }
    /**
     * @Then I should see the message :arg1 in the output
     */
    public function iShouldSeeTheMessageInTheOutput($arg1)
    {
        $actual_output = trim($this->output[0]);
        if ($actual_output !== $arg1) {
            throw new \Exception("Expected message '$arg1', but
found '$actual_output'.");
        }
    }
}

```

U procesu testiranja, metoda `iAmInTheDirectory()` predstavlja prvi korak u scenariju („ukoliko se nalazim u direktorijumu XYZ“) kada se proverava da li direktorijum „XYZ“ postoji, a zatim se postavlja kao aktivni (radni) direktorijum (PHP naredba `mkdir()` kreira direktorijum, a `chdir()` menja aktivni direktorijum). Nakon toga, poziva se metoda `iRunTheCommand()` koja izvršava komandu koja joj je prosleđena kao parametar. U slučaju fajla sa opisom osobine koji smo naveli, u pitanju je najpre komanda `echo 'Hello, world!' > test_file.txt`, a nakon toga `cat test_file.txt`. Nakon njenog izvršavanja putem `exec($arg1, $output)` naredbe, izlaz komande se upisuje kao vrednost polja **output** klase `FeatureContext`. Poslednji korak predstavlja pozivanje metode `iShouldSeeTheMessageInTheOutput()` koja upoređuje očekivanu poruku `$arg` sa stvarnom porukom `$actual_output` koja je dobijena od sistema.

Kao izlaz pokretanja Behat alata dobiće se sledeći prikaz:

```
$ vendor/bin/behat
```

```
Feature: Combining echo and cat commands
```

```
In order to be able to write and read to and from a file
```

```
As a Linux user
```

```
I need to be able to use echo and cat commands
```

```
Scenario: Writing and reading a String to/from a file
```

```
#
```

```
features/write_read.feature:6
```

```
Given I am in the directory "XYZ"
```

```
#
```

```
FeatureContext::iAmInTheDirectory()
```

```
When I run the "echo 'Hello, world!' > test_file.txt" command
```

```
#
```

```
FeatureContext::iRunTheCommand()
```

```
And I run the "cat test_file.txt" command
```

```
#
```

```
FeatureContext::iRunTheCommand()
    Then I should see the message "Hello, world!" in the output  #
FeatureContext::iShouldSeeTheMessageInTheOutput()

    1 scenario (1 passed)
    4 steps (4 passed)
    0m0.01s (9.63Mb)
```

Boldirani delovi izlaza Behat alata pod dvostrukim navodnicima predstavljaju vrednosti parametara koje je Behat u procesu testiranja prosledio odgovarajućim metodama testne klase FeatureContext. Konkretno, metodi `iAmInTheDirectory()` je prosledjena vrednost „XYZ“ kao naziv direktorijuma koji će po potrebi biti kreiran i koji će postati aktivan, a „echo 'Hello, world!' > test_file.txt“ je prva komanda koja će nakon toga biti izvršena. Iz ovog razloga po završetku testiranja možemo očekivati da vidimo postojanje pomenutog direktorijuma i fajla u okviru fajl sistema.

Drugi primer koji ćemo razmotriti kako bismo približili vezu između različitih komponenti koje čine strukturu DB modela testiranja jeste korišćenje `pwd` komande. Ova Linuks komanda prikazuje putanju radnog direktorijuma. Osobina koju testiramo u ovom slučaju bi se mogla zapisati na sledeći način upotrebom Gherkin jezika:

```
Feature: Display the current working directory

Scenario: Checking the current working directory
    Given I am in the directory "/tmp/test_pwd"
    When I run the "pwd" command
    Then I should see the current directory is "/tmp/test_pwd"
```

Kao i u prethodnom „echo-cat“ primeru, u okviru scenarija pod navodnicima se nalaze vrednosti putem kojih se želi proveriti očekivano ponašanje softvera. Odnosno, koraci bi se mogli opisati kao: ako se nalazim u direktorijumu „/tmp/test_pwd“, kada pokrenem komandu „pwd“, trebalo bi (očekujem) da vidim izlaz „/tmp/test_pwd“. Izvršavanjem komande **vendor/bin/behat --append-snippets** i odabirom opcije "[1] FeatureContext", Behat će generisati FeatureContext.php fajl koji u sebi sadrži definicije testnih metoda na osnovu scenarija koji smo odredili. Preostaje da se generička tela metoda prilagode konkretnom slučaju:

```
use Behat\Behat\Context\Context;
use Behat\Gherkin\Node\PyStringNode;
use Behat\Gherkin\Node\TableNode;

class FeatureContext implements Context
{
```

```

private $output;

/**
 * @Given I am in the directory :directory
 */
public function iAmInTheDirectory($directory)
{
    if (!file_exists($directory)) {
        mkdir($directory);
    }
    chdir($directory);
}

/**
 * @When I run the :command command
 */
public function iRunTheCommand($command)
{
    exec($command, $output);
    $this->output = $output;
}

/**
 * @Then I should see the current working directory is
 :expectedDirectory
 */
public function iShouldSeeTheCurrentDirectoryIs($expected
Directory)
{
    $actualDirectory = trim($this->output[0]);
    if ($actualDirectory !== $expectedDirectory) {
        throw new \Exception("Expected directory '$expected
Directory', but found '$actualDirectory'.");
    }
}
}

```

U skladu sa prethodno rečenim, pokretanjem `vendor/bin/behat` komande bi trebalo da se na ekranu prikaže rezultat uspešnog testiranja definisanog ponašanja.

IZRADA SOFTVERSKJE DOKUMENTACIJE



6. IZRADA SOFTVERSKJE DOKUMENTACIJE

Kreiranje softverske dokumentacije predstavlja neizostavan segment životnog ciklusa razvoja softvera. Ona služi kao detaljan zapis o dizajnu i implementacionim odlukama donetim tokom procesa razvoja, dok takođe pruža uputstva za korišćenje i održavanje softvera. Efektivna dokumentacija doprinosi kvalitetu softvera i osigurava da je razumljiv i jednostavan za održavanje od strane drugih. Prvi korak u kreiranju softverske dokumentacije je identifikacija onoga što treba dokumentovati. Ovo je ključno jer određuje smer i obim celokupnog procesa dokumentacije. Identifikacija potreba za dokumentacijom zavisi od prirode softvera i specifičnih potreba korisnika. Takođe, potrebno je razmotriti i ulogu dokumentacije u održavanju softvera, kao i u pružanju podrške korisnicima i razvojnim timovima. Ovaj korak zahteva blisku saradnju sa svim relevantnim učesnicima, uključujući korisnike, programere, testere i tehničku podršku, kako bi se osiguralo da su sve ključne informacije uzete u obzir prilikom donošenja odluke. Uopšteno govoreći, kada je reč o softverskim projektima, mogu se razlikovati sledeći tipovi dokumentacije:

- Dokumentacija arhitekture softvera
- Dokumentacija dizajna softvera
- Dokumentacija implementacije
- Dokumentacija testiranja
- Dokumentacija upravljanja konfiguracijom softvera
- Korisnička dokumentacija

Dokumentacija arhitekture softvera pruža sveobuhvatan pregled strukture sistema, uključujući glavne komponente i njihove međusobne odnose. Cilj ovog dokumenta jeste da obezbedi da svi članovi tima i zainteresovane strane razumeju kako je softverski sistem koncipiran i kako njegovi delovi međusobno komuniciraju. Za ovu svrhu se u najvećoj meri koriste dijagramski prikazi arhitekture i grafičke predstave komponenti i interakcija, a pored toga sadrže i obrazloženja odluka koje su dovele do izbora određene arhitekture. **Dokumentacija dizajna softvera** detaljno opisuje kako će različite komponente sistema biti realizovane na nivou dizajna. Dokument najčešće sadrži dijagrame klasa i aktivnosti, dijagrame toka, korisničke priče ili scenarije, modele objekata ili podataka ili druge opise strukture komponenti i načina njihovog funkcionisanja. Članovi razvojnog tima koriste dokumentaciju dizajna kako bi razumeli način na koji različiti delovi sistema treba da funkcionišu zajedno i kao referencu koja im pomaže u doslednoj implementaciji funkcionalnosti u skladu sa predviđenim dizajnom.

Dokumentacija implementacije podrazumeva tehničke detalje o kodiranim funkcionalnostima koji se najčešće prezentuju u vidu komentara u samom kodu sa opisima funkcija i algoritama. Ova dokumentacija je ključna za programere koji rade na održavanju i proširivanju jer pruža jasan uvid u postojeće načine korišćenja pojedinačnih jedinica koda, čime se izmena i dodavanje novih funkcionalnosti značajno pojednostavljuje. **Dokumentacija testiranja** uključuje sve planove testiranja, slučajeve testiranja, rezultate testiranja, i izveštaje o pronađenim nedostacima. Njen cilj jeste da osigura robusnost softvera kroz jasno definisanu metodologiju, čime se garantuje kvalitet sistema pre nego što se pusti u produkciju. Sadrži informacije o strategijama testiranja, alatima koji se koriste, kao i procedurama za identifikovanje i rešavanje problema. **Dokumentacija upravljanja konfiguracijom** osigurava praćenje svih promena u konfiguraciji softvera tokom njegovog životnog ciklusa. Obuhvata informacije o verzijama softvera, konfiguracionim stavkama i mehanizmima kojima se upravlja konfiguracijom. Osim toga, sastavni deo dokumenta može predstavljati plan upravljanja konfiguracijom koji određuje procedure provere i dokumentovanja izmena u konfiguraciji, kao i proces odobravanja izmena (npr. od strane rukovodioca tima). **Korisnička dokumentacija** pruža uputstva i smernice za krajnje korisnike softvera. Uključuje korisničke priručnike, često postavljana pitanja (eng. *FAQ*), ili druga uputstva. Ova dokumentacija je namenjena da korisnicima pomogne da efikasno koriste softver, rešavaju probleme i maksimalno iskoriste njegove funkcionalnosti.

Sledeći korak nakon što su potrebe za dokumentacijom identifikovane je da se napravi plan za dokumentovanje softvera. Ovaj plan treba da sadrži obim i format dokumentacije, kao i alate i resurse koji će biti korišćeni za njeno kreiranje. Važno je definisati koji delovi softvera zahtevaju detaljno objašnjenje, koje funkcionalnosti treba posebno naglasiti, i kako će dokumentacija biti strukturirana. Na primer, uputstva namenjena korisnicima treba da budu pisana jednostavnim jezikom i da uključuju vizuelne elemente kao što su slike i dijagrami, dok tehnička specifikacija koja je namenjena tehničkim licima treba da pruži detaljan uvid u arhitekturu i dizajn softvera. U najopštijem smislu, dokumentacija može biti namenjena za tri grupe čitalaca: korisnike, administratore i inženjere. Korisnici su osobe koje ostvaruju neku korist od korišćenja softvera i kojima je softver primarno namenjen. Administratori su zaduženi za instalaciju, konfigurisanje i redovno održavanje softvera. Posao inženjera jeste ažuriranje i unapređenje sistema.

Kako bi se obezbedio ujednačen stil dokumenata i standardizovala njegova struktura, za kreiranje dokumentacije se može koristiti neki od popularnih opisnih (eng. *markup*) jezika. Osnovna ideja iza ovakvog pristupa jeste upotreba sintakse odabranog opisnog jezika za automatizovano generisanje stilizovanih elemenata u finalnom dokumentu. Jedan od najpoznatijih jezika ovog tipa jeste XML (Extensible Markup Language). Iako XML predstavlja jedan od mogućih izbora za ručno pisanje dokumentacije, njegova primena u ovom slučaju nije preporučljiva. Razlog tome leži u složenoj sintaksi i veličini teksta koji je potrebno koristiti kako bi se generisao čak i najjednostavniji dokument. Da pretpostavimo da želimo generisati opis upotrebe funkcije `calculateSumInt` na sledeći način:

Function: `calculateSumInt`

Description: This function calculates the sum of two integers.

Parameters:

- **a** (int): The first integer.
- **b** (int): The second integer.

Returns:

- **int:** The sum of the two integers.

Example:

```
int result = calculateSumInt(10, 20); // result is 30
```

Odgovarajući XML kod bi mogao da izgleda ovako:

```
<?xml version="1.0" encoding="UTF-8"?>
<documentation>
  <function>
    <name>calculateSumInt</name>
    <description>
      This function calculates the sum of two integers.
    </description>
    <parameters>
      <parameter>
        <name>a</name>
        <type>int</type>
        <description>The first integer.</description>
      </parameter>
      <parameter>
        <name>b</name>
        <type>int</type>
```

```

        <description>The second integer.</description>
    </parameter>
</parameters>
<return>
    <type>int</type>
    <description>The sum of the two integers.</description>
</return>
<example>
    <code>
        int result = calculateSumInt(10, 20);
        // result is 30
    </code>
</example>
</function>
</documentation>

```

Odnosno, za prikazivanje formatiranog teksta koji sadrži 274 karaktera, neophodno je parsirati 935 karaktera XML koda, skoro tri i po puta više. Sa druge strane, razvijeni su opisni jezici kompaktnije sintakse, prilagođene potrebama pisanja složene i obimne dokumentacije. Jedan od reprezentativnih primera predstavlja **Markdown** opisni jezik, kao najpopularniji jezik za predstavljanje strukture i definisanje stila softverske dokumentacije. Njegova osnovna prednost je jasna i intuitivna sintaksa, a najveći nedostatak donekle ograničene mogućnosti osnovne varijante jezika i nepostojanje stroge standardizacije. To je bio povod za pojavljivanje velikog broja verzija Markdown jezika, od kojih je najrasprostranjenija GitHub Flavored varijanta. Kao što se može zaključiti iz samog imena, GitHub Flavored se koristi za pravljenje dokumentacije na GitHub platformi i podržava korišćenje dodatnih elemenata u odnosu na osnovnu verziju, kao što su tabele, prikazivanje izvornog koda, automatsko linkovanje teksta koji sadrži URL adrese, korišćenje emotikona itd. Ako bismo želeli da generišemo dokument koji sadrži uputstvo za upotrebu funkcije `calculateSumInt` koje je prikazano u prethodnom primeru, mogli bismo koristiti naredni GitHub Flavored Markdown kod:

```

# Function: calculateSumInt
**Description:** This function calculates the sum of two integers.
## Parameters:
- **a** (int): The first integer.
- **b** (int): The second integer.
## Returns:
- **int**: The sum of the two integers.
## Example:
```java
int result = calculateSumInt(5, 10); // result is 15

```

Važno je zapaziti da, za razliku od XML, ovaj kod sadrži svega 30-ak karaktera više od teksta koji se dobija njegovim parsiranjem. Osnovna Markdown sintaksa podržava formatiranje teksta slično HTML prikazu, različite veličine zaglavlja (H1-H6), uređene i neuređene liste, ubacivanje slika i linkova. Iako ograničen broj i stil elemenata može predstavljati ograničavajući faktor prilikom izrade dokumentacije, korišćenje osnovne verzije jezika obezbeđuje najširu kompatibilnost, bez potrebe za izmenom koda u slučaju prebacivanja na neku od proširenih verzija.

Ostali opisni jezici koji se upotrebljavaju za pisanje dokumentacije koje vredi pomenuti jesu reStructuredText, AsciiDoc i LaTeX. Sintaksa prva dva jezika je u velikoj meri slična Markdown sintaksi, sa dodatkom elemenata koji nisu sadržani u osnovnoj Markdown verziji. LaTeX se prvenstveno koristi za pisanje tehničkih dokumenata i naučnih radova. Njegove osnovne prednosti su široka podrška za rad sa matematičkim formulama, jednostavno upravljanje referencama i mehanizmi za jasno definisanje strukture dokumenta. Glavni nedostatak je kompleksnost sintakse što se može videti na narednom kodu (koji generiše ranije pomenuti primer upotrebe funkcije `calculateSumInt`):

```
\documentclass{article}
\usepackage{hyperref}
\usepackage{listings}
\begin{document}
\section*{Function: calculateSumInt}
\textbf{Description:} This function calculates the sum of two integers.
\subsection*{Parameters:}
\begin{itemize}
\item \textbf{a} (int): The first integer.
\item \textbf{b} (int): The second integer.
\end{itemize}
\subsection*{Returns:}
\begin{itemize}
\item \textbf{int}: The sum of the two integers.
\end{itemize}
\subsection*{Example:}
\begin{lstlisting}[language=Java]
 int result = calculateSumInt(5, 10); // result is 15
\end{lstlisting}
\end{document}
```

Konkretni zahtevi i potrebe korisnika igraju ključnu ulogu u procesu izbora jezika za generisanje dokumentacije. XML pruža visoko strukturiran format, ali može biti opširan i složen kada su u pitanju jednostavni tekstovi. Markdown sa svojom jednostavnošću i širokom podrškom predstavlja odličan izbor za jednostavnu dokumentaciju i u slučajevima kratkih rokova, jer mu prvenstveno nedostaju napredne funkcije drugih jezika. reStructuredText nudi jednostavnost i čitljivost, što ga čini idealnim za Python dokumentaciju i druge tehničke tekstove. Sintaksa AsciiDoc jezika predstavlja ravnotežu između čitljivosti i funkcionalnosti, što jezik čini pogodnim za širok spektar tipova dokumentacije, od tehničkih priručnika do članaka i knjiga. Na kraju, LaTeX se izdvaja po svojoj izraženoj sposobnosti da upravlja složenim formatiranjem, pa svoju primenu nalazi posebno u akademskim i naučnim dokumentima, iako zahteva najviše vremena za savladavanje u odnosu na ostale pomenute opisne jezike.

## 6.1. PHPDOCUMENTOR

Kao primer alata za automatsko generisanje dokumentacije koja se odnosi na softverski projekat pomenućemo **phpDocumentor**. Razlog ovog izbora leži u jednostavnosti njegovog korišćenja i reprezentativnom modelu primene koji važi u slučaju većine drugih alata slične namene koji nisu nužno vezani za PHP programski jezik. Najveći broj alata za generisanje dokumentacije obavlja automatsku analizu izvornog koda i kreira dokumentaciju na osnovu komentara i anotacije na koje naiđe. U zavisnosti od konkretnog alata, dokumentacija sadrži hijerarhiju klasa, detaljan opis metoda (funkcija), polja i povezanih fajlova, kao i druge informacije od interesa za korisnike. Osim toga, složeniji alati omogućavaju konvertovanje dokumenata u različite formate kao što su PDF ili HTML. U najvećem broju slučajeva, korisniku je na raspolaganju opcija odabira postojeće ili kreiranja prilagođene teme koja će se primeniti prilikom kreiranja finalnog izgleda dokumenata. Sve navedeno važi i u slučaju phpDocumentor alata.

Instalacija phpDocumentor-a je prilično jednostavna. Pod pretpostavkom da se na računaru nalazi PHP verzija 7.4 ili veća, potrebno je skinuti *phar* fajl sa zvanične URL adrese<sup>11</sup>, dodeliti mu dozvole izvršavanja i iskopirati ga u direktorijum sa globalnim pristupom:

<sup>11</sup> <https://phpdoc.org/phpDocumentor.phar>

```
wget https://phpdoc.org/phpDocumentor.phar
chmod +x phpDocumentor.phar
mv phpDocumentor.phar /usr/local/bin/phpdoc
```

Nakon toga, alat se pokreće sa bilo koje lokacije pozivanjem komande **phpdoc**. Osnovna upotreba phpDocumentor-a se ostvaruje navođenjem putanje na kojoj se nalazi izvorni kod aplikacije korišćenjem **-d** opcije u komandnoj liniji. U ovom slučaju, dokumentacija će biti generisana u podrazumevanom direktorijumu **.phpdoc/build**, što se može promeniti dodavanjem **-t** argumenta nakon koga sledi naziv željenog direktorijuma:

```
$ phpdoc run -d ./app -t ./docs
```

Za demonstraciju mogućnosti alata korišćemo primer jednostavne klase **Pravougaonik** koja sadrži polja i metode za izračunavanje površine i obima:

```
<?php
/**
 * Class Rectangle
 *
 * This class represents a rectangle and includes methods to
calculate
 * its area and perimeter.
 *
 * @category Geometry
 * @package GeometryPackage
 * @author Milos Antonijevic <mantonijevic@singidunum.ac.rs>
 * @license https://opensource.org/licenses/MIT MIT License
 * @link https://usp.singidunum.ac.rs
 */
class Rectangle {
 /**
 * The length of the rectangle.
 *
 * @var float
 */
 private $length;

 /**
 * The width of the rectangle.
 *
 * @var float
 */
 private $width;
```

```

/**
 * Constructor for the Rectangle class.
 *
 * @param float $length The length of the rectangle.
 * @param float $width The width of the rectangle.
 */
public function __construct($length, $width) {
 $this->length = $length;
 $this->width = $width;
}

/**
 * Gets the length of the rectangle.
 *
 * @return float The length of the rectangle.
 */
public function getLength() {
 return $this->length;
}

/**
 * Sets the length of the rectangle.
 *
 * @param float $length The new length of the rectangle.
 * @return void
 */
public function setLength($length) {
 $this->length = $length;
}

/**
 * Gets the width of the rectangle.
 *
 * @return float The width of the rectangle.
 */
public function getWidth() {
 return $this->width;
}

/**
 * Sets the width of the rectangle.
 *
 * @param float $width The new width of the rectangle.
 */

```

```

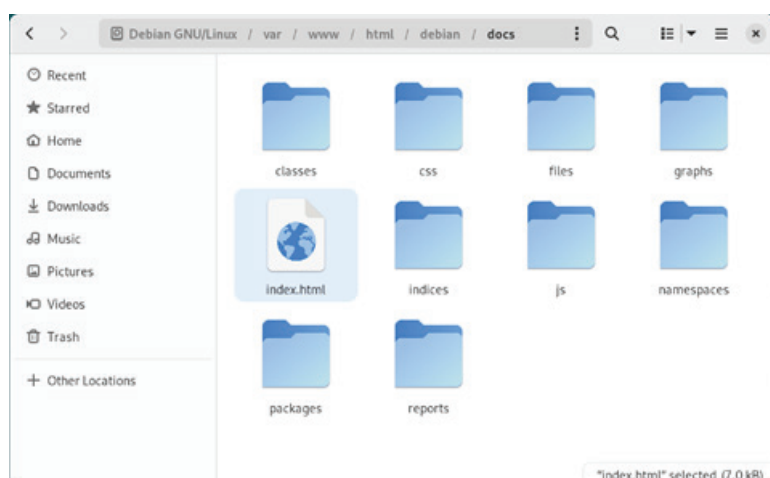
 * @return void
 */
 public function setWidth($width) {
 $this->width = $width;
 }

 /**
 * Calculates the area of the rectangle.
 *
 * @return float The area of the rectangle.
 */
 public function calculateArea() {
 return $this->length * $this->width;
 }

 /**
 * Calculates the perimeter of the rectangle.
 *
 * @return float The perimeter of the rectangle.
 */
 public function calculatePerimeter() {
 return 2 * ($this->length + $this->width);
 }
}

```

Ako navedeni kod snimimo u fajl **Rectangle.php** i izvršimo pomenutu phpdoc komandu, phpDocumentor će kreirati „docs“ direktorijum i u njemu napraviti dokumentaciju klase u formi veb sajta.



Pregled dokumentacije se pokreće otvaranjem index.html fajla. U nastavku je prikazan isečak HTML stranice koji se odnosi na opis metoda klase Pravougaonik. Na desnoj strani dokumenta se može videti spisak polja i metoda koji je linkovan sa odgovarajućim detaljnim prikazom konkretnih elemenata. Navigacija kroz dokumentaciju se obavlja odabirom željene stavke iz liste. U slučaju metoda, generisani opis sadrži njenu namenu, deklaraciju, listu, opis i tip parametara, kao i vrednost koju metoda vraća. Stilom dokumentacije se može upravljati zahvaljujući podršci korišćenju prilagođenih tema koju phpDocumentor pruža.

## Methods

### `__construct()`

[Rectangle.php](#) : 37

*Constructor for the Rectangle class.*

```
public __construct(float $length, float $width) : mixed
```

#### Parameters

`$length` : float

The lenght of the rectangle.

`$width` : float

The width of the rectangle.

### `calculateArea()`

[Rectangle.php](#) : 92

*Calculates the area of the rectangle.*

```
public calculateArea() : float
```

#### Return values

float `__`

The area of the ectangle.

### `calculatePerimeter()`

[Rectangle.php](#) : 102

*Calculates the perimeter of the rectangle.*

## On this page

### Table of contents

[Properties](#)[Methods](#)

### Properties

[\\$length](#)[\\$width](#)

### Methods

[\\_\\_construct\(\)](#)[calculateArea\(\)](#)[calculatePerimeter\(\)](#)[getLenght\(\)](#)[getWidth\(\)](#)[setLenght\(\)](#)[setWidth\(\)](#)

Osnovni element sintakse sa kojim *phpdoc* operiše jeste **DocBlock**. DocBlock predstavlja deo dokumentacije u izvornom kodu koji se odnosi na funkciju određene klase, metode ili drugog strukturnog elementa. DocBlock-ovi su uvek oivičeni konkretnim tipom komentara, nazvanim **DocComment**. DocComment počinje sa `/**` i završava sa `*/`. Svaka linija između početka i kraja DocComment-a treba da počinje zvezdicom `*`<sup>12</sup>. Analiza „Rectangle.php“ koda pokazuje nekoliko osnovnih kategorija rezervisanih reči koje *phpdoc* koristi kako bi odredio funkciju koju navedeni tekst nosi. Tako, na samom početku fajla možemo videti blok koji se odnosi na opšte informacije o fajlu, kao što su podaci o autoru i licenci (`@author`, `@license`). Blokovi koji se odnose na samu klasu su podeljeni u tri kategorije: **sažetak**, **opis** i **tagove** (i anotacije). Sažetak pruža kratak prikaz funkcije pridruženog elementa. Opis sadrži detaljne informacije o elementu (polju ili metodi) i može uključivati opis algoritma, primer upotrebe i namenu elementa. Završava se kada se naiđe na prvi tag ili na kraj samog blok. Tagovi i anotacije započinju znakom `@` i koriste se da prikažu podatke o pridruženim stavkama određenog elementa, najčešće parametrima i povratnim vrednostima (`@param`, `@return`). Na ovaj način, korišćenjem DocBlock i DocComments elemenata, *phpDocumentor* obezbeđuje jasne i detaljne opise klasa, metoda i drugih strukturnih komponenti softvera. Integracijom *phpDocumentor*-a u razvojni tok projekta, timovi mogu pojednostaviti proces pisanja dokumentacije i isporučiti pouzdanije i razumljivije softverske proizvode.

12 <https://docs.phpdoc.org/3.0/guide/getting-started/what-is-a-docblock.html>



# ISPORUKA I ODRŽAVANJE





## 7. ISPORUKA I ODRŽAVANJE

Isporuka i održavanje su faze u životnom veku proizvoda koje započinju nakon završetka projekta. Zatvaranje projekta je poslednja etapa razvoja projekta, kada se rezultati projekta predaju zainteresovanim stranama i kada se obavlja evaluacija uspešnosti projekta zarad budućeg unapređenja razvojnog procesa. Postoji nekoliko načina na koja projekat može da se završi:

- Završetak integracijom proizvoda u poslovanje organizacije. Ovo predstavlja uspešan slučaj završetka projekta.
- Završetak „izgladnjivanjem“. U ovom slučaju projekat se završava usled nedostatka resursa za nastavak njegovog razvoja.
- Završetak gašenjem. Ovo se dešava u situaciji kada se projekt prekida jer ne ispunjava svoje ciljeve ili više nije profitabilan.

Zatvaranje projekta se najčešće obavlja kroz dobijanje formalnog prihvatanja proizvoda od strane klijenta koje prati instalacija proizvoda na produkciono okruženje i isporuka prateće dokumentacije. Sa strane projektnog tima, kao deo ovog procesa se obavlja arhiviranje koda za upotrebu u narednim projektima (u meri u kojoj ugovori sa klijentom to dopuštaju) i oslobađanje resursa. Projekat je uspešno završen kada se klijent i projektni menadžer slože da projekat zadovoljava unapred definisane kriterijume (funkcionalnosti, vreme razvoja i sl.). Kada su u pitanju softverski projekti, formalno prihvatanje uvek uključuje testiranje finalne verzije sistema koja se instalira u poslovno okruženje klijenta. Proces predaje i instalacije softvera može biti različite kompleksnosti. U najjednostavnijem slučaju, softver se dostavlja klijentu na nekom prenosnom medijumu – usb flash uređaju, DVD-ju ili putem instalacionog fajla dostupnog za preuzimanje preko Veb sajta. U složenijim slučajevima, kada novi sistem treba da zameni stari, pri čemu je neophodno obezbediti kontinuitet pružanja usluge ili proizvodnje potrebno je obezbediti postojanje odvojenih okruženja u kojima se proizvod nezavisno implementira. Konačni cilj je osigurati da bilo koja promena na sistemu ne ometa poslovanje klijenta. U ovu svrhu, najčešće se govori o kopijama ili instancama softverskog sistema u okviru tri okruženja:

- **Razvojnog okruženja** – na kome se obavlja razvoj softverskog koda. Pošto je potpuno izolovan od poslovnog okruženja klijenta, izmene i dopune koje se obavljaju na softveru ne mogu da utiču na aktivnosti koje klijent obavlja. Podaci koji se koriste u razvoju bi trebalo da predstavljaju anonimizovane ili pseudomizovane kopije realnih podataka, kako bi se zaštitila privatnost stvarnih osoba koje koriste usluge softvera. Razlika u

dva navedena pristupa je u stepenu privatnosti. U slučaju anonimizacije podaci koji mogu da identifikuju osobe ili kritične informacije (npr. brojeve kreditnih kartica) su u potpunosti uklonjeni ili zamenjeni besmislenim, nasumično generisanim podacima. Kada se govori o pseudomizaciji, podaci koji se koriste u razvojnom okruženju se generišu kodiranjem realnih produkcionih podataka, na način da se ključ za njihovo dekodiranje nalazi na bezbednoj lokaciji zaštićen jakim šifrarskim mehanizmima.

- **Testnog okruženja** – na kome se testira sistem koji je spreman za isporuku kod klijenta. Testno okruženje je izolovano od razvojnog i produkcionog okruženja, tako da, s jedne strane, nijedna promena koju naprave programeri ne ometa testiranje, a s druge strane, testne aktivnosti ne utiču na rad klijenta. Slično razvojnom okruženju, u testiranju se podrazumeva korišćenje kodiranih kopija podataka iz stvarne baze. Testno okruženje bi trebalo da ima identičnu hardversku i softversku konfiguraciju kao okruženje klijenta na koje će se sistem isporučiti, uključujući operativni sistem, bazu podataka i druge softverske komponente (drajvere, fajervol, mrežna podešavanja i sl).
- **Produkciono okruženje** – na kome se sistem realno koristi u sklopu poslovnog okruženja klijenta. Promene u kodu se ne obavljaju na produkcionom okruženju jer direktno utiču na aktivnosti klijenta, već se na njega isporučuje proizvod tek nakon što uspešno prođe kroz fazu testiranja. Imperativ je da ovo okruženje poseduje visoku stabilnost i pouzdanost, jer eventualni prekidi u radu sistema mogu voditi do finansijskih i reputacionih gubitaka. Sistem mora da bude implementiran tako da može efikasno da podnese realno opterećenje vezano za broj korisnika ili veličinu podataka koje obrađuje i da poseduje optimalnu brzinu odgovora na zahteve koji mu se šalju i tako obezbedi konzistentne performanse. Osim toga, od naročite važnosti je primeniti odgovarajuće mere zaštite od nepriviligovanog pristupa i drugih bezbednosnih pretnji. Na kraju, produkciono okruženje mora da poseduje mehanizme pravljenja rezervnih kopija u slučaju nepredviđenih okolnosti.

Uvođenje sistema u produkciono okruženje klijenta se može obaviti korišćenjem više strategija. Osnovni pristup jeste **kompletna zamena** starog sistema novim. Ovaj metod je jednostavan jer eliminiše potrebu za održavanjem dva sistema istovremeno. Međutim, nivo rizika je visok jer ne postoji rezervna opcija u slučaju da novi sistem ne funkcioniše ispravno. Prekid rada starog sistema i prelazak na novi može izazvati značajne probleme ako novi sistem nije dovoljno testiran ili ako korisnici nisu adekvatno obučeni.

Druga strategija jeste tzv. **paralelni pristup**, kada su stari i novi sistem istovremeno u upotrebi neko vreme. Na ovaj način je omogućeno testiranje i evaluacija novog sistema pre gašenja starog i njegove kompletne zamene. Prednost ovog pristupa je smanjenje rizika, jer korisnici mogu da nastavu rad na starom sistemu ako se u novom sistemu pojave nedostaci. Sa druge strane, ovaj pristup je skup i zahteva dodatne resurse za održavanje oba sistema u funkcionalnom stanju. **Pilotiranje** je strategija u kojoj se novi sistem prvo implementira za ograničen broj korisnika ili određenu poslovnu jedinicu. Ovaj pristup pruža mogućnost testiranja novog sistema u kontrolisanom okruženju. Pilotiranje omogućava identifikaciju i ispravku problema pre nego što se sistem proširi na celu organizaciju. Treba imati u vidu da je u ovom slučaju potrebno obezbediti odgovarajuće interfejske koji će osigurati da podaci obrađeni od strane poslovne jedinice koja koristi novi sistem mogu biti korišćeni od strane ostatka organizacije koja koristi stari sistem. Na kraju, sistem je moguće isporučiti postupno u različitim **fazama**. Iako je ovaj pristup najsporiji, on nudi najviše kontrole i najmanji rizik, jer svaka nova funkcija može biti temeljno testirana, a korisnici imaju dovoljno vremena da se na nju prilagode pre nego što se ceo sistem uvede u produkciju. Poslednji korak u isporuci proizvoda predstavlja održavanje obuka za korišćenje softvera kako bi se osiguralo da korisnici poseduju potrebne veštine za upotrebu novog sistema.

Nakon završene isporuke započinje faza **održavanja**, koja za cilj ima da obezbedi da proizvod ostane funkcionalan nakon njegovog puštanja u rad, pa do povlačenja iz upotrebe. Međunarodni standard ISO/IEC 14764:2022<sup>13</sup>, pod nazivom „Softverski inženjering – Životni ciklus softvera – Održavanje“, detaljno opisuje proces održavanja softvera i objašnjava četiri kategorije koje se na njega odnose:

- **Korektivno održavanje** – koje uključuje uklanjanje nedostataka softvera otkrivenih nakon isporuke krajnjem korisniku. Problemi se najčešće identifikuju kroz prijave od strane korisnika (putem portala ili korisničke podrške) ili automatizovanih sistema za praćenje performansi. Nakon potvrđivanja postojanja propusta potrebno je klasifikovati ga u odnosu na ozbiljnost i uticaj koji ima na sistem, kako bi se osiguralo da najkritičniji propusti budu rešavani prioritetno. Proces ispravke nedostatka od strane inženjera započinje otkrivanjem osnovnog uzroka koji je doveo do pojavljivanja problema. Kada se izvrše potrebne modifikacije koda, a pre puštanja zakrpe u produkciono okruženje neophodno je izvršiti njeno temeljno testiranje. Na kraju, obavljaju se potrebne izmene u dokumentaciji i obaveštavaju se zainteresovane strane.

13 <https://www.iso.org/standard/80710.html>

- **Adaptivno (prilagodljivo) održavanje** – koje se odnosi na izmene softverskog sistema promenama u spoljašnjem okruženju. Ovaj tip održavanja ima za cilj da obezbedi da softver ostane funkcionalan prilikom promena u tehnološkoj ili organizacionoj sferi. Promene najčešće uključuju ažuriranje operativnog sistema ili drugih softverskih komponenti, hardverske promene i promene u mrežnoj infrastrukturi.
- **Perfektivno (unapređujuće) održavanje** – koje predstavlja proces koji uključuje nadogradnju sistema kako bi se unapredile njegove performanse, funkcionalnost i upotrebljivost. Pобољшanje performansi se može postići optimizacijom koda ili prilagođavanjem postojećih algoritama kako bi se smanjila potrošnja resursa. Perfektivno održavanje se za identifikovanje oblasti za poboljšanja uglavnom oslanja na povratne informacije od strane korisnika sistema.
- **Preventivno održavanje** – koje za cilj ima da spreči pojavljivanje budućih problema rešavanjem potencijalnih nedostataka pre nego što prerastu u probleme. Najvažnija aktivnost obuhvata kontinuirano praćenje rada sistema i održavanje softvera ažurnim kroz implementiranje bezbednosnih zakrpi i ažuriranje eksternih biblioteka i alata. Redovno pravljenje rezervnih kopija sistema i izrada plana oporavka u slučaju nepredviđenih okolnosti je kritičan zadatak koji osigurava zaštitu podataka i brzi oporavak u slučaju havarije ili incidenta. Kako bi se sprečilo iskorišćavanje novih ranjivosti koji se gotovo svakodnevno pojavljuju, neophodno je sprovesti stalne bezbednosne provere, kako interne, tako i eksterne.

Sastavni deo procesa održavanja predstavlja pružanje tehničke podrške. Podrška se odnosi na pružanje pomoći korisnicima, klijentima i zainteresovanim stranama u slučaju kada naiđu na tehničke probleme. Aktivnosti koje su uključene u proces podrške obuhvataju dijagnostifikovanje problema, implementiranje rešenja i pružanje ekspertize u korišćenju softvera. Sa druge strane, služba tehničke podrške pruža povratne informacije o proizvodu razvojnom timu. Podrška korisnicima može biti obezbeđena putem različitih kanala, najčešće preko šaltera za informacije, onlajn foruma, elektronske pošte i četa ili *tiketing sistema*. Tiketing sistem predstavlja alat za upravljanje i praćenje propusta, incidenata ili drugih zahteva koji dolaze od zainteresovanih strana (korisnika, zaposlenih, klijenata itd.) prema razvojnom timu. Tiket (kartica) obično sadrži tip zahteva (potencijalni nedostatak softvera, incident ili predlog za poboljšanjem), prirodu problema i prioritet. Dobra praksa podrazumeva da svaki tiket ima dodeljenu osobu odgovornu za upravljanje zahtevom od njegovog kreiranja do zatvaranja.

## 7.1. KONTINUIRANA INTEGRACIJA I KONTINUIRANA ISPORUKA (CI/CD)

U mnogim softverskim projektima, distribucija proizvoda se obavlja ručno, od strane različitih, nezavisnih individua ili timova. Tako, produkciono okruženje na kome će se softver izvršavati postavlja i konfiguriše jedan tim, instaliranje potrebnih biblioteka i alata drugi, a sam softverski sistem isporučuje i konfiguriše treći tim. U ovom procesu, programerski tim koji je razvijao softver kreira potrebne instalacione i konfiguracione fajlove, obavlja migraciju baze podataka i isporučuje dokumentaciju osobama koje su zadužene za isporuku softvera na okruženje klijenta. Zbog umešanosti velikog broja ljudi, prostor za greške prilikom ručne isporuke proizvoda je veliki. Iz tog razloga poželjno je koristiti automatizovane modele isporuke koji su brzi, pouzdani, predvidivi i niskog rizika. Osim isporuke, zadatak je automatizovati i procese testiranja i upravljanja konfiguracijom, kako bi se sva tri procesa integrisala u jedinstveni model koji bi omogućio uvođenje softverskog sistema nezavisno od okruženja (razvojnog, testnog ili produkcionog) pritiskom na dugme.

Prva komponenta u procesu unapređenja modela isporuke jeste upravljanje konfiguracijom. Postoji nekoliko tehnika za skladištenje i pribavljanje informacija o konfiguraciji softvera, zavisno od faze u kojoj se sistem nalazi. Najčešći oblik u kome se čuvaju konfiguracioni podaci jesu parovi „naziv“ : „vrednost“ ili standardizovani formati kao što su YAML i XML. Ove informacije se obično čuvaju u bazi podataka, registrima (u slučaju Windows operativnog sistema) ili u tekstualnim fajlovima. Pristup konfiguraciji od strane softvera se može realizovati putem kreiranja skripti koje se izvršavaju u vreme *kompajliranja* izvornog koda, koje će konfiguracione parametre ugraditi u samu binarnu (izvršnu) formu aplikacije. Drugi način jeste izrada skripti koje se pokreću prilikom *isporuke* (eng. *deployment scripts*) ili instaliranja sistema, koje pribavljaju potrebne podatke sa neke eksterne lokacije ili kroz interakciju sa korisnikom. Ovo je jedini način u slučajevima kada je neophodno da softver dobije podatke karakteristične za konkretno okruženje, npr. kredencijali za pristup bazi podataka. Na kraju, sam softver može prikupljati konfiguracione informacije prilikom prvog pokretanja sistema ili prilikom svakog pokretanja putem argumenata koji se prosleđuju komandi koja pokreće softver.

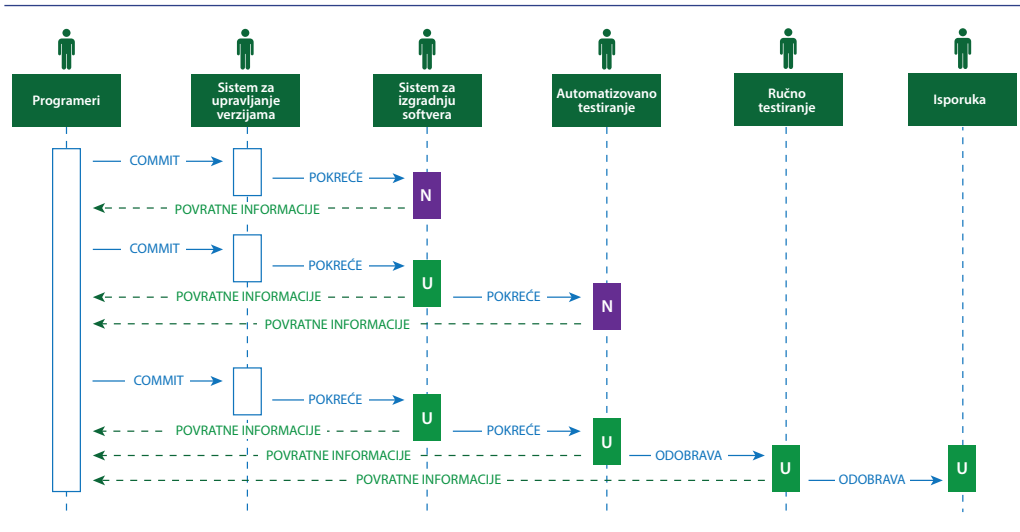
Sastavni deo automatizacije procesa konfiguracije softvera predstavlja održavanje centralizovanog repozitorijuma kroz koji softver dobija konfiguracione podatke koji su mu potrebni. Ovo se može postići uspostavljanjem REST veb servisa kome aplikacija prosleđuje zahtev sa parametrima koji opisuju verziju softvera i okruženje na kome se sistem nalazi i na osnovu toga vraća odgovarajuće informacije.

Sistem upravljanja konfiguracijom bi trebalo u sebi da sadrži mogućnost slanja različitih podataka u odnosu na operativni sistem (u slučaju višepatformskih aplikacija), na mrežnu infrastrukturu i ostale zavisne interne i eksterne komponente (biblioteke, zakrpe ...). Po završetku konfigurisanja operativnog sistema i samog softvera, potrebno je obezbediti da promene u podešenom produkcijskom okruženju ne mogu da se obave izvan definisanog procesa koji je pod kontrolom tima zaduženog za isporuku softvera. Razlog tome leži u činjenici da i najmanja promena može da dovede do prestanka funkcionisanja sistema, pa zato mora da bude testirana pre nego što se isporuči klijentu.

**Kontinuirana integracija** podrazumeva da aplikacija bude u radnom stanju u svakom trenutku razvoja, odnosno, svaki put kad se izvrši neka izmena u softveru, ceo sistem se kompajlira i izvrši se skup detaljnih automatizovanih testova. Ukoliko dođe do problema u procesu kompajliranja, razvojni tim prestaje sa kodiranjem i posvećuje se rešavanju novonastalog problema. U tradicionalnom pristupu nije poznato da li softver radi sve dok neko to ne dokaže u okviru faze testiranja ili integracije. Kada se govori o kontinuiranoj integraciji, podrazumeva se da softver stalno funkcioniše, jer je to dokazano opširnim testiranjem nakon uvođenja svake izmene. Ovakva praksa omogućava timovima bržu isporuku, sa manje propusta, jer se oni otkrivaju u ranoj fazi kada je njihovo otklanjanje jeftinije. Implementacija kontinuirane integracije u razvoj softverskog projekta podrazumeva korišćenje sistema za kontrolu verzija (npr. Git), skripti za automatizovano *build*-ovanje (izgradnju) softvera i testiranje.

U okviru sistema za kontrolu verzija neophodno je pratiti sve aspekte razvoja: kod, testove, bazu podataka, skripte za izgradnju i instalaciju softvera, osim konfiguracionih parametara koji sadrže poverljive podatke, kao što su kredencijali ili ključevi za enkripciju. Automatizovane skripte za izgradnju bi trebalo da se pokreću iz komande linije, bez interakcije sa korisnikom. Osim skripti za izgradnju, neophodno je razviti temeljne mehanizme za automatsko obavljanje testiranja, u suprotnom proces izgradnje pokazuje samo da softver može da se kompajlira i uspešno sastavi. U idealnom slučaju, testovi bi trebalo da se pišu pre nego što programeri započnu sa razvojem određene funkcionalnosti i treba da uključe i nefunkcionalne zahteve kako bi omogućili da se potvrdi da softver obavlja posao za koji je napravljen i da ga obavlja na valjan način.

Kontinuirana integracija se odnosi na fazu razvoja softvera. Međutim, pomenute principe je moguće (i poželjno) primeniti i na proces isporuke i održavanja.



Na prikazanom dijagramu se mogu videti akcije koje se dešavaju prilikom obavljanja izmena u softveru. Kada programerski tim potvrdi izmenu na sistemu za upravljanje verzijama, automatski se pokreće kompajliranje i sastavljanje softvera. U slučaju da postupak izgradnje ne uspe („N“), alat vraća programerima informacije o greškama koje su se javile. Ukoliko je softver uspešno napravljen („U“), pokreće se proces automatizovanog testiranja. U zavisnosti od toga da li je testiranje uspešno ili ne, odobrava se faza ručnog testiranja od strane korisnika. Na kraju, kada korisnici potvrde ispravnost funkcionalnosti i karakteristika proizvoda, odobrava se njegova isporuka na okruženje klijenta. Automatizacija faza, sa jedne strane, omogućava brzu isporuku proizvoda klijentu, a sa druge, mogućnost vraćanja na prethodne verzije softvera u slučaju potrebe. Svaka faza ima svoju ulogu u toku procesa isporuke. Tako, izgradnja softvera potvrđuje da je ispravan na tehničkom nivou, automatizovani testovi pokazuju ispunjenost funkcionalnih i nefunkcionalnih zahteva, a ručno testiranje od strane korisnika za cilj ima da proverí da li je sistem upotrebljiv i da li donosi vrednost klijentu. Proces isporuke modelovan na prikazani način se naziva **kontinuirana isporuka**.

Zadaci uvođenja koncepta kontinuirane isporuke jesu uključivanje svih zainteresovanih strana u izradu plana isporuke, smanjenje uticaja ljudskog faktora kroz automatizaciju procesa, a time i verovatnoće pojavljivanja grešaka, usavršavanje procesa kroz uvežbavanje na okruženju identičnom produkcionom, razvijanje mogućnosti vraćanja na stariju verziju u slučaju da se jave problemi prilikom isporuke i postojanje strategije selidbe produkcionih podataka i konfiguracije. Finalni cilj ovakvog pristupa jeste smanjenje rizika koje proces isporuke softvera sa sobom nosi.

Kontinuirana isporuka podstiče kulturu stalnog unapređenja i agilnosti, omogućavajući timovima da brzo reaguju na promene na tržištu i povratne informacije korisnika. Ova praksa nije samo tehnička implementacija nekog rešenja, već fundamentalna promena u pristupu razvoju softvera, koja akcenat stavlja na saradnji, automatizaciji i kvalitetu. Primena pristupa kontinuirane isporuke značajno unapređuje sposobnost organizacije da inovira i zadrži konkurentsku prednost u rastućem digitalnom okruženju.



# Literatura



- Boehm, B. W. (1988). *A spiral model of software development and enhancement*. Computer, 21, 5, 61-72, doi: 10.1109/2.59.
- Brousseau, J. (2010). *Software Quality Attributes: Following All the Steps*. Addison-Wesley
- Bruxam, P., Diefenbach, H., & Hess, T. (2013). *The Software Industry—Economic Principles, Strategies, Perspectives*. Springer-Verlag, Berlin, Heidelberg, Germany
- Cohn, M. (2005). *Agile Estimating and Planning*. Pearson Education.
- Crispin, L., & Gregory, J. (2008). *Agile Testing: A Practical Guide for Testers and Agile Teams, 1st Edition*. Addison Wesley.
- Dennis, A., Wixom, B. H., & Roth, R. M. (2012). *Systems analysis and design, Fifth Edition*. Hoboken, NJ: John Wiley & Sons.
- Davidson, N. (2012). *Don't Just Roll the Dice*. Efendi Minibooks, United Kingdom.
- Etter, A. (2016). *Modern technical Writing: An Introduction to Software Documentation*. CreateSpace Independent Publishing Platform.
- Fowler, M., & Scott, K. (1999). *UML Distilled Second Edition A Brief Guide to the Standard Object*. Addison Wesley.
- Goldratt, E. M. (1999). *Theory of Constraints*. North River Press.
- Humble, J., & Farley, D. (2010). *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Crawfordsville, Indiana: Pearson Education.
- J. R., M., & S. J., M. (2002). *Project Management: A Managerial Approach, 4th ed*. New York: Wiley.
- Kaner, C., Falk, J., & Nguyen, H. Q. (1999). *Testing Computer Software, 2nd Edition*. Wiley.
- Martin, R. (2002). *Agile Software Development, Principles, Patterns, and Practices, First Edition*. Pearson Education.
- Myers, G. J., Sandler, C., & Badgett, T. (2011). *The Art of Software Testing, 3rd Edition*. Wiley.
- P. Brooks, Jr., F. (2002). *The mythical man-month: Essays on software engineering*. Addison Wesley.
- Pohl, K., & Rupp, C. (2015). *Requirements Engineering Fundamentals: A Study Guide for the Certified Professional for Requirements Engineering Exam, foundation level, IREB compliant, 2nd edition*. Santa Barbara, CA: Rocky Nook.
- Poppendieck, M., & Poppendieck, T. (2003). *Lean Software Development: An Agile Toolkit*. Addison Wesley.

- Portny, S. E. (2010). *Project Management For Dummies®*, 3rd Edition. Indianapolis, Indiana: Wiley Publishing.
- Project Management Institute. (2017). A guide to the project management body of knowledge (PMBOK guide). Newtown Square, Pennsylvania: Project Management Institute.
- Ries, E. (2011). *Lean Startup: Kako današnji preduzetnici koriste neprestanu inovativnost za stvaranje u potpunosti uspešnih poslova*, 2. izdanje. Beograd: iLearn d.o.o.
- Ries, E. (2017). *Startup Način: Kako moderne kompanije koriste preduzetnički menadžement da transformišu kulturu i upravljaju dugoročnim rastom*. Beograd: iLearn d.o.o.
- Robertson, S., & Robertson, J. (2014). *Masterin the Requirements Process: Getting Requirements Right*. Westford, Massachusetts: Pearson Education.
- Rose, S., Wynne, M., & Hellesøy, A. (2015). *The Cucumber for Java Book: Behaviour-Driven Development for Testers and Developers*. Dallas, Texas: The Pragmatic Programmers.
- Royce, W. (2000). *Software Project Management: A Unified Framework*. Addison Wesley.
- Sutherland, J., & Sutherland, J. J. (2014). *Scrum: The Art of Doing Twice the Work in Half the Time*. Crown Currency.
- Thimbleby, H. (1988). Delaying Commitment. *IEEE Software*, 5, 78-86. doi:10.1109/52.2027
- Villaflorita, A. (2014). *Introduction to Software Project Management*. CRC Press, Taylor & Francis Group.
- Wieggers, K., & Beatty, J. (2013). *Software Requirements, Third Edition*. Microsoft Press.
- Wysocki, R. K. (2011). *Effective Project Management: Traditional, Agile, Extreme*, 6th ed. Indianapolis: Wiley.



CIP - Каталогизација у публикацији  
Народна библиотека Србије, Београд

004.42:004.9(075.8)

АНТОНИЈЕВИЋ, Милош, 1982-

Upravljanje softverskim projektom / Miloš Antonijević. - 1. izd. - Beograd :  
Univerzitet Singidunum, 2024 (Beograd : Birograf). - V, 165 str. : ilustr. ; 24 cm

Tiraž 500. - Napomene i bibliografske reference uz tekst. - Bibliografija: str.  
164-165.

ISBN 978-86-7912-824-9

а) Апликативни софтвер -- Пројектовање

COBISS.SR-ID 150934793

© 2024

Sva prava zadržana. Nijedan deo ove publikacije ne može biti reprodukovан u bilo kom vidu i putem bilo kog medija, u delovima ili celini bez prethodne pismene saglasnosti izdavača.



Miloš Antonijević

# UPRAVLJANJE SOFTVERSKIM PROJEKTOM

"Upravljanje softverskim projektima" pruža sveobuhvatan pregled ključnih koncepata, metodologija i alata potrebnih za uspešno upravljanje softverskim projektima. Knjiga počinje uvodom u osnovne koncepte razvoja projekta, uključujući definiciju i karakteristike projekata, upravljanje zainteresovanim stranama i ključne principe poput Gvođenog trougla. Zatim se detaljno obrađuju kritični aspekti analize i specifikacije zahteva, sa posebnim naglaskom na važnost prikupljanja, analize i validacije zahteva kako bi se obezbedio uspeh projekta.

Knjiga pokriva različite modele razvoja, od tradicionalnih pristupa upravljanju projektima do agilnih metodologija poput Scrum-a i Kanban-a. Čitaoci će steći uvid u izgradnju i upravljanje efikasnim projektnim timovima, uz praktične smernice za korišćenje softverskih alata kao što su Gantogrami, mrežni dijagrami i Git za kontrolu verzija.

Sa posebnim delovima posvećenim testiranju softvera, dokumentaciji i isporuci, ova knjiga pruža holistički pogled na životni ciklus razvoja softvera. Iako primarno namenjena studentima, ova knjiga pruža potrebna znanja i veštine za uspešnu navigaciju kroz ključne faze upravljanja modernim softverskim projektima koje mogu biti od koristi i iskusnim menadžerima.