UNIVERZA V LJUBLJANI
EKONOMSKA FAKULTETA

MAGISTRSKO DELO

ZAGOTAVLJANJE KAKOVOSTI IN IZBOLJŠAVE PROCESA RAZVOJA IN VZDRŽEVANJA PROGRAMSKE OPREME V MANJŠEM RAZVOJNEM PODJETJU

Ljubljana, september 2021 JERNEJ MUHA

IZJAVA O AVTORSTVU

Podpisani Jernej Muha, študent Ekonomske fakultete Univerze v Ljubljani, avtor predloženega dela z naslovom Zagotavljanje kakovosti in izboljšave procesa razvoja in vzdrževanja programske opreme v manjšem razvojnem podjetju, pripravljenega v sodelovanju s svetovalcem red. prof. dr. Alešem Popovičom

I Z J A V L J A M

1. da sem predloženo delo pripravil samostojno;

2. da je tiskana oblika predloženega dela istovetna njegovi elektronski obliki;

3. da je besedilo predloženega dela jezikovno korektno in tehnično pripravljeno v skladu z Navodili za izdelavo zaključnih nalog Ekonomske fakultete Univerze v Ljubljani, kar pomeni, da sem poskrbel, da so dela in mnenja drugih avtorjev oziroma avtoric, ki jih uporabljam oziroma navajam v besedilu, citirana oziroma povzeta v skladu z Navodili za izdelavo zaključnih nalog Ekonomske fakultete Univerze v Ljubljani;

4. da se zavedam, da je plagiatorstvo – predstavljanje tujih del (v pisni ali grafični obliki) kot mojih lastnih – kaznivo po Kazenskem zakoniku Republike Slovenije;

5. da se zavedam posledic, ki bi jih na osnovi predloženega dela dokazano plagiatorstvo lahko predstavljalo za moj status na Ekonomski fakulteti Univerze v Ljubljani v skladu z relevantnim pravilnikom;

6. da sem pridobil vsa potrebna dovoljenja za uporabo podatkov in avtorskih del v predloženem delu in jih v njem jasno označil;

7. da sem pri pripravi predloženega dela ravnal v skladu z etičnimi načeli in, kjer je to potrebno, za raziskavo pridobil soglasje etične komisije;

8. da soglašam, da se elektronska oblika predloženega dela uporabi za preverjanje podobnosti vsebine z drugimi deli s programsko opremo za preverjanje podobnosti vsebine, ki je povezana s študijskim informacijskim sistemom članice;

9. da na Univerzo v Ljubljani neodplačno, neizključno, prostorsko in časovno neomejeno prenašam pravico shranitve predloženega dela v elektronski obliki, pravico reproduciranja ter pravico dajanja predloženega dela na voljo javnosti na svetovnem spletu preko Repozitorija Univerze v Ljubljani;

10. da hkrati z objavo predloženega dela dovoljujem objavo svojih osebnih podatkov, ki so navedeni v njem in v tej izjavi.

V Ljubljani, dne _____ Podpis študenta: _____

KAZALO

KAZALO TABEL

KAZALO SLIK

SEZNAM KRATIC

angl. – angleško
BPMN – (angl. Business Process Modeling Notation); Grafična notacija za modeliranje poslovnih procesov in delovnih tokov
DBA – (angl. Database Administrator); Administrator podatkovnih zbirk
ER – (angl. Entity-Relationship); Relacijski model
ERP – (angl. Enterprise Resource Planning); Celovita programska rešitev
IT – (angl. Information Technology); Informacijska tehnologija
SaaP – (angl. Software as a Product); Programska oprema kot produkt
SaaS – (angl. Software as a Service); Programska oprema kot storitev
SDLC – (angl. Software Development Life Cycle); Proces razvoja programske opreme
XP – (angl. Extreme Programming); Ekstremno programiranje

UVOD

Programska oprema je ključnega pomena za delovanje družbe, vlad, podjetij ter različnih institucij in organizacij. Brez nje si sodobnega sveta ne moremo predstavljati. Zagotavlja delovanje mednarodne infrastrukture, omogoča proizvodnjo in distribucijo ter podpira finančni sistem. Zaradi pospešenega napredka tehnologije in pocenitve strojne opreme jo vse bolj srečujemo tudi v vsakdanjem življenju. Programska oprema je bila eden izmed temeljev za razvoj sodobnega sveta in bo ostala ključno sredstvo, na katerega se bo človeštvo zanašalo za reševanje največjih izzivov in vprašanj, ki nas čakajo v prihodnosti (Sommerville, 2016).

Gartner 2020 ocenjuje, da je bila globalna poraba denarja za informacijsko tehnološke (v nadaljevanju IT) namene v letu 2019 3,7 trilijone ameriških dolarjev. Od tega naj bi se v segmentu profesionalne programske opreme, namenjene za delovanje podjetij (angl. enterprise software), porabilo 458 milijard ameriških dolarjev (Gartner, 2020). Profesionalna programska oprema (angl. professional software) je tista vrsta programske opreme, katere razvoj je poklicna dejavnost in je razvita za poslovne namene. Ključna razlika od preostalih vrst programske opreme je, da je namenjena uporabi tistemu, ki ni njen avtor. Ali drugače povedano, profesionalna programska oprema je tista programska oprema, ki se prodaja v obliki produkta ali storitve in katere namen je večanje poslovne vrednosti njenega avtorja in tudi kupca. Takšna programska oprema zahteva visok nivo kakovosti, hiter in zanesljiv proces razvoja ter dolgoročno in obvladljivo vzdrževanje (Sommerville, 2016).

Kako razvojno podjetje pristopi k procesu razvoja in vzdrževanja ter zagotavljanju kakovosti programske opreme, je odvisno od velikosti in organiziranosti podjetja, znanja vključenih posameznikov ter končnega namena programske opreme. Namen magistrskega dela je prispevati k razumevanju specifik in možnosti izboljšav procesa razvoja in vzdrževanja ter zagotavljanja kakovosti programske opreme v manjšem razvojnem podjetju. Za dosego namena v magistrskem delu želim odgovoriti na naslednji dve raziskovalni vprašanji:

- Kako lahko manjše razvojno podjetje izboljša proces razvoja in vzdrževanja programske opreme?
- Kako lahko manjše razvojno podjetje izboljša zagotavljanje kakovosti programske opreme?

Kot odgovor na raziskovalni vprašanji sta osnovna cilja magistrskega dela analiza literature na področju razvoja, vzdrževanja in upravljanja kakovosti programske opreme ter izvedba študije primera razvoja, vzdrževanja in zagotavljanja kakovosti programske opreme na primeru izbranega podjetja. Cilji magistrskega dela v okviru študije primera so:

- Opredelitev, modeliranje in analiza procesa razvoja programske opreme.
- Priprava predlogov za izboljšave procesa razvoja programske opreme.
- Opredelitev, modeliranje in analiza procesa vzdrževanja programske opreme.
- Priprava predlogov za izboljšave procesa vzdrževanja programske opreme.

- Analiza sedanjega stanja zagotavljanja kakovosti programske opreme.
- Priprava predlogov za izboljšave zagotavljanja kakovosti programske opreme.

Ob prispevanju k razumevanju specifik in možnosti izboljšav razvoja, vzdrževanja ter zagotavljanja kakovosti programske opreme je praktičen namen magistrskega dela tudi možnost dejanske uporabe predstavljenih predlogov za večanje poslovne vrednosti izbranega podjetja. Prav tako lahko predstavljeni predlogi služijo kot ideje in smernice za večanje poslovne vrednosti kateregakoli drugega razvojnega podjetja, ki deluje v podobnem kontekstu.

1 RAZVOJ PROGRAMSKE OPREME

K razvoju programske opreme je potrebno pristopiti odgovorno in skrbno, tako da bo končen rezultat obvladljiv in čim dlje uporaben (van Vliet, 2007). Za zagotovitev optimalnega rezultata ne obstaja splošno sprejet pristop, saj je način razvoja programske opreme odvisen predvsem od njenega končnega namena. Razvoj poslovno informacijskega sistema se lahko precej razlikuje od razvoja programske opreme, nameščene v medicinskih aparatih. Uporabljane metode in tehnike pri razvoju programske opreme so odvisne od vrste programske opreme, ki jo razvijamo (Sommerville, 2016).

Obstaja več možnih razvrstitev profesionalne programske opreme. Najpogosteje uporabljena razvrstitev je glede na njenega naročnika (Sommerville, 2016):

- *Generični produkti*. To so aplikacije ali informacijski sistemi, ki so na voljo za nakup katerekoli stranki, ki jo lahko kupi, na primer urejevalniki besedila ali mobilne aplikacije. Pod to vrsto se uvrščajo tudi tako imenovane navpične aplikacije. To so aplikacije, ki so razvite samo za določeno vrsto organizacij, kot je na primer sistem za podporo računovodskemu poslovanju ali knjižnični informacijski sistem.
- *Programska oprema po naročilu*. To so aplikacije ali informacijski sistemi, ki se jih razvije za določenega kupca. Izvajalec načrtuje in implementira programsko opremo, ki mora zadoščati potrebam njenega naročnika. Najpogostejši primer takšne vrste programske opreme so poslovni informacijski sistemi, ki nudijo podporo specifičnim poslovnim procesom.

Bistvena razlika med opredeljenima vrstama programske opreme je, da pri generičnih produktih organizacija, ki razvija programsko opremo, tudi nadzira specifikacijo programske opreme. Kar pomeni, da če se pri razvoju pojavijo težave, lahko ponovno razmislijo, kako in kdaj razviti določeno funkcionalnost. Pri programski opremi po naročilu pa specifikacijo programske opreme nadzira naročnik. Naloga izvajalca je, da znotraj predpisanih časovnih okvirjev zagotovi vse opredeljene funkcionalnosti.

Vendar pa je potrebno poudariti, da postaja ta razlika tudi vse bolj nejasna, saj se vedno več informacijskih sistemov v osnovi razvija kot generični produkt, kasneje pa se razširi oziroma prilagodi na način, ki ustreza potrebam stranke (Sommerville, 2016). Primer takšnega pristopa so sodobne celovite programske rešitve (angl. Enterprise Resource Planning, v nadaljevanju ERP), kjer se zapleten informacijski sistem prilagodi stranki na podlagi njenih poslovnih pravil in lastnosti poslovnih procesov.

Druga najpogostejša razvrstitev profesionalne programske opreme je glede na uporabljen poslovni model pri prodaji. Uveljavljena sta naslednja dva poslovna modela:

- *Programska oprema kot produkt* (angl. Software as a Product, v nadaljevanju SaaP) je tradicionalen poslovni model, pri katerem stranka kupi produkt z enkratnim plačilom ali naroči razvoj novega, ki bo zadoščal njenim potrebam. V primeru programske opreme po naročilu po razvoju tipično sledi obdobje vzdrževanja, v katerem izvajalec nudi podporo stranki ter neprestano spreminja in dograjuje programsko opremo.
- *Programska oprema kot storitev* (angl. Software as a Service, v nadaljevanju SaaS) je novejši poslovni model, pri katerem stranka plačuje periodično naročnino za uporabo programske opreme. Višina naročnine je tipično odvisna od obsega uporabe. SaaS-model je prišel v širšo uporabo z razcvetom računalništva v oblaku.

Ni pa vrsta programske opreme edini dejavnik, ki vpliva na strukturo procesa razvoja. V praksi je način izvedbe procesa odvisen tudi od zahtev stranke, od omejene količine virov, s katerimi razvojno podjetje razpolaga, ter znanja in izkušenj oseb, ki programsko opremo dejansko razvijejo.

Proces razvoja programske opreme (angl. Software Development Life Cycle, v nadaljevanju SDLC) je zelo širok pojem. V literaturi obstaja mnogo različnih definicij. V nadaljevanju povzemam tri primere opredelitve pojma:

- SDLC je zaporedje faz, skozi katere prehaja informacijski sistem od svoje zasnove do trenutka, ko ga odstranimo iz operativnega delovanja in ga nadomestimo z novejšim ali ustreznejšim informacijskim sistemom, ki je potreben za nadaljnje izpolnjevanje poslovnih potreb (Soriano, 2012).
- SDLC zajema sklop aktivnosti (komunikacija, načrtovanje, modeliranje, izgradnja in uveljavitev), ki jih lahko opišemo z različnimi procesnimi toki. Je način zaporedne in kronološke organizacije naštetih aktivnosti (Pressman, 2010).
- SDLC je niz medsebojno povezanih aktivnosti, ki vodijo v izgradnjo programske opreme (Sommerville, 2016).

Ker ne obstaja splošno sprejet seznam metod in tehnik, ki bi bil primeren za razvoj katerekoli programske opreme, tudi ne obstaja en univerzalen način izvedbe procesa razvoja programske opreme, ki bi vedno zagotovil optimalen rezultat. Ne glede na to, da obstaja mnogo različnih načinov izvedbe procesa, lahko za obravnavano vrsto programske opreme

opredelimo štiri temelje aktivnosti, ki morajo biti v določeni meri zajete pri izvedbi procesa. Te aktivnosti so (Sommerville, 2016):

- *Specifikacija programske opreme*. Opredeljene morajo biti funkcionalnosti programske opreme in njihove omejitve pri izvajanju.
- *Izgradnja programske opreme*. Na podlagi specifikacije je potrebno izdelati programsko opremo, ki zadovoljuje opredeljene zahteve.
- *Potrjevanje programske opreme*. Izdelana programska oprema mora biti preverjena in potrjena, da zagotavlja potrebam kupca.
- *Spreminjanje programske opreme*. Programska oprema se mora konstantno dopolnjevati in prilagajati, da izpopolnjuje spreminjajoče se potrebe kupca.

Proces razvoja programske opreme je zahteven, intelektualen in kreativen proces, zato je uspešnost njegove izvedbe odvisna od znanja udeležencev v procesu in njihovih odločitev med izvajanjem procesa. Ker ne obstaja en univerzalen način izvedbe procesa, je večina razvojnih podjetij razvila svojo obliko izvedbe, ki optimizira izkoristek zmogljivosti svojih razvijalcev in se prilagaja vrsti programske opreme, ki jo razvija (Sommerville, 2016). Za podjetja, ki razvijajo varnostno kritičen informacijski sistem, je primerna močno strukturirana in natančna izvedba procesa. Velikokrat v takšnih podjetij tudi vodijo obsežno dokumentacijo o izvajanju posameznih aktivnosti. Za podjetja, ki razvijajo poslovno informacijski sistem, katerega specifikacije se pogosto spreminjajo, pa je bolj primeren prilagodljiv in agilnejši način izvedbe procesa.

Torej, glavna razlika med pristopoma je v načrtovanju posameznih aktivnosti procesa. V prvem (tradicionalnem) pristopu se aktivnosti načrtujejo vnaprej, njihov napredek pa se meri v skladu z zastavljenim načrtom (angl. plan-driven approach). V drugem (agilnem) pristopu pa je načrtovanje aktivnosti postopno in konstantno skozi celoten proces. Tako lahko podjetje pridobi odziv svojih strank že med razvojem in se lažje odzove na morebitne spremembe njihovih potreb.

1.1 Tradicionalen pristop

Za lažje nadzorovanje in spremljanje napredka pri razvoju programske opreme se je najprej pojavil način, pri katerem se določijo smiselni mejniki med začetkom in koncem projekta. V splošnem lahko mejnike v razvoju programske opreme opredelimo kot trenutke v procesu, kadar postane dosegljiv določen dokument (van Vliet, 2007):

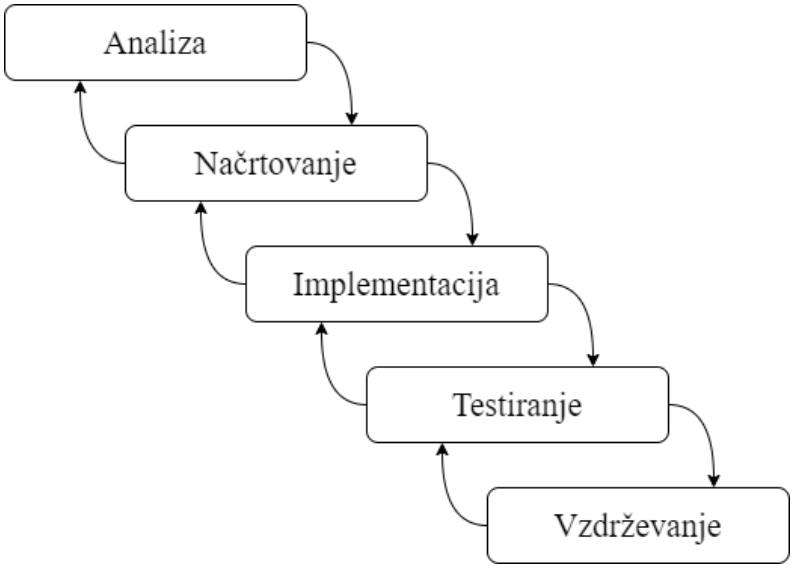
- Dokument, ki vsebuje vsebinsko specifikacijo programske opreme, postane dosegljiv po analizi zahtevanih funkcionalnosti.
- Dokument, ki vsebuje tehniško specifikacijo programske opreme, postane dosegljiv po načrtovanju in arhitekturni zasnovi.
- Po implementaciji programske opreme nastane skupek izvorne kode.

- Po testiranju programske opreme pa postane dosegljiv dokument, ki vsebuje poročilo o uspešnosti testiranja in morebitnih težavah.

Tradicionalen pristop k razvoju programske opreme torej temelji na dokumentaciji (angl. document-driven approach) (van Vliet, 2007). To pomeni, da je gonilo in vodilo projekta skupek dokumentov, ki nastajajo ob zaključku posameznih aktivnosti.

Najbolj znana izvedba tradicionalnega pristopa k razvoju programske opreme je model procesa, ki ima analogijo na slap (angl. waterfall process). Slika 1 prikazuje diagram takšnega pristopa in aktivnosti, ki nastopajo v procesu.

Slika 1: Diagram izvedbe tradicionalnega pristopa k razvoju programske opreme



Prirejeno po van Vliet (2007).

V določeni literaturi je ta model predstavljen tudi brez povratne zanke na predhodne aktivnosti. Kar pomeni, da izvajanje aktivnosti sledi strogo eno za drugim. Takšni primeri v praksi sicer redko obstajajo, saj je potrebno rezultat posamezne aktivnosti skoraj vedno primerjati z njenimi zahtevami. Le tako se lahko ocenita pravilnost in kvaliteta rezultata (Pressman, 2010). V praksi se ti prehodi običajno zgodijo ob potrditvi prej omenjenih dokumentov, ki se podpišejo ob zaključku posamezne aktivnosti (Sommerville, 2016).

Koraki oziroma faze v diagramu neposredno odražajo pet temeljnih aktivnosti v procesu razvoja programske opreme:

- *Analiza*. Funkcionalnosti, zahteve in omejitve programske opreme se opredelijo s sodelovanjem s končnimi uporabniki. Vsi načrtovani cilji so podrobno zapisani in služijo kot specifikacija programske opreme.
- *Načrtovanje*. Arhitekturna zasnova programske opreme, ki jasno definira vse nivoje abstrakcije v programski kodi in njihova razmerja.

- *Implementacija*. V tej fazi se načrt programske opreme dejansko realizira v skupek programske kode.
- *Testiranje*. Da zagotovimo ustreznost delovanja programske opreme, je potrebno glede na opredeljene zahteve in omejitve nastali rezultat dobro preizkusiti.
- *Vzdrževanje*. Faza vzdrževanja vključuje popravljanje napak, ki niso bile odkrite v prejšnjem koraku ter izboljševanje in dograjevanje programske kode v skladu s spreminjajočimi se zahtevami.

Ključna pomanjkljivost tradicionalnega pristopa k razvoju programske opreme je, da ne dopušča veliko sprememb in premislekov v kasnejših fazah razvoja. Na primer, ko je programska oprema že v fazi testiranja in se ugotovijo večje pomanjkljivost v prvotni zasnovi, se je zelo težko vrniti v fazo analize ter korenito spremeniti specifikacijo programske opreme.

1.2 Agilen pristop

Ker živimo v globalno povezanem in hitro se spreminjajočem okolju, morajo biti podjetja, da ostanejo konkurenčna, pripravljena se hitro prilagoditi novim razmeram in pogojem. Če se pojavi konkurenčna ponudba oziroma produkt ali se ponuja priložnost na novonastalem trgu, se mora podjetje hitro odzvati, da lahko ostane konkurenčno. Ker je programska oprema ključnega pomena za delovanje podjetij, se je moral prilagoditi tudi pristop k njenemu razvoju. Zmožnost hitrega razvoja in dostave programske opreme se je izkazala kot kritična prednost, zato so podjetja postala pripravljena zamenjati kvaliteto programske opreme z njeno čim hitrejšo dostavo (Sommerville, 2016).

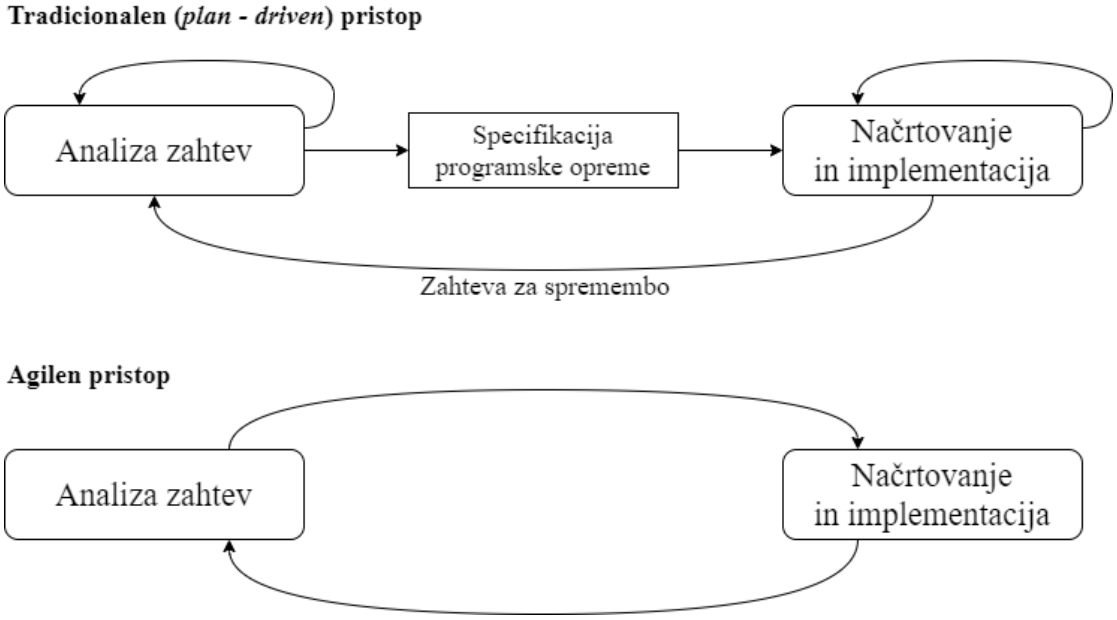
Za zagotovitev potreb nove dinamike tradicionalen pristop k razvoju programske opreme ni ustrezen, saj ne dopušča konstantne spremembe specifikacij v kasnejših fazah razvoja. V ta namen se je na prelomu stoletja izoblikoval agilen pristop k razvoju programske opreme, ki dopušča večjo prilagodljivost skozi celoten proces in bolj ustreza dinamiki novonastalega okolja. Glavni koncepti agilnega pristopa oziroma smernice za nov način razmišljanja pri razvoju programske opreme so (Beck in drugi, 2001):

- Prostor za inovacije pred skladnostjo s procesom.
- Delujoča programska oprema pred obsežno dokumentacijo.
- Sodelovanje s strankami pred pogodbenim pogajanjem.
- Odziv na spremembe pred sledenjem načrtu.

Agilen pristop omogoča razvijalcem, da se, namesto na obsežno načrtovanje in pisanje dokumentacije, osredotočijo na razvijanje programske opreme. Tako se lahko strankam hitreje dostavi delujoče funkcionalnosti in pridobi povratne informacije neposredno od končnega uporabnika. Uporabniki nato predlagajo nove funkcionalnosti ali spremembe obstoječih, ki se zajamejo v naslednji iteraciji procesa. Cilj je izogibanje pisanju preobsežne

dokumentacije, ki upočasnjuje izvajanje celotnega procesa. Slika 2 prikazuje ključno razliko med tradicionalnim in agilnim pristopom k razvoju programske opreme.

Slika 2: Ključna razlika med tradicionalnim in agilnim pristopom



Prirejeno po Sommerville (2016).

Pri tradicionalnem pristopu se iteracije v procesu izvajajo znotraj aktivnosti, podpisani formalni dokumenti pa se uporabljajo za komunikacijo med aktivnostmi. Na primer, po končani analizi zahtev naročnika se pripravi dokument, ki vsebuje vso potrebno specifikacijo programske opreme. Ta dokument se nato uporabi kot vhod za aktivnosti načrtovanja in implementacije. Pri agilnem pristopu pa se iteracije v procesu izvajajo skozi več aktivnosti. Aktivnosti analize zahtev in načrtovanja rešitve se izvajajo skupaj in ne ločeno.

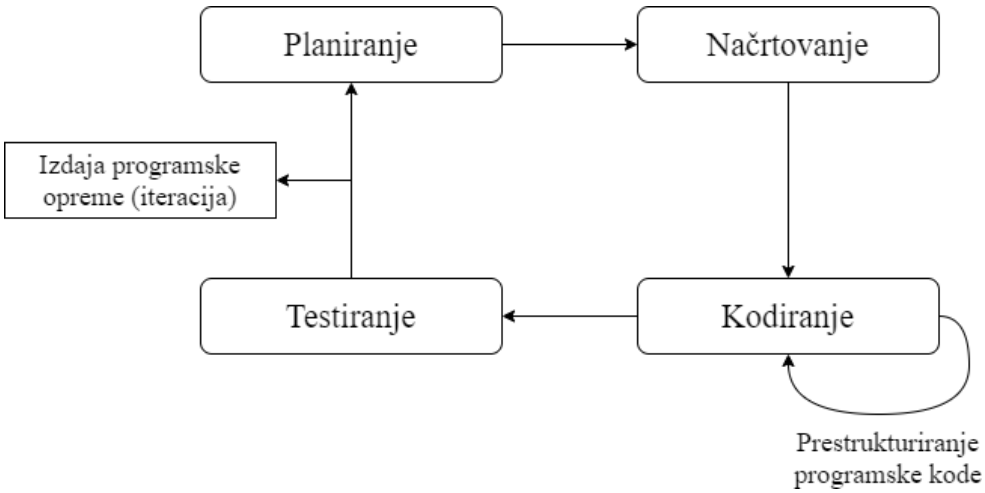
Večina manjših in mikro podjetij, ki razvijajo profesionalno programsko opremo, prakticira agilen pristop k razvoju programske opreme (Sommerville, 2016). Skozi leta se je izoblikovalo več različnih agilnih praks. Vse imajo naslednje skupne značilnosti:

- Aktivnosti analize, načrtovanja in implementacije so prepletene. Obsežnih in podrobnih specifikacij programske opreme ni, ostala dokumentacija je minimizirana oziroma jo samodejno ustvari programsko okolje, v katerem se izvede implementacija. Opredeljene zahteve končnih uporabnikov služijo kot okvir za implementacijo glavnih značilnosti programske opreme.
- Programska oprema se razvija v korakih oziroma iteracijah, v katerih sodelujejo naročnik in končni uporabniki. Ti predlagajo spremembe programske opreme in opredelijo nove funkcionalnosti, ki se nato vključijo v razvoj naslednje iteracije. V vsaki iteraciji se oceni trenutno stanje programske opreme in se da povratna informacija razvijalcem.
- Obsežna uporaba orodij, ki olajšujejo proces razvoja programske opreme. Največkrat uporabljena orodja so orodje za samodejno testiranje programske kode, orodje za

vodenje konfiguracij distribucij programske opreme, orodja, ki omogočajo enostavnejšo integracijo s preostalimi sistemi ter orodja za samodejno izdelavo uporabniških vmesnikov.

Najbolj razširjena praksa agilnega pristopa k razvoju programske opreme je ekstremno programiranje (angl. Extreme Programming, v nadaljevanju XP). Slika 3 prikazuje diagram prakse XP in njene štiri glavne aktivnosti.

Slika 3: Diagram prakse XP



Prirejeno po Pressman (2010).

Praksa opisuje pet načel, ki predstavljajo temelj za vso opravljeno delo znotraj XP (Beck, 2004):

- *Komunikacija*. Da dosežemo učinkovito komunikacijo med razvojnim podjetjem in naročnikom programske opreme, XP poudarja tesno, a neformalno sodelovanje med razvijalci in končnimi uporabniki, vzpostavitev skupnih izrazov za komuniciranje pomembnih konceptov, konstantne povratne informacije ter izogibanje obsežni dokumentaciji, ki bi služila kot komunikacijski medij.
- *Preprostost*. Da dosežemo preprostost, XP omejuje razvijalce na način, da načrtujejo samo takojšnje potrebe in ne tistih potreb, ki jih bo treba upoštevati šele v prihodnje. Namen je ustvariti preprost načrt, ki se lahko enostavno implementira v programsko kodo. Če je potrebno arhitekturno zasnovno izboljšati, lahko kasneje programsko kodo prestrukturiramo.
- *Povratne informacije*. Z načrtovanjem in izvajanjem učinkovite strategije testiranja programske kode razvijalci pridobijo povratne informacije glede pravilnosti delovanja programske opreme. Z izvajanjem testiranja končnih uporabnikov pa razvijalci pridobijo povratne informacije glede ustreznosti delovanja funkcionalnosti. Ta informacija se uporabi kot vhod v naslednjo iteracijo procesa.
- *Pogum*. Za dosledno upoštevanje XP-načel razvijalci potrebujejo pogum. Morajo biti dovolj disciplinirani, da ne padejo pod pritiskom za načrtovanje prihodnjih potreb,

čeprav vedo, da lahko sprememba določene potrebe pomeni drastično prestrukturiranje že implementirane programske kode.
- *Spoštovanje*. Z upoštevanjem vseh opisanih XP-načel, agilna razvojna ekipa vzbudi medsebojno spoštovanje. Ne samo med razvijalci, temveč tudi z naročnikom ter posredno tudi s programsko kodo. Z uspešnim sledenjem XP-načel narašča tudi spoštovanje do celotnega procesa.

V nadaljevanju so opisane glavne značilnosti ter seznam predlaganih tehnik v kontekstu XP (Pressman, 2010):

- Aktivnost načrtovanja se prične s poslušanjem in zajemom zahtev, ki omogoča razvijalcem, da dojamejo širši kontekst poslovnega problema in razumejo vsebino uporabnikovih potreb. Poslušanje vodi do seznama uporabniških zgodb (angl. user stories), ki opisujejo funkcionalnosti programske opreme. Uporabniške zgodbe napiše stranka oziroma končni uporabnik in vsaki dodeli določeno prioriteto, odvisno od doprinosa poslovne vrednosti vseh vključenih funkcionalnosti. Razvijalci nato ocenijo, koliko časa bo potrebno za razvoj posamezne uporabniške zgodbe.
- Aktivnost načrtovanja v XP daje prednost preprostim arhitekturnim zasnovam. Načrtujejo se samo tiste uporabniške zgodbe, ki so vključene v trenutni iteraciji procesa. Kadar se pojavi potreba po izboljšavi zasnove, se izdela načrt za prestrukturiranje programske kode (angl. refactoring). XP tudi poudarja, da se aktivnost načrtovanja izvaja pred in po začetku programiranja, saj se vzroki in ideje za prestrukturiranje kode največkrat pojavijo med programiranjem.
- Ključen koncept med aktivnostjo programiranja v XP je programiranje v paru (angl. pair programming), ki priporoča, da dva razvijalca sočasno programirata na eni delovni postaji. Takšen koncept zagotavlja reševanje problemov v realnem času, saj dve glavi pogosto več vesta kot samo ena, prav tako zagotavlja tudi ustrezno kakovost programske kode, saj se koda pregleduje med pisanjem. Tako se razvijalci lažje osredotočijo na trenutni problem.
- Med aktivnostjo testiranja v XP se izvajajo testi posameznih delov programske kode, ki jih razvijalci napišejo pred začetkom programiranja (angl. test-first development). Cilj je, da celoten mehanizem testiranja podpira orodje, ki povezuje vse teste v celoto in je sposobno povsem avtomatizirati postopke izvajanja. Tako imajo razvijalci celovit pregled nad trenutnim stanjem programske kode, prav tako lahko izvajajo teste pogosteje in brez dodatnega napora.

1.2.1 Uporabniške zgodbe

Zahteve programske opreme se vedno spreminjajo. V agilnem pristopu za obvladovanje teh sprememb ni samostojne oziroma namenske aktivnosti, temveč je opredelitev zahtev integrirana v razvoj. Ideja je, da končni uporabniki skupaj z razvijalci izdelajo scenarij, ki predstavlja potek njihovega dela. Scenarij je sestavljen iz kratkih zapisov, ki zajemajo

naročnikove potrebe. Ko so uporabniške zgodbe opredeljene, jih razvijalci razbijejo na opravila in ocenijo, koliko napora bo potrebnega za implementacijo. Stranka nato dodeli prioriteto uporabniškim zgodbam. Namen je čimprej zagotoviti tiste funkcionalnosti, ki so nujno potrebne za podporo poslovanja stranke. V naslednji iteraciji razvoja se lahko ustvarijo nove zgodbe, popravijo obstoječe ali spremenijo prioritete. Posledično se tudi ponovno oceni napor.

Prednost uporabniških zgodb je, da jih ljudje lažje razumejo kot tradicionalno obsežno dokumentacijo. Prav tako omogočajo, da se čim več ljudi vključi v predlaganje in opredeljevanje zahtev. Slabost pa je njihova nepopolnost. Težko je oceniti, ali so bile z njimi zajete vse ključne funkcionalnosti sistema. Prav tako je težko presoditi, ali zgodba daje realno sliko stanja. Izkušenejši končni uporabniki lahko med pisanjem zgodbe spustijo določene funkcionalnosti, saj so tako vpeljani v potek dela, da se jim določene stvari zdijo samoumevne (Sommerville, 2016).

Po analizi novejše literature na tem področju, v nadaljevanju povzemam ključne ugotovitve:

- Kar 90 odstotkov razvijalcev, ki prakticirajo agilen pristop, uporabljajo uporabniške zgodbe za zajem zahtev svojih strank (Dalpiaz & Brinkkemper, 2018). Podatki kažejo, da se razvijalci strinjajo, da uporaba uporabniških zgodb izboljša njihovo produktivnost in končno kakovost programske opreme (Lucassen, Dalpiaz, van der Werf & Brinkkemper, 2016).
- Pravilno razumevanje uporabniške zgodbe zahteva tudi razumevanje njenih odvisnosti in povezav z ostalimi zgodbami. Pomanjkanje takšnega vpogleda velikokrat vodi do preslabega razumevanja konteksta uporabniške zgodbe. Raziskava v ta namen predlaga uporabo modelov poslovnih procesov. Raziskovalci so empirično dokazali, da povezovanje uporabniških zgodb z modeli poslovnih procesov pozitivno vpliva na boljše razumevanje vrstnega reda izvajanja, medsebojne odvisnosti in povezanosti ter celostnega konteksta uporabniških zahtev. Predpogoj je, da so poslovni procesi že zmodelirani (Trkman, Mendling & Krisper, 2016).
- Kadar so uporabniške zgodbe nepopolne ali na grobo napisane, je težje predvideti njihove odvisnosti od ostali zgodb. Raziskava ugotavlja, da se, kadar ima zgodba ocenjen napor na štiri ali več dni, pogosto pojavijo problemi pri implementaciji. Zato raziskovalci poudarjajo, da je primerna granulacija pomemben koncept pri ustvarjanju uporabniških zgodb (Liskin, Pham, Kiesling & Schneider, 2014).
- Mnoga manjša razvojna podjetja velikokrat nimajo časa in virov, da bi spremenila svoj proces razvoja programske opreme. V nekaterih primerih se podjetja niti ne zavedajo, da so številni vidiki agilnega pristopa že vzpostavljeni. Študija primera na manjšem razvojnem podjetju, v kateri so raziskovalci vpeljali uporabniške zgodbe kot začetek prehoda na agilen pristop, je pokazala korist in zadovoljstvo pri zaposlenih ter predlaga uporabniške zgodbe kot primeren začetek za prehod na agilen pristop (Diebold, Theobald, Wahl & Rausch, 2018).

1.2.2 Prestrukturiranje programske kode

Eno izmed temeljnih pravil tradicionalnega pristopa k razvoju programske opreme je predvidevanje prihodnjih sprememb že pri načrtovanju programske opreme, zato da bo implementacija teh sprememb v prihodnje enostavno izvedljiva. XP-praksa to pravilo zavrača, saj takšno načrtovanje ni vredno časa in napora. Pričakovane spremembe se pogoste ne uresničijo ali pa se pojavijo povsem nove potrebe po spremembah.

V praksi je sprememba programske opreme neizbežna. Za lažjo izvedbo sprememb XP-praksa predlaga konstantno prestrukturiranje programske kode. To pomeni, da razvojna ekipa poišče možnosti izboljšave programske opreme ter jih takoj implementira. Tudi v primeru, kadar ni nujne potrebe za izboljšavo. Temeljna lastnost programske opreme je, da se s spremembami poslabša njena struktura. Posledično je vsako nadaljnjo spremembo težje implementirati. Prestrukturiranje programske kode izboljša strukturo in berljivost programske opreme, kar pripomore k lajšanju implementacije prihodnjih sprememb (Sommerville, 2016). Kadar je prestrukturiranje programske kode konstanten del procesa razvoja, je načeloma programska oprema vedno enostavna za razumevanje in spreminjanje. Vendar pa se s prestrukturiranjem v praksi velikokrat zamuja, saj se zaradi časovnih pritiskov napor namenja raje implementaciji novih funkcionalnosti.

Že pretekle raziskave agilnih pristopov v manjših podjetjih nakazujejo, da prestrukturiranje programske kode pozitivno vpliva na produktivnost razvijalcev in kvaliteto programske opreme (Moser, Abrahamsson, Pedrycz, Sillitt & Succi, 2008). Po analizi novejše literature na tem področju povzemam glavne ugotovitve:

- Cilj agilnega razvojnega podjetja je razviti programsko opremo, ki jo bo mogoče čim hitreje dostaviti do stranke in ki bo prinašala poslovno vrednost na dolgi rok. Za doseganje tega cilja je potrebno minimizirati stroške vzdrževanja. S tem povezan je pojav arhitekturnega tehniškega dolga (angl. architectural technical debt). Če arhitektura programske opreme ni optimalna za doseganje dolgoročnih poslovnih ciljev, jo je potrebno prestrukturirati. Raziskava je pokazala, da je koncept modularnosti ključnega pomena pri prestrukturiranju programske kode, saj omogoča poseg v posamezne module programske opreme, ne da s tem ogrozimo delovanje celotnega sistema. Raziskovalci so ugotovili, da modularna arhitektura eliminira tehniške dolgove na dolgi rok, kar lahko podjetju privarčuje več mesecev razvoja in vzdrževanja (Martini, Sikander & Madlani, 2018).
- Raziskava, ki proučuje načrtovanje in uporabo prestrukturiranja programske kode v razvojnih podjetjih, je ugotovila, da ni splošno sprejete strategije. Mnenja o pravem času za izvedbo prestrukturiranja so različna. Vedno se je potrebno odločiti med hitro dostavo funkcionalnosti na eni strani ter kakovostjo programske opreme na drugi strani. Ugotovitve potrjujejo, da se večina podjetij strinja, da mora imeti prestrukturiranje visoko prioriteto, ampak tudi priznavajo, da so za odločitve med načrtovanjem prestrukturiranja največkrat odgovorni poslovni pritiski, ne glede na prioriteto. Nekatere

ekipe, namesto da bi takoj izvedle prestrukturiranje, raje počakajo na poslovno upravičeno priložnost, torej počakajo na naročilo novih zahtev. Ali pa celo počakajo, da produktivnost ekipe pade na poslovno neupravičen nivo. Raziskovalci prav tako ugotavljajo, da je odločitev odvisna tudi od položaja odgovorne osebe in njene vpletenosti v poslovni oziroma operativni vidik (Chen, Xiao, Wang, Osterweil & Li, 2014).

1.2.3 Programiranje v paru

Ideja programiranja v paru je, da razvijalci razvijajo programsko opreme v dvoje, da sedijo pred isto delovno postajo in skupaj rešujejo probleme. Praksa tudi uči, da je potrebno pare ustvariti dinamično, torej da ne programirata skupaj vedno isti dve osebi, temveč se skozi proces osebe menjavajo, tako da vsi razvijalci sodelujejo z vsemi. Takšna tehnika razvoja ima naslednje prednosti (Sommerville, 2016):

- Podpira koncept o kolektivnem lastništvu (angl. collective ownership) in odgovornosti za programsko opremo. To odraža idejo o nesebičnem programiranju, ki sega v začetke razvoja programske opreme (Weinberg, 1971). Ideja je, da je programska oprema v celoti v lasti celotne ekipe, ne pa posameznikov. Kadar obstaja težava v kodi, je za njo kolektivno odgovorna ekipa, ne pa dejanski avtor.
- Deluje kot neformalni pregled programske kode, saj vsako vrstico pregledata vsaj dve osebi. Pregled programske kode je učinkovita tehnika za odkrivanje napak v programski opremi. Vendar pa je lahko tudi zamudna, kadar se izvaja v neprimerno formalni obliki. Programiranje v paru verjetno odkrije manj napak kot formalni pregled programske kode, je pa cenejše in lažje za izvajanje.
- Spodbuja prestrukturiranje programske kode. V tradicionalnem procesu se lahko razvijalca, ki vlaga napor v prestrukturiranje, oceni kot manj učinkovitega od tistega, ki preprosto razvija funkcionalnosti. Kadar uporabimo programiranje v paru in kolektivno lastništvo, so koristi prestrukturiranja takoj vidne, zato razvijalci hitreje podprejo celoten proces.

Nekatera podjetja, ki so uvedla agilni pristop, so zadržana do programiranja v paru oziroma ga sploh ne prakticirajo. Predpostavka je, da je neučinkovito, saj par razvijalcev v določenem času napiše dvakrat manj kode, kot dva razvijalca, ki individualno pišeta kodo. Druga podjetja mešajo programiranje v paru z individualnim programiranjem, tako da izkušenejši razvijalec razvija skupaj z manj izkušenim, kadar ima ta težave (Sommerville, 2016).

Že pretekle raziskave so pokazale, da je produktivnost programiranja v paru v splošnem primerljiva z individualnim programiranjem (Williams, Kessler, Cunningham & Jeffries, 2000). Izpostavljeni razlogi so, da par razvijalcev pred začetkom razvoja najprej razpravlja o programski opremi, kar zmanjšuje premisleke med razvojem. Prav tako se zmanjša število napak in posledično tudi stroški vzdrževanja, saj se vzporedno izvaja neformalni pregled

programske kode. V primeru zelo izkušenih razvijalcev pa so raziskave pokazale ravno nasprotno (Arisholm, Gallis, Dybå & Sjøberg, 2007). Ugotovili so koristi pri kakovosti programske opreme, vendar to ni v celoti nadomestilo izgubo produktivnosti. Kljub temu je prenos znanja med programiranjem v paru ključnega pomena.

Čeprav je prenos znanja prisoten v vseh oblikah programiranja v paru, pride najbolj do izraza v paru, kjer je občutna razlika v izkušnjah in znanju posameznikov. Raziskava, ki proučuje, kako se znanje prenaša, je ugotovila, da novinec največ pridobi, kadar ekspert razlaga svoj miselni proces. Čeprav se morajo eksperti dodatno potruditi za prenos znanja, je takšna tehnika koristna tudi za njih, saj jim postavljanje vprašanj novincev pomaga razmišljati o svojih praksah in se tako naučiti iz izkušnje (Plonka, Sharp, Van Der Linden & Dittrich, 2015).

2 VZDRŽEVANJE PROGRAMSKE OPREME

Zgodovinsko gledano je med razvojem in vzdrževanjem programske opreme vedno obstajala določena meja. Ljudje so na razvoj gledali kot ustvarjalen proces, ki poteka od zametka osnovnega koncepta do delujočega sistema v končnem okolju. Vzdrževanje pa je veljajo kot dolgočasen in manj zahteven proces od razvoja. Ker je zelo malo sistemov povsem novih, je na razvoj in vzdrževanje programske opreme veliko bolj smiselno gledati kot kontinuiteto. Bolje kot o dveh ločenih procesih je realistično razmišljati o evoluciji programske opreme (angl. software evolution) oziroma o neprekinjenem postopku, v katerem se programska oprema, zaradi spreminjajočih se potreb in zahtev strank, neprestano spreminja (Sommerville, 2016).

2.1 Evolucija programske opreme

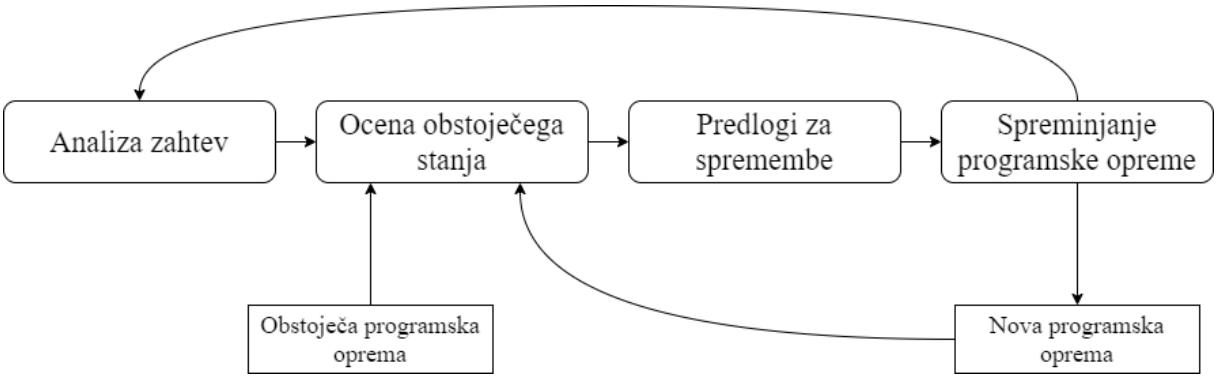
Evolucija programske opreme je veda na področju znanosti računalništva in informatike, ki se ukvarja s preučevanjem spreminjanja in prilagajanja programske opreme, kot posledica različnih zunanjih in notranjih dejavnikov (Herraiz, Rodriguez, Robles & Gonzalez-Barahona, 2013). Vodilni znanstveniki na tem področju si že vrsto let prizadevajo razviti enotno teorijo, ki bi nudila znanstveno podlago za preučevanje programske opreme (Lehman & Ramil, 2001). Predlagana teorija temelji na osmih zakonih evolucije programske opreme (angl. Lehman's Laws of Software Evolution) (Lehman, 1996):

1. *Neprestane spremembe* (angl. continuing change). Programska oprema, ki se uporablja v realnem okolju, se mora neprestano spreminjati, sicer postane postopoma manj uporabna.
2. *Povečevanje zapletenosti* (angl. increasing complexity). Spreminjanje programske opreme povečuje njeno zapletenost. Za ohranjanje in poenostavljanje arhitekturne zasnove je potreben dodaten napor.

3. *Samoregulacija* (angl. self regulation). Proces evolucije programske opreme je uravnotežen. Značilnosti, kot so velikost, čas med posameznimi izdajami in število javljenih napak, so približno konstantne za vsako izdajo programske opreme.
4. *Ohranjanje organizacijske stabilnosti* (angl. conservation of organisational stability). Stopnja aktivnosti ostaja približno konstantna skozi celoten proces evolucije programske opreme. Prav tako ostaja neodvisna od virov, namenjenih v razvoj programske opreme.
5. *Ohranjanje razpoznavnosti* (angl. conservation of familiarity). Količina spremembe programske opreme ostaja konstantna skozi celoten proces evolucije programske opreme. Ob preveliki spremembi se lahko zmanjša zmožnost obvladovanja programske opreme ter razpoznavnost s strani udeležencev v procesu.
6. *Neprestana rast* (angl. continuing growth). Funkcionalnosti, ki jih nudi programska oprema, morajo neprestano rasti, da ohranijo zadovoljstvo končnih uporabnikov.
7. *Padanje kakovosti* (angl. declining quality). Kakovost programske opreme pada, če se slabo vzdržuje in ne prilagaja na spremembe operativnega okolja.
8. *Povratne informacije* (angl. feedback system). Za bistveno izboljšanje je potrebno proces evolucije programske opreme obravnavati kot večnivojski sistem s ponavljajočimi zankami, ki udeležencem nudijo povratne informacije.

Opredeljeni zakoni torej predpisujejo, da so spremembe okolja, v katerem deluje programska oprema ter spremembe uporabniških zahtev, neizogibne. Programska oprema modelira del realnosti in realnost se neprestano spreminja. Torej so mora spreminjati tudi programska oprema (van Vliet, 2007). Slika 4 prikazuje diagram evolucije programske opreme.

Slika 4: Diagram evolucije programske opreme



Prirejeno po Sommerville (2016).

2.2 Stroški vzdrževanja programske opreme

V splošnem obstajajo tri različne kategorije vzdrževanja programske opreme (Lientz & Swanson, 1980):

- *Popravljanje* (angl. corrective maintenance). Ukvarjanje s popravljanjem napak v sistemu. Napake pri programiranju lahko poceni popravimo. Napake pri načrtovanju pa

so dražje, ker lahko vključujejo potrebo po ponovnem programiranju posamezne komponente sistema. Najdražje je popravilo napak pri zahtevah, saj lahko pomenijo obsežno prenovo sistema.

- *Prilagajanje* (angl. adaptive maintenance). Ukvarjanje s prilagajanjem sistema na spremembo operativnega okolja. Takšno vzdrževanje je največkrat potrebno, kadar pride do spremembe strojne opreme ali kadar se posodobi operacijski sistem oziroma platforma, na kateri se izvaja programska oprema. Prilagajanje ne vodi v spremembo funkcionalnosti programske opreme.
- *Izpopolnjevanje* (angl. perfective maintenance). Dopolnjevanje obstoječih ter dodajanje novih funkcionalnosti programske opreme. Zaradi organizacijskih sprememb ali poslovnih zahtev se spremenijo uporabniške potrebe, zato je potrebno spremeniti programsko opremo na način, da podpira nove funkcionalnosti. Ta vrsta vzdrževanja je običajno najobsežnejša. Vključuje tudi aktivnosti za izboljšanje delovanja sistema, kot je na primer pohitritev določenih funkcionalnosti.
- *Preprečevanje* (angl. preventive maintenance). Ukvarjanje z obvladljivostjo (angl. maintainability) sistema. To običajno vključuje posodabljanje tehniške dokumentacije, dodajanje komentarjev v izvorno kodo ali izboljševanje modularne strukture sistema.

V praksi ni povsem strogega razlikovanja med naštetimi vrstami vzdrževanja programske opreme. Ko sistem prilagajamo novemu okolju, lahko hkrati dodamo funkcionalnosti, ki izkoriščajo nove funkcije operativnega okolja (Sommerville, 2016).

Najnovejša razpoložljiva dolgoročna študija, ki proučuje stroške vzdrževanja programske opreme, je ugotovila, da se je porazdelitev stroškov zelo malo spremenila v zadnjih tridesetih letih (Davidsen & Krogstie, 2010). Avtorji so ugotovili, da popravljanje ni največji delež vzdrževanja programske opreme, temveč je izpopolnjevanje programske opreme. Delež popravljanja je približno 24 odstotkov, delež prilaganja približno 19 odstotkov ter delež izpopolnjevanja približno 58 odstotkov. Čeprav ni novejših podatkov, vse kaže na to, da ugotovljena porazdelitev velja še danes (Sommerville, 2016).

Empirični zgodovinski podatki kažejo, da je delež stroškov, povezanih z evolucijo programske opreme, običajno med 60 in 90 odstotkov vseh stroškov, ki nastanejo med celotno življenjsko dobo določene programske opreme (Erlikh, 2000).

Podobno stanje nakazuje študija, ki preučuje razmerje med zaposlenimi, ki se ukvarjajo z razvojem programske opreme, in zaposlenimi, ki se ukvarjajo z vzdrževanjem programske opreme. Približno 75 odstotkov vseh zaposlenih naj bi se ukvarjalo samo z vzdrževanjem programske opreme. Ta delež se je skozi zgodovino povečeval in študija razlaga, da zaenkrat ni kazalnikov, ki bi predvidevali zmanjševanje tega deleža (Jones, 2006).

Vzdrževanje programske opreme je torej dražje od njenega začetnega razvoja. Dodajanje novih funkcionalnosti v sistem je običajno cenejše med začetnim razvojem. Razlogi za dražje vzdrževanje so (Sommerville, 2016):

- Novi zaposleni v podjetju morajo najprej spoznati in razumeti programsko opremo, ki jo podjetje vzdržuje. Nekaj povsem običajnega je, da razvijalci, ki so bili zadolženi za razvoj programske opreme, niso več zaposleni v tem podjetju ali pa delajo na novih projektih. Novozaposleni, ki so zadolženi za vzdrževanje programske opreme, pa nimajo zadostnega razumevanja za vse odločitve, ki so bile sprejete med razvojem. Zato je najprej potreben čas, da se osebje priuči vseh konceptov obstoječe programske opreme, preden jo lahko prične spreminjati.
- Če ločimo ekipo za razvoj in vzdrževanje, razvojna ekipa nima zadostne motivacije za razvoj obvladljivega in vzdržljivega sistema. Pogodba za razvoj programske opreme je običajno ločena od pogodbe za vzdrževanje. Zato je lahko že pred razvojem jasno, da bo za vzdrževanje programske opreme zadolženo drugo podjetje. V tem primeru razvojno podjetje oziroma ekipa, ki je zadolžena za začetni razvoj, nima koristi, ne glede na trud za razvoj in spremembe programske opreme, ki bi zagotovile večjo obvladljivost in vzdržljivost v prihodnosti. Če ima ekipa možnost za skrajšanje določene aktivnosti, bo to storila, ne glede na posledice oziroma težave, ki se bodo pojavile ob prihodnjih spremembah.
- Vzdrževanje programske opreme ni popularno oziroma ima slabo konotacijo med inženirji programske opreme. Proces vzdrževanja je kvalificiran kot manj zahteven proces, zato za vzdrževanje običajno skrbijo razvijalci z manj izkušnjami. Izkušenejše razvijalce je bolje vključiti v razvoj novih sistemov. Poleg tega so lahko stari sistemi implementirani v zastarelih programskih jezikih, kar pomeni, da se morajo razvijalci za vzdrževanje najprej naučiti te programske jezike.
- S starostjo se zmanjšuje robustnost arhitekture programske opreme. Ko se sistem spreminja, se običajno poslabša njegova struktura. Posledično postane programska oprema težja za razumevanje in spreminjanje. Nekateri sistemi so bili razviti brez sodobnih metod in tehnik razvoja, prav tako so morda bili razviti za čim večjo učinkovitost, ne glede na slabo arhitekturo in težjo razumljivost. Tehniška dokumentacija se lahko izgubi oziroma postane med spreminjanjem nekonsistentna z obstoječim stanjem.

2.3 DevOps

Teorija o evoluciji programske opreme razlaga razvoj in vzdrževanje programske opreme kot združen in neprekinjen proces. V praksi je mejnik med razvojem in vzdrževanjem običajno prehod v produkcijo, ki pa je vedno veljal za najbolj kritičen del procesa, saj zahteva največjo mero pazljivosti in natančnosti. Nepravilnosti ob prehodu v produkcijo lahko imajo hude posledice, običajno v obliki izpada dela sistema ali v nepredvidenih napakah obstoječih funkcionalnosti programske opreme, kar pa vedno pomeni večanje stroškov vzdrževanja (Kim, Debois, Willis & Humble, 2016).

Če želimo zabrisati mejo med razvojem in vzdrževanjem, sta najbolj očitni dve rešitvi. Bodisi vzpostavimo okolje in delovni proces na način, da prehod v produkcijo ne predstavlja

posebnega izziva, bodisi izvajamo prehod v produkcijo tako pogosto, da postane integriran del vsakdanje rutine. To najbolje razloži DevOps, eden izmed najpopularnejših konceptov zadnjega desetletja v IT-sektorju.

Preden lahko definiram, kaj DevOps pomeni, je potrebno opredeliti, kaj predstavlja posamezni del izraza (Sharma, 2017):

- *Dev* (angl. development) predstavlja ekipo oziroma osebe, zadolžene za razvoj programske opreme. Njihov glavni cilj je čim hitreje ustvariti in dostaviti nove funkcionalnosti h končnim uporabnikom.
- *Ops* (angl. IT operations) predstavlja ekipo oziroma osebe, zadolžene za delovanje informacijskega okolja. Njihov glavni cilj je zagotoviti, da imajo končni uporabniki dostop do zanesljivega, hitrega in varnega sistema.

Pred prihodom agilnih metod nasprotje navedenih ciljev ni predstavljalo problema, saj so *Dev* in *Ops* ekipe delovale povsem ločeno. Srečali so se poredko, običajno samo ob prehodih v produkcijo. Danes ni več tako, frekvenca in količina novih funkcionalnosti *Dev* ekipe je vse večja, kar pa predstavlja problem za *Ops* ekipo. Ta jih mora zanesljivo in hitro dostaviti strankam, ob tem pa vseskozi ohranjati stabilnost in varnost obstoječih sistemov (Sharma, 2017).

S pomočjo DevOps skušamo doseči ravnovesje med inovacijami in stabilnostjo ter hitrostjo dostave in kakovostjo. Da to dosežemo, pa morajo *Dev* in *Ops* ekipe izboljšati svoj način delovanja ter se med sabo uskladi (Sharma, 2017). V nadaljevanju navajam nekaj definicij pojma DevOps:

- DevOps je kulturno gibanje, ki spreminja način razmišljanja posameznikov o svojem delu, ceni raznolikost opravljenega dela, podpira procese, ki pospešujejo stopnjo, s katero podjetje realizira vrednost ter meri učinek družbenih in tehničnih sprememb. Je način razmišljanja in dela, ki organizacijam omogoča razvoj in ohranjanje trajnostnih delovnih praks. Nudi okvir za iskanje inovativnih struktur in tehnologij za učinkovitejšo delo (Davis & Daniels, 2016).
- DevOps je gibanje, ki spodbuja večje sodelovanje med vsemi vključenimi pri dostavi programske opreme na način, s katerim podjetje hitreje in zanesljivejše izda programsko opremo. To lahko dosežemo tako, da vpeljemo agilne pristope tudi v svet sistemske administracije in upravljanja infrastrukture (Humble & Farley, 2010).
- DevOps je manifestacija ustvarjanja dinamičnih, učečih se organizacij, ki nenehno krepijo zaupanja vredne norme (Kim, Debois, Willis & Humble, 2016).

Vse prakse in tehnike, ki jih lahko danes opazimo pri izvajanju DevOps, temeljijo na sklopu treh osnovnih načel (Kim, Behr & Spafford, 2013):

- Načelo toka (angl. the principle of flow).

- Načelo povratnih informacij (angl. the principle of feedback).
- Načelo nenehnega učenja in eksperimentiranja (angl. the principle of continual learning and experimentation).

Načelo toka (Slika 5) omogoča hiter pretok dela od leve proti desni, od razvoja (*Dev*) do sistemske administracije (*Ops*) in nato do stranke. Da povečamo pretok, mora biti naše delo vidno, zmanjšati moramo dolžino iteracij in velikost posameznih izdaj programske opreme. S preprečevanjem napak v smeri pretoka moramo vgraditi kvaliteto v naše delo ter ga nenehno optimizirati za doseganje zastavljenih poslovnih ciljev (Kim, Debois, Willis & Humble, 2016).

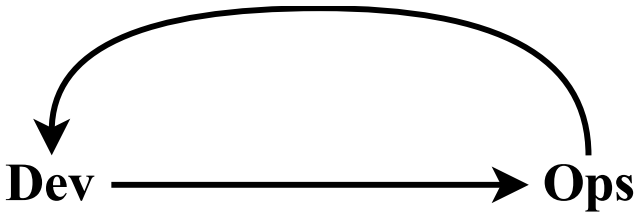
Slika 5: Načelo toka

Prirejeno po Kim, Debois, Willis & Humble (2016).

S pospeševanjem pretoka preko uporabe tehnologije lahko zmanjšamo čas, ki je potreben za izpolnitev internih ali strankinih zahtev. Na takšen način lahko še posebej zmanjšamo čas, ki je potreben za dostavo programske opreme v produkcijsko okolje. Tako lahko povečamo kvaliteto dela in zmogljivosti ekipe, kar daje poslovno vrednost razvojnemu podjetju (Kim, Debois, Willis & Humble, 2016). Najpogosteje uporabljene prakse, ki so se razvile na načelu pretoka, so zvezna integracija (angl. continuous integration) in zvezna dostava (angl. continuous delivery).

Načelo povratnih informacij (Slika 6) omogoča konstanten tok informacij od desne proti levi ter na vseh korakih pretoka. Načelo zahteva, da povečamo količino pridobljenih povratnih informacij, saj lahko tako preprečimo ponavljanje napak ter pohitrimo njihovo zaznavo in odpravljanje. Tako zagotovimo kvaliteto že pri viru in se osredotočimo na ustvarjanje dodatne poslovne vrednosti (Kim, Debois, Willis & Humble, 2016).

Slika 6: Načelo povratnih informacij



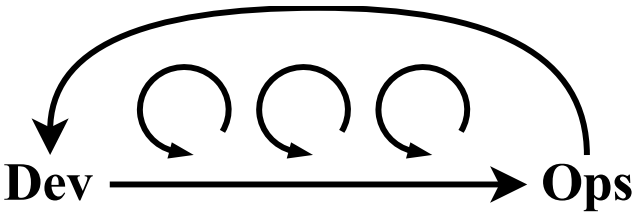
Prirejeno po Kim, Debois, Willis & Humble (2016).

Prav tako z načelom povratnih informacij ustvarimo varnejšo in zanesljivejšo programsko opremo, v kateri lahko najdemo napake še pred pojavom kakšne usodne težave. Z zaznavanjem napak, takoj ko se pojavijo, lahko vzpostavimo potrebne protiukrepe ter

povratne zanke za njihovo odpravo. Takšen pristop je tudi jedro večine sodobnih metodologij za izboljšave procesov. Opisan pristop maksimira možnosti razvojnega podjetja za učenje in izboljšanje (Kim, Debois, Willis & Humble, 2016).

Načelo nenehnega učenja in eksperimentiranja (Slika 7) omogoča ustvarjanje kulture, ki spodbuja visoko medsebojno zaupanje ter dinamičen, discipliniran in celo znanstven pristop k preizkušanju novih idej. S tem ustvarjamo okolja in ekipe, ki se znajo učiti iz uspehov in tudi neuspehov (Kim, Debois, Willis & Humble, 2016).

Slika 7: Načelo nenehnega učenja in eksperimentiranja



Prirejeno po Kim, Debois, Willis & Humble (2016).

Z večanjem pretoka od leve proti desni in količine povratnih informacij ustvarimo okolje, v katerem lahko lažje in varneje izvajamo eksperimente, ki nam pomagajo, da se hitreje učimo in smo bolj konkurenčni na trgu. Prav tako načelo nenehnega učenja in eksperimentiranja omogoča vzpostavitev procesov, ki povečujejo učinek na novo pridobljenega znanja. Ne glede na to, od kod je znanje prišlo ali kdo ga bo uporabil (Kim, Debois, Willis & Humble, 2016).

2.3.1 Zvezna integracija

Zvezna (v nekaterih prevodih tudi neprekinjena ali nenehna) integracija je postopek visoko frekvenčne integracije na novo napisane izvorne kode razvijalcev v glavno vejo repozitorija. Namesto da razvijalci tedne ali celo mesece individualno razvijajo funkcionalnosti na neodvisnih vejah repozitorija in svoje spremembe združijo z glavno vejo šele, ko je delo popolnoma dokončano, se izvorna koda združuje od vsaj enkrat dnevno do zvezno, torej ob vsaki minimalni in smiselno zaključeni spremembi. Namreč, daljše kot so periode med združitvami kode, več je sprememb za združiti in posledično tudi več možnosti za težave ob združitvi, saj je težje izolirati in prepoznati vzrok napake pri ogromnem sklopu sprememb kot pa pri manjšem. Cilj zvezne integracije je torej izogniti se vsem vrstam integracijskih problemov, ki izhajajo iz velikih in redkih združitev (Davis & Daniels, 2016).

Običajno se za zagotovitev uspešnih integracij po združitvi izvorne kode v praksi samodejno zažene vrsta testnih postopkih, ki preverijo izvedljivost in pravilnost novega stanja programske opreme. S takšnim načinom dela lahko težave prepoznamo in odpravimo karseda hitro (Davis & Daniels, 2016).

Vpeljava zvezne integracije v proces razvoja in vzdrževanja programske opreme pomeni ustvarjanje nove paradigme razmišljanja v razvojni ekipi. Brez zvezne integracije je programska oprema v nedelujočem oziroma nepravilnem stanju, dokler ne dokažemo nasprotno. Z zvezno integracijo pa je privzeto stanje programske opreme vseskozi delujoče, z določeno stopnjo zaupanja v avtomatizirane postopke, katerih učinkovitost je odvisna od obsega pokritosti s testi. Zvezna integracija ustvarja testno povratno zanko, ki omogoča takojšno identifikacijo težav še v trenutku, ko jih je najceneje odpraviti (Humble & Farley, 2010).

Prakticiranje zvezne integracije tudi zahteva določeno mero discipline znotraj razvojne ekipe. V nadaljevanju so našteta nekatera pravila, ki se jih morajo razvijalci upoštevati za uspešno prakticiranje zvezne integracije (Sharma, 2017):

- Izvorna koda celotne programske opreme (vsi deli in komponente sistema) mora biti samo v enem skupnem repozitoriju.
- Kreiranje rezultatov ob izgradnji izvorne kode v izvršne datoteke, ki so pripravljene za uporabo (v nadaljevanju artefakti), mora biti avtomatizirano.
- Artefakti morajo biti sposobni sami zagnati pripadajoče testne postopke.
- Vsak razvijalec mora potisniti izvorno kodo v glavno vejo repozitorija vsaj enkrat na dan.
- Postopki kreiranja artefaktov se morajo izvesti hitro.
- Testiranje se mora izvajati v okolju, ki je kopija končnega produkcijskega okolja.
- Dostop do artefaktov mora biti omogočen vsem članom ekipe.
- Potrebno je zagotoviti, da vsi člani vidijo, kaj se v določenem trenutku dogaja in kakšno je trenutno stanje.

Že pretekle večletne raziskave so empirično dokazale, da prakticiranje DevOps prakse zvezne integracije pripomore k večanju poslovne vrednost razvojnega podjetja. Raziskovalci so ugotovili, da so podjetja, ki prakticirajo DevOps, veliko bolj produktivna kot podjetja, ki ga ne prakticirajo. Tako so za 30-krat povečali število novih sprememb izvorne kode, imeli so 6-krat več možnosti za uspešen prehod v produkcijo ter v treh letih za 50 odstotkov povečali svojo tržno kapitalizacijo v primerjavi s podjetji, ki ne prakticirajo zvezne integracije (Kim, Debois, Willis & Humble, 2016). V nadaljevanju povzemam še nekaj literature na področju zvezne integracije:

- Raziskava, ki nudi kvantitativno analizo korelacij med velikostjo razvojnega podjetja ter kontinuiteto prakticiranja zvezne integracije, je pokazala, da obstaja očitna negativna korelacija med številom razvijalcev in številom sprememb izvorne kode. Torej, večja kot je razvojna ekipa, manj je sprememb izvorne kode na posameznika v določenem obdobju. Raziskovalci so prav tako ugotovili, da se v večjih podjetjih le redko držijo pravila vsakodnevnih potiskov izvorne kode v glavno vejo repozitorija ter da modularna arhitektura programske opreme močno pripomore k uspešnemu prakticiranju zvezne integracije (Ståhl, Mårtensson & Bosch, 2017).

- Pregledovanje programske kode in zvezna integracija se pogosto prepletata v sodobnem upravljanju kakovosti programske opreme. Študija, v kateri so raziskovalci izvedli eksplorativno analizo vpliva zvezne integracije na pregledovanje programske kode, je ugotovila, da ima avtomatizirano kreiranje artefaktov opazen vpliv na frekvenco pregledovanja programske kode. Večkrat ko se izvedejo samodejni postopki, več je izvedb pregledovanja. Prav tako ima zvezna integracija pozitiven vpliv na kakovost programske opreme. Raziskava ugotavlja, da imajo projekti, kjer se prakticira zvezna integracija, stabilen nivo zagotavljanja kakovosti programske opreme (Rahman & Roy, 2017).

2.3.2 Zvezna dostava

Zvezna dostava (v nekaterih prevodih tudi namestitev) je nadgradnja koncepta zvezne integracije. Ne samo da zagotovimo pravilno delovanje programske opreme ob konstantni integraciji vseh sprememb, temveč da tudi zagotovimo, da je programska oprema vedno v stanju možne dostave v testno ali produkcijsko okolje. Cilj zvezne dostave je čimbolj pogosto ter z avtomatiziranimi postopki dostaviti nove izdaje programske opreme v različna končna okolja (Davis & Daniels, 2016).

To sicer ne pomeni, da se vsak artefakt, ki je rezultat zvezne integracije, tudi dostavi v naslednje okolje preko zvezne dostave, kajti niso vse spremembe pripravljene za nadaljnjo obdelavo. Še posebej, če je frekvenca sprememb v glavno vejo repozitorija visoka in je posamezna funkcionalnost programske opreme sestavljena iz več manjših smiselno zaključenih sprememb (Sharma, 2017).

Cilj zvezne dostave je torej čim hitreje strankam in uporabnikom ponuditi nove funkcionalnosti, ki so jih implementirali razvijalci. V praksi to pomeni ustvarjanje cevovodov (angl. pipelines), ki so zmožni avtomatizirano dostaviti, primerno konfigurirati in namestiti rezultat zvezne integracije v osnovna testna okolja, nato v integracijska okolja in na koncu še v končna produkcijska okolja. Kdaj in koliko sprememb se prestavi v naslednjo okolje, je najpogosteje odvisno od vrste programske opreme ter internih ali eksternih zahtev (Humble & Farley, 2010).

Kadar pa ima razvojno podjetje vzpostavljen proces, kjer se vsaka delujoča sprememba programske opreme samodejno uveljavi tudi v končnem produkcijskem okolju, pomeni, da podjetje prakticira zvezno uvedbo (angl. continuous deployment). Zvezna uvedba je DevOps praksa, ki maksimira načelo toka do svojega ekstrema. V praksi ga večinoma prakticirajo razvojna podjetja, ki uporabljajo SaaS poslovni model. Čeprav je možno prakso izvajati tudi v drugih kontekstih, se je najbolj razvila v svetu spletnih aplikacij. Običajno to pomeni, da podjetje konstantno uvaja spremembe v produkcijsko okolje, le da jih končni uporabniki ne opazijo. Na voljo postanejo šele, ko so uveljavljene vse potrebne spremembe, ki tvorijo novo funkcionalnost programske opreme (Kim, Debois, Willis & Humble, 2016).

Vpeljava DevOps prakse zvezne dostave se je že v zgodnjih poizkusih izkazala kot zelo uspešna. Kadar praksa postane vsakodnevna rutina, se občutno zmanjša napor ob namestitvah novih izdaj programske opreme in prehodih v produkcijo. Z uporabo zvezne dostave lahko razvojno podjetje dostavi vrednost za svoje stranke v minutah, namesto v tednih ali celo mesecih (Kim, Debois, Willis & Humble, 2016). Vendar pa, ker je prakticiranje zvezne dostave relativno nova stvar, še vedno obstajajo ovire in izzivi, s katerimi se podjetja srečujejo pri njeni vpeljavi v proces. Raziskava, ki preučuje, kako lahko razvojna podjetja osvojijo to prakso, predlaga naslednjo strategijo (Chen, 2016):

- Za ključne osebe (vodstvo razvojne ekipe) je potrebno identificirati in izpostaviti vse težave, ki jih lahko zvezna dostava reši. Brez prave podpore je težko doseči vse cilje, ki jih zahteva implementacija zvezne dostave.
- Tvorjenje namenske ekipe iz različnih oddelkov, ki bodo skrbeli za implementacijo zvezne dostave in komunikacijo z ostalimi člani oddelkov.
- Zvezna dostava implementacije zvezne dostave. Učinki prakse (dodajanje poslovne vrednosti razvojnemu podjetju) naj bodo vidni takoj, ko je to mogoče. To pripomore k upravičevanju naložbe v interne izboljšave.
- Pričetek z manj zahtevno, vendar pomembo programsko opremo. Manj zahtevno zato, ker bodo učinki zvezne dostave vidni hitreje, pomembno pa zaradi večje podpore vodstva.
- Transparentnost napredka in rezultatov. Vsi implementirani cevovodi naj bodo vidni in na voljo vsem vpletenim v projekt, tudi kadar še niso dokončani in so v začetni obliki.
- Določitev eksperta za zvezno dostavo, da se vključi v projekt. Ekspert v ekipi pomaga izoblikovati in ohranjati motivacijo ter zagon ekipe za učenje prakse.

3 UPRAVLJANJE KAKOVOSTI PROGRAMSKE OPREME

Eden izmed ključnih dejavnikov, ki ločujejo zelo uspešna podjetja od manj uspešnih, je zavezanost h kakovosti svojega produkta ali storitve. Zavezanost h kakovosti pa omogoča dolgoročno donosnost podjetja (Peters & Waterman, 1982). Za podjetja, ki razvijajo programsko opremo, to še posebej velja, saj že zapletenost ter pogoste spremembe, ki jih potrebno vključiti med razvojem, neposredno vplivajo na kakovost programske opreme. Zato je potrebna nenehna pozornost in ocena kakovosti, če želi podjetje razviti dober produkt. Ta potreba se povečuje tudi zaradi vse večje prisotnosti programske opreme v našem vsakdanjem življenju. Nizkokakovostni produkti ne bodo zadovoljili kupcev, uporabniki pa bodo zanemarili sisteme, ki naj bi bili podpora njihovemu delu (van Vliet, 2007).

Upravljanje kakovosti programske opreme (angl. software quality management) skrbi za zagotavljanje, da razvita programska oprema služi svojemu namenu. Kar pomeni, da mora izpopolnjevati potrebe svojih končnih uporabnikov ter delovati učinkovito in zanesljivo, dostaviti pa jo je potrebno pravočasno in znotraj finančnih okvirjev. Tehnike upravljanja

kakovosti programske opreme imajo svoje korenine v metodah in tehnikah, ki so bile razvite za predelovalno industrijo, kjer sta izraza zagotavljanje kakovosti (angl. quality assurance) in nadzor kakovosti (angl. quality control) pogosto uporabljena. Zagotavljanje kakovosti je opredelitev procesov in standardov, ki naj bi vodili do izdelave visokokakovostnih produktov. Nadzor kakovosti pa je uporaba teh procesov za identifikacijo in odstranjevanje produktov, ki ne dosegajo potrebne kakovosti. Oba pojma sta integralni del upravljanja kakovosti (Sommerville, 2016).

V IT-sektorju nekatera podjetja vidijo zagotavljanje kakovosti programske opreme kot definiranje postopkov, procesov in standardov, ki bodo zagotovili potrebno kakovost. V drugih podjetjih zagotavljanje kakovosti vključuje tudi celotno upravljanje konfiguracije programske opreme ter aktivnosti preverjanja in potrjevanja, ki se izvajajo po izročitvi produkta razvojne ekipe. V večjih podjetjih in pri projektih, kjer se izvaja tradicionalni proces razvoja, je pomembna določena formalizacija upravljanja kakovosti. V ta namen obstajajo ekipe upravljanja kakovosti (angl. quality management team), ki skrbijo za neodvisen pregled procesa razvoja in rezultata.

Ta ekipa preverja projektno dokumentacijo in skladnost njenega izvajanja, skrbi za doseganje organizacijskih ciljev ter standardov in je odgovorna za testiranje programske opreme pred izdajo stranki in upravljanje izdaj. Ekipa upravljanja kakovosti mora biti neodvisna in ne sme biti del razvojne ekipe. Le tako lahko ima povsem objektiven pogled na kakovost programske opreme, ne da bi nanj vplivale težave, ki se pojavljajo pri razvoju. V idealnem primeru bi morala imeti tudi odgovornost za upravljanje kakovosti na nivoju celotne organizacije oziroma podjetja. Kar pomeni, da poroča direktno vodstvu podjetja in ne projektnemu vodstvu. Ker mora vodstvo projekta upoštevati časovne in finančne okvirje, se lahko zgodi, da žrtvuje določeno mero kakovosti za doseganje ciljev.

Naloga ekipe upravljanja kakovosti je torej, da prepreči vpliv kratkoročnih finančnih in časovnih pomislekov na končno kakovost programske opreme. Vendar pa je to v manjšem razvojem podjetju, kjer se večinoma prakticirajo agilni pristopi, praktično nemogoče, saj sta ekipa upravljanja kakovosti in razvojna ekipa neizogibno prepleteni. Kar pomeni, da ima v večini primerov določena oseba razvojne in kakovostne odgovornosti (Sommerville, 2016). V nadaljevanju so zato najprej opredeljene lastnosti kakovostne programske opreme ter tehnike zagotavljanja kakovosti, ki jih lahko izvaja manjše razvojno podjetje.

3.1 Kakovost programske opreme

V najbolj splošnem smislu se lahko kakovost programske opreme (angl. software quality) opredeli kot učinkovit proces, uporabljen na način, katerega rezultat je uporaben produkt, ki zagotavlja merljivo poslovno vrednost tistemu, ki ga razvija, ter tistemu, ki ga uporablja (Bessin, 2004). Iz navedene opredelitve lahko poudarim naslednje tri pomembne točke (Pressman, 2010):

- Učinkovit proces vzpostavlja infrastrukturo, ki podpira vsa prizadevanja za razvoj visokokakovostne programske opreme. Če gledamo z vodstvenega vidika, to pomeni ustvarjanje okolja, ki preprečuje nastanek kaosa pri projektu, kar je ključni prispevek k nižji kakovosti. Uporaba dobrih inženirskih praks omogoča razvijalcem, da lažje analizirajo težave in načrtujejo robustne rešitve, kar pa je kritičen prispevek za razvoj visokokakovostne programske opreme. Prav tako ostale aktivnosti in tehnike, kot je pregledovanje programske kode, ki prispevajo k splošni kakovosti.
- Uporaben produkt ponuja vsebino, funkcije in funkcionalnosti, ki zadovoljujejo želje končnih uporabnikov. Pomembneje pa je, da uporaben produkt to ponuja na zanesljiv način in brez napak. Dodatna lastnost visokokakovostne programske opreme so tudi vse implicitne funkcionalnosti, ki jih ponuja, kot je na primer enostavnost uporabe. Takšne funkcionalnosti niso eksplicitno izražene s strani končnih uporabnikov, temveč so samodejno pričakovane.
- Z dodajanjem poslovne vrednosti tistemu, ki razvija, in tistemu, ki uporablja produkt, pridobijo korist podjetje, ki ustvarja produkt, in organizacija, kjer so zaposleni končni uporabniki. Razvojno podjetje pridobi dodano poslovno vrednost, ker visokokakovostna programska oprema zahteva manj truda pri vzdrževanju, manj popravkov napak ter zmanjšuje količino podpore strankam. To podjetju omogoča, da lahko svoje vire preusmeri v razvoj novih produktov in porabi manj časa za obvladovanje obstoječih. Organizacija končnih uporabnikov pridobi dodano poslovno vrednost, ker produkt nudi uporabne sposobnosti, ki pospešujejo izvajanje njenih poslovnih procesov. Končni rezultat je zato večja donosnost ob uporabi ponujene informacijske podpore ter izboljšana razpoložljivost informacij, ki so ključnega pomena za poslovanje.

Kakovost programske opreme lahko opredelimo tudi preko kakovostnih faktorjev oziroma atributov programske opreme, ki vplivajo na nivo kakovosti.

V nadaljevanju je predstavljena najbolj znana kategorizacija kakovostnih faktorjev, ki velja še danes (McCall, Richards & Walters, 1977):

Operativne značilnosti:

- *Pravilnost* (angl. correctness). Mera izpopolnjevanja opredeljenih specifikacij in ciljev strank.
- *Zanesljivost* (angl. reliability). Mera pričakovanja, da bo program opravil predvideno funkcijo z zahtevano natančnostjo.
- *Učinkovitost* (angl. efficiency). Količina računalniških virov in programske kode, ki jih program potrebuje, da izvede svojo funkcijo.
- *Integriteta* (angl. integrity). Mera nadzora nepooblaščenih dostopov do funkcionalnosti ali podatkov.
- *Uporabnost* (angl. usability). Količina napora, potrebnega za razumevanje delovanja programa, pripravo vnosa podatkov in razlago izhoda programa.

Zmožnosti spreminjanja:

- *Obvladljivost* (angl. maintainability). Količina napora, potrebnega za iskanje in odpravo napake v programu.
- *Prilagodljivost* (angl. flexibility). Količina napora, potrebnega za spremembo operativne funkcionalnosti programa.
- *Zmožnost testiranja* (angl. testability). Količina napora, potrebnega za preizkušanje programa za pravilnost izvajanja svoje načrtovane funkcije.

Prilagodljivost novim okoljem:

- *Prenosljivost* (angl. portability). Količina napora, potrebnega za prenos programa v novo okolje ali zamenjavo strojne opreme.
- *Ponovna uporaba* (angl. reusability). Obseg funkcij programa, ki jih lahko ponovno uporabimo znotraj drugih funkcionalnosti.
- *Povezljivost* (angl. interoperability). Količina napora, potrebnega pri povezovanju programa z drugimi sistemi.

Programsko opremo ni mogoče optimizirati za vse naštete atribute. Na primer, če želimo izboljšati integriteto, lahko to zniža končno učinkovitost programa. Zato je potrebno definirati najpomembnejše kakovostne faktorje, ki jih želimo doseči ob razvoju. Neizogibno je, da bo za doseganje zastavljenih ciljev potrebno določene lastnosti žrtvovati (Sommerville, 2016).

Glede na tip merljivosti lahko naštete kakovostne faktorje razvrstimo v dve skupini. Tiste, ki jih lahko neposredno izmerimo, na primer število odkritih napak pri testiranju sistema, in tiste, ki jih lahko izmerimo samo posredno, na primer uporabnost in obvladljivost, pri katerih nastopa določena mera subjektivnosti. Takšne lastnosti nam dajejo tako imenovani mehek pogled na kakovost programske opreme (Pressman, 2010).

Ob že naštetih lastnostih izpostavljam še naslednja dva vidika kakovosti (Garvin, 1987):

- *Transcendentni vidik*. Ena izmed transcendentnih lastnosti je estetika programske opreme. Brez dvoma ima vsak posameznik svoj pogled na estetiko, vendar v večini primerov ob njej pomislimo na celosten izgled aplikacije ter edinstven tok uporabe, ki ga ponuja. Naslednja lastnost je percepcija programske opreme. V nekaterih situacijah ima posameznik določene predsodke, ki vplivajo na njegovo dojemanje kakovosti, na primer ob spoznavanju nove programske opreme, ki jo je razvilo podjetje, katerega produkt smo že uporabljali v preteklosti in ni dosegal določene kakovosti. V tem primeru ima posameznik negativen predsodek. V nasprotnem primeru, če se spoznavamo z novo programsko opreme proizvajalca, s katerim smo imeli dobre izkušnje v preteklosti, pa bomo nedvomno začeli s pozitivnim predsodkom (Pressman, 2010).

- *Ekonomski vidik*. Pogled na kakovost z vidika uravnavanja časa in stroškov na eni strani ter dobička na drugi strani. Lastnosti, ki jih lahko izrazimo v obliki denarja, so zmanjšanje stroškov zaradi učinkovitejšega izvajanja procesa ter izboljšanje kazalnikov uspešnosti zaradi boljše razpoložljivosti informacij za poslovno odločanje. Ostale lastnosti so strateške narave, kot je povečanje tržnega deleža ali tržnosti produkta podjetja (van Vliet, 2007).

Ker je razvoj programske opreme kreativen proces, imajo bistven vpliv na kakovost tudi znanja in izkušnje posameznikov, ki jo razvijajo. Prav tako imajo vpliv zunanji dejavniki, kot na primer pritiski za zgodnjo izdajo produkta. Kvaliteta procesa razvoja ima nedvomno vpliv na končno kvaliteto produkta. Rezultat dobrega procesa je dobra kvaliteta programske opreme. Upravljanje kakovosti programske opreme se zato ukvarja tudi z izboljšavo procesa razvoja.

Ker je težko oceniti določene kakovostne faktorje, preden je produkt dlje časa v uporabi, je tudi težko oceniti, kako karakteristike procesa vplivajo na te lastnosti. Prav tako je pomembo poudariti, da prevelika standardizacija in rigoroznost procesa včasih zavirata ustvarjalnost, kar lahko vodi k slabši kakovosti programske opreme. Jasno definiran proces je pomemben, vendar mora biti cilj vodstva tudi ustvariti kulturo kakovosti, znotraj katere so vsi zavezani k doseganju čim višje kakovosti svojih produktov.

Prav tako je pomembno spodbujanje odgovornosti posameznikov za kakovost svojega dela ter iskanje novih pristopov za izboljšavo kakovosti. Standardi in postopki v procesu so osnova za dobro upravljanje kakovosti programske opreme, vendar pa vodilni managerji na tem področju poudarjajo, da so določene lastnosti visokokakovostne programske opreme, kot sta na primer eleganca in berljivost programske kode, ki jih ne moremo zajeti znotraj definiranih postopkov. Takšne lastnosti posameznikov je potrebno podpirati in spodbujati tudi ostale člane razvojne ekipe, da skušajo doseči enak nivo kakovosti (Sommerville, 2016).

3.2 Zagotavljanje kakovosti programske opreme

Zagotavljanje kakovosti programske opreme (angl. software quality assurance) zajema širok seznam aktivnosti znotraj področja upravljanja kakovosti. Povzete so lahko v naslednje sklope (Horch, 2003):

- *Pregledi in revizije*. Tehnični pregledi so aktivnost nadzora kakovosti, ki jo izvajajo razvijalci za druge razvijalce. Njihov primarni namen je iskanje napak. Revizije so vrsta pregleda, ki ga izvaja ekipa upravljanja kakovosti, da zagotovi sledenje zastavljenim smernicam.
- *Testiranje*. Preizkušanje programske opreme je ključna aktivnost nadzora kakovosti, katere glavni cilj je odkrivanje napak.

- *Standardi*. Organizacije za standardizacijo so izdelale širok seznam standardov in z njimi povezanih dokumentov na področju upravljanja kakovosti. Standarde v večini primerov uvede vodstvo ali stranka, lahko pa jih prostovoljno sprejmejo tudi razvijalci sami.
- *Zbiranje in analiza napak*. Za boljše razumevanje, kako nastanejo napake v programski opremi in kateri postopki zagotavljanja kakovosti so najbolj primerni za posamezno podjetje, je potrebno zbirati in analizirati podatke, ki nastajajo ob izvajanju procesa.
- *Upravljanje sprememb*. Spremembe programske opreme zahtevajo skrben pristop, saj lahko hitro pride do kaosa v programski kodi in to večinoma privede do slabše kakovosti. Zagotoviti je potrebno primerne prakse, ki nadzirajo spremembe programske opreme.
- *Izobraževanje*. Vsako razvojno podjetje želi izboljšati svoje razvojne prakse. Ključen prispevek k izboljšanju je izobraževanje razvijalcev, vodstva in ostalih deležnikov.
- *Upravljanje varnosti*. S porastom vprašanj glede zasebnosti podatkov bi moralo vsako razvojno podjetje uvesti politike, ki ščitijo občutljive podatke na vseh nivojih.

V nadaljevanju sta opredeljeni dve tehniki zagotavljanja kakovosti programske opreme, ki se najpogosteje izvajata v manjših razvojnih podjetjih oziroma v razvojnih podjetjih, kjer ni namenske ekipe za upravljanje kakovosti, temveč so razvijalci sami odgovorni za končno kakovost produkta.

3.2.1 Pregledovanje programske kode

Pregled programske kode (angl. code review) je vrsta tehniškega pregleda (angl. tudi peer review), kjer razvijalci pregledujejo izvorno kodo, ki so jo napisali drugi razvijalci. Primarni namen je odkrivanje napak pred uporabo produkta v končnem okolju, splošna izboljšava kakovosti programske opreme, izobraževanje razvijalcev ter predaja znanja. Pregled se lahko izvede v dveh oblikah (Pressman, 2010):

- *Neformalno*. Na primer preprost pogovor med dvema sodelavcema v isti pisarni glede uporabljene metode dela in strukture izvorne kode ali spontan sestanek, kjer se rešujejo težave. Neformalna oblika pregledovanja programske kode se tudi izvaja pri programiranju v paru.
- *Formalno*. Napovedan sestanek, na katerem je udeleženih več ljudi. Običajno je rezultat sestanka dokument, v katerem so zapisane vse odkrite napake in težave v programski kodi ter izvajalci, ki so zadolženi za odpravo posamezne slabosti.

Že pretekle raziskave poročajo, da je pregledovanje programske kode (angl. tudi software inspection) zelo učinkovita tehnika zagotavljanja kakovosti programske opreme. Z njo lahko odkrijemo tudi več kot 60 odstotkov napak v programski opremi (Fagan, 1986). Tudi kasnejše raziskave, ki so primerjale pregledovanje z ostalimi tehnikami, so ugotovile enak odstotek učinkovitosti (McConnell, 2004).

Prav tako tudi novejše raziskave potrjujejo, da ima tehnika pregledovanja programske kode visok pozitiven učinek na kakovost programske opreme. Po analizi literature na tem področju v nadaljevanju povzemam ugotovitve ter prednosti in slabosti:

- Nepregledana programska koda ima več kot dvakrat večjo možnost, da bo bodo potrebni popravki napak kot pregledana programska koda. Po pregledu se bistveno poveča berljivost kode. Večje kot je število pregledovalcev, manjša je možnost za nastanek napak (Bavota & Russo, 2015).
- Eden izmed rezultatov pregledovanja programske kode je sodelovanje med razvijalci v obliki pripomb, s katerimi posamezniki izražajo svoje mnenje glede napisane kode. Čeprav so pripombe namenjene diskusiji o morebitnih težavah, lahko vsebujejo tudi škodljive konotacije, ki pa ogrožajo korist tehnike. Raziskava ugotavlja, da je način izražanja povezan s statusom razvijalca znotraj ekipe. Izražanje novih članov je običajno nevtralno. Pripombe, ki vsebujejo negativen pridih, podaljšujejo čas, ki je potreben za zaključek procesa. Dodatne sporne pripombe pa ta čas še bistveno podaljšajo (El Asri, Kerzazi, Uddin, Khomh & Idrissi, 2019).
- Glavni dejavniki, ki negativno vplivajo na učinkovitost pregledovanja programske kode in se ne pojavljajo pri ostalih tehnikah zagotavljanja kakovosti programske opreme, so negativna čustva, neenakost oziroma pristranskost med razvijalci ter šibek oziroma nezadosten odziv avtorja pregledane kode (Fatima, Nazir & Chuprat, 2020).

Neformalni pregled programske kode znotraj razvojne ekipe, kjer si posamezniki delijo delo znotraj iste domene izvorne kode, imenujemo tudi družabno programiranje (angl. social coding). Raziskava, ki proučuje družabno programiranje in prenos kolektivnega spomina (angl. transactive memory), je ugotovila, da ima družabno programiranje pozitivne učinke tudi v manjših razvojnih ekipah, ki delujejo na isti lokaciji oziroma v isti pisarni in razvijajo programsko opremo za stranke. Raziskovalci so ugotovili, da izvajanje te tehnike v omenjenem kontekstu ni potrata časa, temveč dejanski gonilnik kakovosti programske opreme, ki prinaša poslovno vrednost stranki in razvojnemu podjetju. Razlog za takšne ugotovitve ni samo dejstvo, da ob izvajanju tehnike več ljudi pregleda določen odsek programske kode, temveč je razlog tudi izmenjava znanja, ki se zgodi zaradi različnih strokovnih znanj posameznikov. Vse to prispeva k prenosu in gradnji kolektivnega znanja. Raziskava nudi naslednje implikacije, ki jih lahko vodstvo manjših razvojnih ekip uporabi v praksi (Spohrer, Kude, Schmidt & Heinzl, 2016):

- Uvedba družabnega programiranja lahko dejansko bistveno izboljša kakovost programske opreme.
- Družabno programiranje pomaga managerjem uravnoteženo razdeliti delo med svoje zaposlene. Če izkušenejši razvijalci nimajo časa za razvoj določene težje funkcionalnosti, lahko sodelujejo samo kot pregledovalci, delo pa naj prevzamejo manj izkušeni razvijalci, ki bodo skozi izvajanje pregleda pridobili strokovni odziv na njihovo rešitev. To implikacijo potrjujejo tudi druge raziskave (McIntosh, Kamei, Adams & Hassan, 2016).

3.2.2 Vzpostavitev standardov programiranja

Koncept vzpostavitve standardov programiranja (angl. coding standard) znotraj razvojne ekipe je del prakse XP in tudi ena izmed tehnik zagotavljanja kakovosti programske opreme. V določeni literaturi se koncept imenuje tudi pravila oziroma predpisi programiranja (angl. coding convention). Je pomemben del upravljanja kakovosti programske opreme, saj (Sommerville, 2016):

- Zajema skupno modrost, ki jo razvojna ekipa ima in prinaša vrednost podjetju. Temelji na znanju o najboljših in najprimernejših praksah za določeno podjetje. To znanje izvira iz preteklih izkušenj reševanja težav in napak. Vpeljava standardov pomaga podjetju, da se v prihodnje izogne znanim problemom.
- Standardi postavijo okvir, ki določi, kaj konkretno pomeni kakovost v določeni razvojni ekipi. Ker je kakovost programske opreme tudi subjektivne narave, se z uporabo standardov vzpostavi podlaga za odločitev, ali je bil dosežen zahtevan nivo kakovosti.
- Standardi pomagajo ohranjati konstanten tok dela, kadar določeno opravilo prevzame druga oseba. Zagotavljajo, da vsi razvijalci v ekipi izvajajo enake prakse. Posledično se zmanjša začetni napor pri zagonu novega projekta.

Dobro definirani standardi programiranja vodijo k razumljivi in samodokumentirani izvorni kodi, kar povečuje obvladljivost programske opreme in prispeva h končni kakovosti produkta (Pressman, 2010). To potrjujejo tudi pretekle raziskave (Fang, 2001). Po analizi novejše literature na tem področju v nadaljevanju povzemam ugotovitve ter prednosti in slabosti:

- Visoko berljiva izvorna koda je ključnega pomena za razumevanje konteksta ter namena njenega avtorja. Omogoča, da razvijalci razumejo drug drugega in lažje komunicirajo med sabo, kar posledično niža stroške vzdrževanja. Raziskava empirično dokazuje, da nedosledno upoštevanje standardov niža berljivost programske kode. Najmočnejši dejavniki za slabo berljivost so nekonsistentni zamiki ter odvečni komentarji, ki niso v skladu z napisano kodo. V profesionalnem okolju, kjer se razvija programska oprema za stranke, se razvijalci mnogokrat ne držijo pravil, saj imajo na voljo omejen čas zaradi vnaprej opredeljenih časovnih okvirjev. Zato je naloga vodstva, da uvidi poslovno vrednost v sledenju vzpostavljenim standardom in sprejme pravo odločitev, kadar se je potrebno odločiti med visoko berljivo izvorno kodo in hitrejšo izdajo produkta. Omenjena raziskava proučuje tudi korelacije med doslednostjo upoštevanja standardov programiranja ter izkušnjami razvijalcev in velikostjo razvojne ekipe. Raziskovalci so ugotovili, da so izkušenejši razvijalci bolj dosledni pri upoštevanju standardov ter da z večanjem razvojne ekipe narašča število neupoštevanj standardov (Lee, Lee & In, 2015).
- Če razvijalec dosledno sledi vzpostavljenim standardom, lahko pride do izraza njegov stil programiranja (angl. coding style). Ker je stil povsem subjektivne narave, je nemogoče oceniti, kateri je najboljši. Takšno ocenjevanje niti ni smiselno, saj stil ne sme vplivati na nivo skladnosti izvorne kode z vzpostavljenimi standardi. Raziskovalci, ki so

preučevali manjše razvojne ekipe in vpliv standardov programiranja na število končnih napak, so ugotovili, da nov član ekipe proizvede več napak, saj je doslednost upoštevanja standardov manjša kot pri obstoječih razvijalcih. Vendar pa z večanjem doslednosti skozi čas število napak upada, kar je v skladu s predhodno analizirano raziskavo. Raziskovalci predlagajo, da se v manjših razvojnih ekipah daje prednost standardom in ne stilu programiranja. (Popic, Velikic, Jaroslav, Spasic & Vulic, 2018).

- Raziskava, ki proučuje nastanek zmede (in posledično zamude) pri pregledovanju programske kode, je ugotovila, da so glavni razlogi za zmedo manjkajoče utemeljitve, pomanjkanje poznavanja kode ter razprave o načinu izvedbe rešitve. Za obvladovanje zmede razvijalci največkrat zahtevajo dodatna pojasnila avtorja. Vpeljava standardov programiranja zmanjša potrebo po razlagi, kar pa zmanjša čas, potreben za izvedbo pregleda. Posledično se zmanjša čas izvajanja celotnega procesa razvoja programske opreme (Ebert, Castor, Novielli & Serebrenik, 2019).
- Ker se z vzpostavitvijo standardov omeji kreativna svoboda pri pisanju programske kode, je ob časovni omejitvi dodaten razlog za nedosledno upoštevanje standardov programiranja tudi volja posameznega razvijalca, da bi bilo njegovo delo skladno z ekipnim delom. Raziskava je pokazala, da je gamifikacija (angl. gamification) lahko učinkovit pristop, s katerim vodstvo motivira svoje razvijalce, da bolj spoštujejo vzpostavljene standarde, ne da bi ob tem negativno vplivali na medsebojne odnose. Gamifikacija omogoča, da ljudje trenutno delo obravnavajo bolj lahkotno in da se ustvari določeno okolje, znotraj katerega so vsi razvijalci med sabo enaki, ne glede na starost ali izkušnje (Jarke & Prause, 2015).

4 ŠTUDIJA PRIMERA ZAGOTAVLJANJA KAKOVOSTI IN IZBOLJŠAVE PROCESA RAZVOJA IN VZDRŽEVANJA PROGRAMSKE OPREME NA PRIMERU IZBRANEGA PODJETJA

Izbrano podjetje je manjše slovensko razvojno podjetje. Povprečno število zaposlenih v zadnjih treh letih je bilo 32. Povprečni letni čisti prihodki od prodaje v zadnjih treh letih so bili 1,5 milijona evrov. Torej gre za majhno družbo na področju računalništva in informatike, ki se ukvarja z informacijskim inženiringom in svetovanjem. Temeljni poslovni proces podjetja je razvoj in vzdrževanje programske opreme.

Podjetje razvija lastno programsko opremo. Glavna dva produkta, ki prinašata večino poslovne vrednosti podjetju, sta:

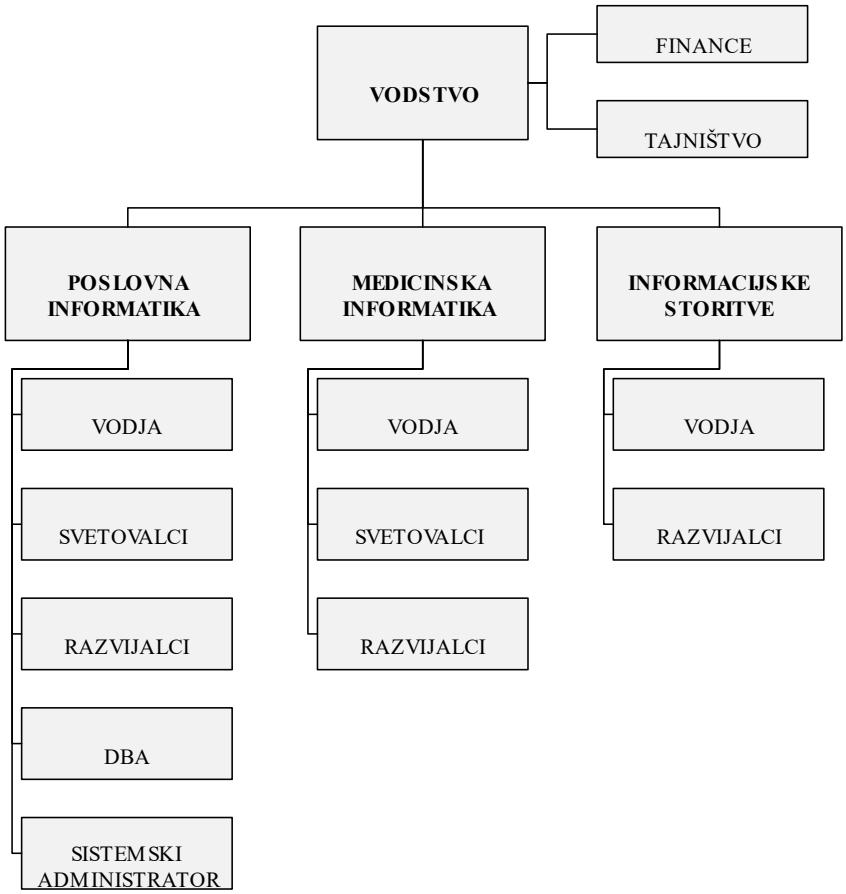
- ERP, ki nudi podporo osnovnemu poslovanju in specifičnim poslovnim procesom v različnih panogah.
- Medicinski informacijski sistem, ki nudi podporo poslovnim procesom na primarnem nivoju zdravstvenega varstva.

Ob glavnih dveh produktih podjetje razvija tudi informacijske sisteme za podporo poslovanja visokošolskih zavodov in drugih javnih institucij. Podjetje prav tako nudi razvoj programske opreme po naročilu. V preteklosti je izvajalo naročniške projekte v javnem in zasebnem sektorju.

Slika 8 prikazuje organigram podjetja, iz katerega je razvidno, da je podjetje sestavljeno iz treh enot oziroma oddelkov. Prvi oddelek (Poslovna informatika) deluje na področju poslovne informatike in se ukvarja z razvojem in vzdrževanjem ERP-produkta. Drugi oddelek (Medicinska informatika) deluje na področju medicinske informatike in se ukvarja z razvojem in vzdrževanjem produkta na področju medicinske informatike. Tretji oddelek (Informacijske storitve) pa deluje na več različnih področjih in se ukvarja z razvojem ter vzdrževanjem ostalih produktov podjetja.

Ti trije oddelki izvajajo temeljni poslovni proces razvoja in vzdrževanja programske opreme, ki ustvarja poslovno vrednost podjetja. Pri vzdrževanju programske opreme se izvaja tudi podproces podpore in svetovanja strankam. Ob temeljnem poslovnem procesu se v podjetju izvajajo vodstveni proces (Vodstvo) in ostali podporni procesi (Finance in Tajništvo).

Slika 8: Organigram izbranega podjetja



Vir: lastno delo.

Tabela 1 prikazuje opis posameznega delovnega mesta in opredelitev njegovih zadolžitev oziroma aktivnosti, ki jih izvaja v procesu.

Tabela 1: Opis in zadolžitve delovnih mest v oddelkih

Delovno mesto	**Opis in zadolžitve**
Vodja oddelka	Vodi celoten oddelek (je lastnik procesa). Je odgovoren za izvajanje procesa ter doseganje ciljev procesa in poslovne uspešnosti. Znotraj temeljnega procesa praviloma izvaja le podporo strankam in svetovanje.
Svetovalec	Je odgovoren za delovanje posameznega vsebinskega področja (modula) produkta. Praviloma izvaja le podporo strankam in svetovanje.
Razvijalec	Izvaja temeljni poslovni proces razvoja in vzdrževanja programske opreme.
DBA	Administrator podatkovnih zbirk (angl. Database Administrator, v nadaljevanju DBA) skrbi za pripravo nadgradenj programske opreme. Pravilom izvaja le vzdrževanje programske opreme.
Sistemski administrator	Skrbi za delovanje računalniške infrastrukture. Znotraj temeljnega procesa praviloma izvaja le podporo strankam.

Vir: lastno delo.

4.1 Cilji ter tehnike in metode dela

Na začetku poglavja je predstavljeno izbrano podjetje, kjer je opredeljena njegova velikost in organizacijska struktura. Navedeni so produkti podjetja in področje delovanja ter opisana delovna mesta.

V nadaljevanju poglavja je na primeru izbranega podjetja izvedena študija primera zagotavljanja kakovosti in izboljšave procesa razvoja in vzdrževanja programske opreme. Potrebni podatki za izvedbo študije primera temeljijo na informacijah, ki sem jih pridobil z izvedbo neformalnih pogovorov z udeleženci analiziranih poslovnih procesov in informacijah, ki sem jih pridobil z metodo opazovanja izvedbe analiziranih poslovnih procesov. Študija primera je razdeljena na tri podpoglavja, znotraj katerih z različnimi metodami in tehnikami poskušam doseči naslednje zastavljene cilje.

Razvoj programske opreme:

- Opredelitev procesa razvoja z opisom dogodkov, aktivnosti in organizacijskih enot v procesu.
- Modeliranje procesa razvoja z grafično notacijo za modeliranje poslovnih procesov in delovnih tokov (angl. Business Process Modelling Notation, v nadaljevanju BPMN).

- Analiza procesa razvoja s kvalitativno metodo dodajanja vrednosti.
- Identifikacija pomanjkljivosti in izdelava predlogov za izboljšave v kontekstu manjšega razvojnega podjetja.

Vzdrževanje programske opreme:

- Opredelitev procesa vzdrževanja z opisom dogodkov, aktivnosti in organizacijskih enot v procesu.
- Modeliranje procesa vzdrževanja z BPMN-tehniko modeliranja.
- Analiza procesa vzdrževanja s kvalitativno metodo dodajanja vrednosti.
- Identifikacija pomanjkljivosti in izdelava predlogov za izboljšave v kontekstu manjšega razvojnega podjetja.

Zagotavljanje kakovosti programske opreme:

- Analiza sedanjega stanja zagotavljanja kakovosti.
- Identifikacija pomanjkljivosti in izdelava predlogov za izboljšave zagotavljanja kakovosti v kontekstu manjšega razvojnega podjetja.

4.2 Razvoj programske opreme

Čeprav literatura poudarja, da je razvoj in vzdrževanje programske opreme bolj smiselno obravnavati kot neprekinjen proces (Sommerville, 2016), za namene te študije primera obravnavam temeljni poslovni proces izbranega podjetja kot dva ločena procesa. Tako je lažje izvedena poglobljena analiza temeljnega procesa.

4.2.1 Opredelitev procesa

Ker temeljni poslovni proces izbranega podjetja obravnavam kot dva ločena procesa, je najprej potrebno pojasniti mejo med razvojem in vzdrževanjem programske opreme. Zgodovinsko gledano je ta meja vedno obstajala, danes pa je to mejo zaradi agilnih pristopov vse težje opredeliti (Sommerville, 2016). Enako velja za izbrano podjetje, ki že dlje časa uporablja samo agilen pristop k razvoju. Razlog je njegova velikost ter vrsta programske opreme, ki jo razvija. Ker gre za manjše podjetje, so človeški viri omejeni, kar zmanjšuje zmožnost pisanja obsežne dokumentacije. Programska oprema, ki jo razvija, ni varnostno ali življenjsko kritična, prav tako se specifikacije redno spreminjajo. Oboje zmanjšuje potrebo po pisanju obsežne dokumentacije.

Proces razvoja programske opreme v izbranem podjetju je torej agilen, kar pomeni, da je težje določiti, kdaj se konča razvoj in začne vzdrževanje programske opreme. Mejo lahko postavim glede na:

- *Poslovni vidik.* Prvi možni mejnik med razvojem in vzdrževanjem programske opreme je lahko dan podpisa vzdrževalne pogodbe oziroma datum pričetka njene veljavnosti.
- *Način uporabe programske opreme.* Drugi možni mejnik med razvojem in vzdrževanjem programske opreme je lahko dan prehoda v produkcijo oziroma pričetek uporabe v končnem produkcijskem okolju.
- *Vzroke za spremembo programske opreme.* Tretji možni mejnik med razvojem in vzdrževanjem programske opreme je lahko, kadar v določeni iteraciji agilnega procesa količina popravljanja programske opreme preseže količino dograditev programske opreme.

V izbranem podjetju je meja med razvojem in vzdrževanjem najpogosteje kombinacija prvega in drugega mejnika. Ko stranka vzpostavi programsko opremo v končnem produkcijskem okolju in prične s produkcijsko uporabo, se praviloma podpiše tudi vzdrževalna pogodba. Ni pa pri vseh projektih enaka meja. Vzdrževanje programske opreme je v izbranem podjetju večji del operativnega dela, kamor tudi sodi podpora in svetovanje strankam. Razvoj nove programske opreme pa se razume kot projekt, katerega trajanje je odvisno od njegove narave. Kadar podjetje razvija lastno programsko opremo, se po koncu razvoja njeno vzdrževanje vključi v operativno delo podjetja. Način izvedbe procesa razvoja programske opreme se v izbranem podjetju razlikuje med posameznimi projekti. Najbolj je odvisen od velikosti projekta in mere vključenosti naročnika oziroma stranke. V nadaljevanju bom proces razvoja programske opreme opredelil na primeru dveh projektov. Prvi (i) večji in daljši naročniški projekt, pri katerem je bil naročnik močno vključen v izvedbo procesa ter drugi (ii), manjši in krajši interni razvojni projekt.

Tabela 2 opisuje dogodke, ki se v izbranem podjetju zgodijo v procesu razvoja programske opreme.

Tabela 2: Dogodki v procesu razvoja programske opreme

Dogodek	**Vrsta**	**Opis**
Zahteve za informacijsko podporo	Začetni	Posamezna iteracija procesa razvoja programske opreme se začne z novimi zahtevami za informacijsko podporo. V naročniškem (i) primeru izvedbe procesa dogodek najpogosteje sproži oseba na strani naročnika, ki je odgovorna za vsebino oziroma je lastnik poslovnega procesa, ki potrebuje informacijsko podporo. V internem (ii) primeru izvedbe procesa pa dogodek najpogosteje sproži svetovalec, ki vidi potrebo po prenovi obstoječe ali poslovno vrednost v razvoju nove programske opreme.

se nadaljuje

Tabela 2: Dogodki v procesu razvoja programske opreme (nad.)

Dogodek	**Vrsta**	**Opis**
Dokumentirana vsebina zahtev	Vmesni	Dogodek, ki ga lahko razumemo kot prehod med fazo analize in fazo načrtovanja programske opreme. V naročniškem (i) primeru izvedbe procesa ga sproži prejeta specifikacija razvojne naloge, ki jo naročnik pošlje izbranemu podjetju. Specifikacija vsebuje opredeljene in analizirane zahteve za posamezno iteracijo. V internem (i) primeru izvedbe procesa se ta dogodek ne sproži, saj je sta fazi analize in načrtovanja programske opreme močno prepleteni. Proces se izvaja samo znotraj izbranega podjetja. Specifikacije zahtev se pogosto spreminjajo. Prav tako se med analizo izvaja načrtovanje in obratno.
Dokumentirana informacijska podpora	Vmesni	Ko izvajalec (izbrano podjetje) zaključi z načrtovanjem informacijske podpore, pošlje dopolnjeno specifikacijo razvojne naloge naročniku v pregled. Ta dogodek se sproži samo v naročniškem (i) primeru izvedbe procesa.
Potrjena specifikacija razvojne naloge	Vmesni	Dogodek, ki ga lahko razumemo kot prehod med fazo načrtovanja in fazo implementacije programske opreme. Sproži ga potrditev razvojne naloge s strani naročnika. Pogosto ta dogodek spremljajo tudi poslovni dogodki v odnosu med naročnikom in izvajalcem. Ta dogodek se sproži samo v naročniškem (i) primeru izvedbe procesa.
Zahteve, pripravljene za testiranje	Vmesni	Dogodek, ki ga lahko razumemo kot prehod med fazo implementacije in fazo testiranja programske opreme. Sproži ga podpisan zapisnik testiranj s strani izvajalca (izbrano podjetje) in dogovor o začetku testiranja s strani naročnika. Tudi ta dogodek se sproži samo v naročniškem (i) primeru izvedbe procesa, saj je v internem (ii) primeru izvedbe faza testiranja močno prepletena z implementacijo programske opreme.
Zahteve, informacijsko podprte	Končni	Posamezna iteracija procesa razvoja programske opreme se konča, ko so zahteve iz začetka iteracije informacijsko podprte. V naročniškem (i) primeru izvedbe procesa dogodek najpogosteje sproži oseba, odgovorna za vsebino na strani naročnika, kadar zaključi fazo testiranja in hkrati podpiše zapisnik testiranja. V internem (ii) primeru izvedbe procesa pa dogodek najpogosteje sproži svetovalec, ko ugotovi, da spremembe programske opreme izpolnjujejo opredeljene zahteve oziroma zadovoljujejo zastavljen poslovni cilj.

Vir: lastno delo.

Tabela 3 006Fpisuje organizacijske enote, ki v izbranem podjetju izvajajo proces razvoja programske opreme.

Tabela 3: Vloge v procesu razvoja programske opreme

Vloga	**Opis**
Svetovalec	Oseba na delovnem mestu svetovalec.
Razvijalec	Oseba na delovnem mestu razvijalec.
Tester	Oseba na delovnem mestu razvijalec, ki je hkrati zadolžena za testiranje programske opreme.
(Naročnik) Odgovorna oseba za vsebino	Lastnik procesa, katerega informacijsko podpora se razvija v posamezni iteraciji procesa.
(Naročnik) Odgovorna oseba za izvedbo	Običajno projektni vodja.

Vir: lastno delo.

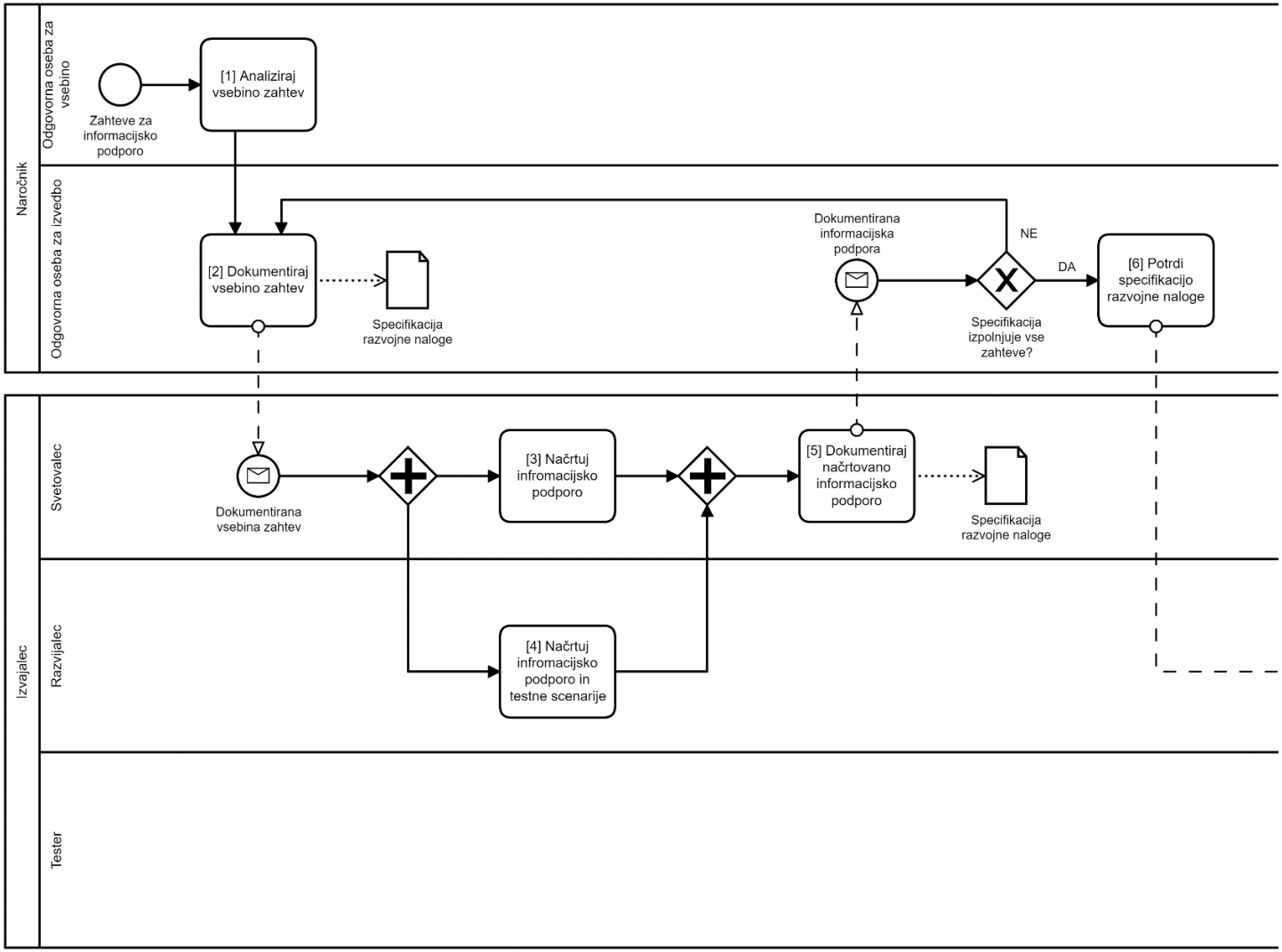
4.2.2 Modeliranje procesa

V nadaljevanju je za oba primera predstavljena najpogostejša oblika izvedbe procesa razvoja programske opreme. Za naročniški (i) primer izvedbe procesa je predstavljena ena iteracija agilnega pristopa. Za eno iteracijo se v izbranem podjetju uporablja tudi izraz razvojna naloga.

Iz modeliranja procesa sem izpustil aktivnosti, ki se zgodijo pred začetkom prve oziroma po koncu zadnje iteracije. Kot na primer začetni sestanek (angl. kickoff) in prehod v produkcijo. Prav tako sem izpustil poslovni vidik, kot so na primer ocena dela, postavitev roka izvedbe, vodenje projekta in podpis pogodbe.

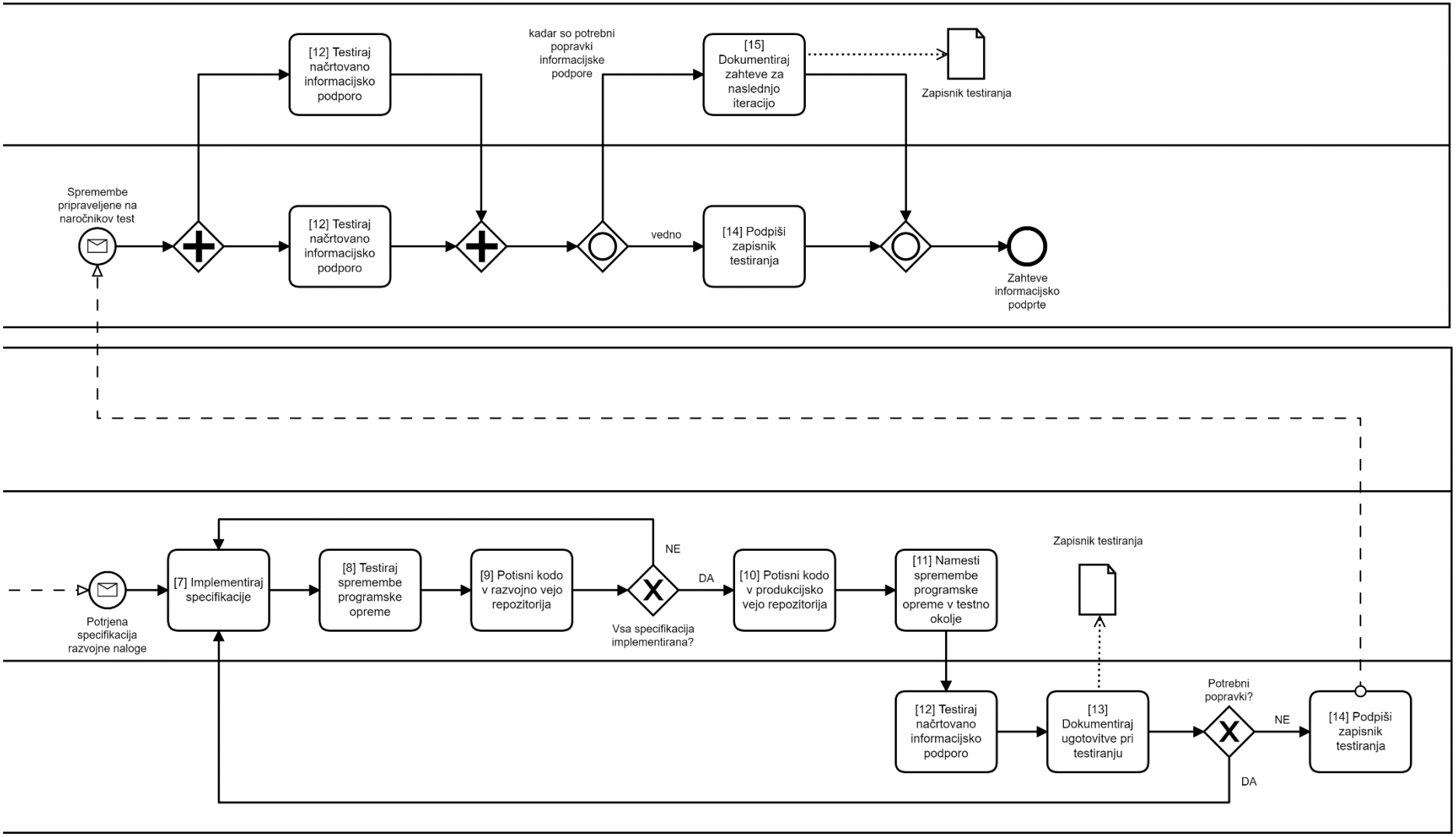
Na slikah 9 in 10 je prikazana BPMN-vizualizacija za naročniški (ii) primer izvedbe procesa razvoja programske opreme. Na sliki 11 je prikazana BPMN-vizualizacija za interni (ii) primer izvedbe procesa razvoja programske opreme.

Slika 9: BPMN-vizualizacija procesa razvoja za naročniški (i) primer izvedbe



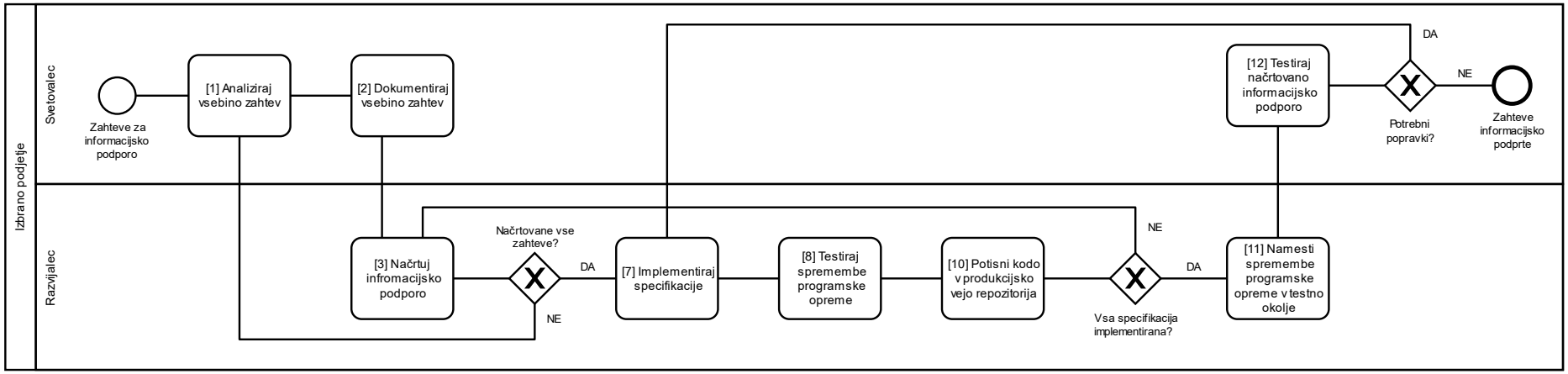
Vir: lastno delo.

Slika 10: BPMN-vizualizacija procesa razvoja za naročniški (i) primer izvedbe (nad.)



Vir: lastno delo.

Slika 11: BPMN-vizualizacija procesa razvoja za interni (ii) primer izvedbe



Vir: lastno delo.

V nadaljevanju so opisane lastnosti, izvajalci, cilji in rezultati posameznih aktivnosti, ki se izvajajo v procesu razvoja programske opreme.

[1] Analiziraj vsebino zahtev. Začetna aktivnosti v procesu razvoja programske opreme, ki jo sproži zahteva za informacijsko podporo. V naročniškem (i) primeru izvedbe procesa jo izvaja odgovorna oseba za vsebino na strani naročnika. Pogosto ji tudi pomaga svetovalec izbranega podjetja v obliki svetovanja. V internem (ii) primeru izvedbe procesa pa aktivnost izvaja svetovalec. V aktivnosti se izvaja analiza zahtev za informacijsko podporo. Izvajalci aktivnosti analizirajo in opredeljujejo lastnosti zahtev. Razmislijo, kaj je namen zahteve in kakšni so cilji, ki se jih želi doseči z informacijsko podporo. V aktivnosti se ne uporabljajo posebne tehnike ali metode razvoja programske opreme. Večinoma gre za sestanke, na katerih se diskutira o vsebini in hkrati izriše miselni vzorec ali skico za boljše razumevanje zahtev. Izvedba aktivnosti nima informacijske podpore.

[2] Dokumentiraj vsebino zahtev. Popis in dokumentacija izhoda prejšnje aktivnosti. Lahko se izvaja sočasno s prvo [1] aktivnostjo. V naročniškem (i) primeru izvedbe procesa jo izvaja odgovorna oseba za izvedbo na strani naročnika. V internem (ii) primeru izvedbe pa aktivnost izvaja svetovalec. Obseg dokumentacije so razlikuje med projekti. Pravilom pa je v naročniškem (i) primeru izvedbe procesa dokumentacija obsežnejša. Podpora izvedbi aktivnosti so tekstovni urejevalniki besedila, saj je večina dokumentacije v tekstovni obliki. Ob podrobni opredelitvi zahtev lahko dokumentacija vsebuje tudi opis obstoječega stanja informacijske podpore in predpogoje za izvedbo. V naročniškem (i) primeru izvedbe procesa je izhod aktivnosti prva verzija specifikacije razvojne naloge, ki se posreduje izvajalcu (izbranemu podjetje) preko e-pošte. V internem (ii) primeru izvedbe pa je izhod aktivnosti seznam opredeljenih zahtev, posredovan izvajalcu preko internih orodij za komunikacijo (več o internih orodjih v naslednjem podpoglavju).

[3] Načrtuj informacijsko podporo. Na podlagi dokumentirane vsebine zahtev se nato načrtuje informacijska podpora. V obeh primerih izvedbe procesa to aktivnost izvaja razvijalec. Kadar je načrtovana informacijska podpora vsebinsko zahtevna, jo izvaja tudi svetovalec. V aktivnosti se izvaja načrtovanje arhitekture programske opreme, modeliranje podatkovnih zbirk, definicija strukture podatkov, opredelitev funkcionalnosti in risanje žičnih diagramov. Obseg načrtovanja je odvisen od velikosti in zahtevnosti informacijske podpore. V naročniškem (i) primeru izvedbe procesa je praviloma več načrtovanja, saj so vse načrtovane funkcionalnosti informacijske podpore tudi dokumentirane. V aktivnosti se uporabljajo orodja za risanje diagramov relacij (angl. Entity-Relationship, v nadaljevanju ER) v logičnem podatkovnem modelu, kar je tudi ključni izhod te aktivnosti.

[4] Načrtuj informacijsko podporo in testne scenarije. V naročniškem (ii) primeru izvedbe procesa se ob načrtovanju istočasno pripravijo tudi testni scenariji za ročno testiranje informacijske podpore. To aktivnost izvaja razvijalec. Testni scenariji se napišejo na podlagi vsebine zahtev in načrtovanih funkcionalnosti. Za izvedbo aktivnosti se uporabljajo tekstovni urejevalniki besedila.

[5] Dokumentiraj načrtovano informacijsko podporo. Vhod v aktivnost sta izhoda aktivnosti [3] in [4], na podlagi katerih se dopolni specifikacija razvojne naloge. Ta aktivnost se izvaja v naročniškem (ii) primeru izvedbe procesa. Izvaja jo svetovalec. Za izvedbo aktivnosti se uporabljajo tekstovni urejevalniki besedila. Izhod aktivnosti je dokončana specifikacija razvojne naloge, ki se jo pošlje naročniku v pregled.

[6] Potrdi specifikacijo razvojne naloge. Ko naročnik prejme dopolnjeno specifikacijo razvojne naloge z načrtovano informacijsko podporo, odgovorna oseba za izvedbo pregleda dokument in oceni, ali specifikacija izpolnjuje vse zahteve. Nato specifikacijo potrdi in jo pošlje izvajalcu (izbrano podjetje). V nasprotnem primeru se izvedba aktivnosti [2], [3], [4] in [5] ponovi. Ta aktivnost se izvaja samo v naročniškem (ii) primeru izvedbe procesa. Potrditev specifikacije se izvede s podpisom dokumenta.

[7] Implementiraj specifikacije. Ključna aktivnost, ki dodaja vrednost v procesu razvoja programske opreme. Izvaja jo razvijalec. V naročniškem (ii) primeru izvedbe procesa je vhod v aktivnost potrjena specifikacija razvojne naloge, v internem (ii) primeru izvedbe pa seznam zahtev z opredeljeno vsebino in delno dokumentirano načrtovano informacijsko podporo. Razvijalec spreminja izvorno kodo programske opreme in dopolnjuje fizični model podatkovne zbirke. Za izvedbo aktivnosti uporablja različna orodja v integriranih razvojnih okoljih. Uporaba posameznih orodij je odvisna od programskega jezika, v katerem je napisana programska oprema. Rezultat aktivnosti so spremembe programske opreme, ki se odražajo v novih ali dopolnjenih funkcionalnostih načrtovane informacijske podpore.

[8] Testiraj spremembe programske opreme. Vsaka sprememba programske opreme v razvojnem okolju se testira. To aktivnost se najpogosteje izvaja istočasno z aktivnostjo [7]. Izvaja jo razvijalec, tako da ročno preveri vse implementirane spremembe. Če ugotovi odstopanja od načrtovanje informacijske podpore, se vrne k izvajanju aktivnosti [7]. S to aktivnostjo se zagotavlja kakovost programske opreme.

[9] Potisni kodo v razvojno vejo repozitorija. Ko je razvijalec zadovoljen z rezultatom aktivnosti [7] in [8], odda svoje spremembe programske opreme in fizičnega modela podatkovne zbirke v repozitorij izvorne kode. Zaporedje in frekvenca izvedbe aktivnosti [7], [8] in [9] je različna med razvijalci, saj gre za interni proces posameznega razvijalca.

[10] Potisni kodo v produkcijsko vejo repozitorija. Število vej v repozitoriju izvorne kode programske opreme je različno med projekti. Praviloma sta v naročniškem (ii) primeru izvedbe procesa vsaj dve veji, razvojna in produkcijska. To je predvsem zaradi večjega števila razvijalcev, ki sodelujejo pri projektu. Izvajanje aktivnosti [7] do [10] je močno povezano med sabo. Orodje za upravljanje repozitorijev izvorne kode je del integriranega razvojnega okolja.

[11] Namesti spremembe programske opreme v testno okolje. Po končani implementaciji in osnovnem testiranju sprememb programske opreme v razvojnem okolju nastopi še testiranje ostalih vlog v procesu. Predpogoj je dosegljivo testno okolje. Aktivnost izvede razvijalec,

tako, da artefakte in spremembe podatkovnega modela ročno namesti v testno okolje. Za kreiranje artefaktov se uporabljajo orodja znotraj integriranih razvojnih okolij.

[12] Testiraj načrtovano informacijsko podporo. Ključna aktivnost zagotavljanja kakovosti programske opreme. V naročniškem (i) primeru izvedbe procesa jo najpogosteje izvajajo tester ter odgovorni osebi za vsebino in izvedbo na strani naročnika. V internem (ii) primeru izvedbe pa jo izvaja svetovalec. Gre za ročno izvedbo integracijskih testov funkcionalnosti programske opreme. Cilj je preverjanje, ali informacijska podpora izpolnjuje zahteve oziroma poslovne cilje. V naročniškem (i) primeru izvedbe procesa so vhod v aktivnost testni scenariji iz aktivnosti [4], v internem (ii) primeru izvedbe procesa pa seznam opredeljenih zahtev iz aktivnosti [2]. Če se pri izvajanju aktivnosti ugotovijo pomanjkljivosti ali neskladja z zahtevami, se ta preko internih orodij za komunikacijo posredujejo razvijalcu v ponovno implementacijo.

[13] Dokumentiraj ugotovitve pri testiranju. V naročniškem (i) primeru izvedbe procesa se ugotovitve pri izvajanju aktivnosti [12] dokumentirajo v zapisnik testiranja. Dokument služi kot poročilo o stanju trenutne razvojne naloge in kot vhod v naslednjo iteracijo. Podpora izvedbi aktivnosti so tekstovni urejevalniki besedila.

[14] Podpiši zapisnik testiranja. V naročniškem (i) primeru izvedbe procesa se po testiranju načrtovane informacijske podpore podpiše zapisnik o testiranju. Po končanem testiranju s strani izvajalca (izbrano podjetje) ga podpiše tester, po končanem testiranju s strani naročnika pa odgovorna oseba za izvedbo. Izvedba te aktivnosti pomeni začetek testiranja s strani naročnika oziroma konec trenutne iteracije izvedbe procesa razvoja programske opreme.

[15] Dokumentiraj zahteve za naslednjo iteracijo. Kadar se pri testiranju informacijske podpore s strani naročnika ugotovijo pomanjkljivosti oziroma neskladja s specifikacijo razvojne naloge, se zapisnik o testiranju dopolni z novimi zahtevami za naslednjo iteracijo izvedbe procesa. Ta aktivnost se izvaja samo v naročniškem (i) primeru izvedbe procesa. Najpogosteje jo izvede odgovorna oseba za vsebino.

4.2.3 Analiza procesa

V izbranem podjetju se, ne glede na način izvedbe procesa, uporablja agilen pristop k razvoju programske opreme, čeprav se pri posamezni iteraciji v naročniškem (ii) primeru izvedbe, opazijo tudi vidiki tradicionalnega pristopa:

- Jasno definirani trenutki v procesu, kadar posamezna iteracija preide v naslednjo fazo razvoja. Faze si sledijo zaporedno, kar pomeni, da med implementacijo ali načrtovanjem ni sprememb v prvotnih zahtevah. Če se med iteracijo (večinoma med izvajanjem aktivnosti ročnega testiranja) pojavijo potrebe po spremembi zahtev, se te vključilo v naslednjo iteracijo.

- Pisanje podrobne dokumentacije in njeno potrjevanje. Rezultat posamezne faze razvoja je podrobno dokumentiran in služi kot vhod v naslednjo fazo. Vodijo se spremembe dokumentacije in njeni avtorji. Podpis posameznega dokumenta označuje prehod med fazami.

Vendar pa način izvedbe agilnega pristopa ni povsem enak za vse primere izvedbe procesa razvoja. Razlika ni samo med naročniškimi in internimi projekti, temveč se proces razlikuje tudi med posameznimi oddelki in ekipami. Razloga sta dva: velikost podjetja in pomanjkanje predpisanega načina dela na nivoju celotnega podjetja. Ekipe pri projektih so majhne, včasih tudi samo dve osebi ali celo posameznik, zato je posledično več individualnosti. Če je nekdo sam pri projektu, hitreje opusti izvedbo tistih aktivnosti v procesu, ki se mu zdijo nepotrebne. Ekipa ali posameznik izvede proces razvoja na svoj način. Sledi samo dobrim primerom prakse, ki pa niso predpisane na nivoju podjetja.

V nadaljevanju so povzete težave in pomanjkljivosti v procesu razvoja in pri izvajanju posameznih aktivnosti, ki sem jih identificiral pri analizi in ki nudijo možnosti za izboljšave.

Ni jasno predpisanega načina izvedbe procesa razvoja na nivoju celotnega podjetja. Zato se vsak nov projekt izpelje malo po svoje. Način izvedbe je predvsem odvisen od velikosti ekipe in narave projekta. Ker gre za manjše podjejte, je omejeno z viri. Zato na nekaterih projektih sodelujejo samo dve ali celo ena oseba. Posledično se:

- Opušča izvajanje določenih aktivnosti, saj se s tem privarčuje čas. Zmanjšan obseg načrtovanja in testiranja običajno privede do slabše kvalitete končnega produkta.
- Težje se menjuje izvajalce med različnimi projekti, saj se mora nov član ekipe najprej privaditi na specifični način dela.
- Ker vsaka ekipa ali posameznik dela samo po svojem ustaljenem postopku, ni težnje po vpeljavi novosti ali izboljšav procesa razvoja.

Ker pri določenih projektih delajo samo posamezniki, se povečuje individualnost v podjetju. Posledično je manj sodelovanja med oddelki in ekipami. Dlje časa kot nekdo individualno deluje na določenem področju, težje bo sprejel nove člane v svoj delovni proces, saj se z individualnostjo veča odpor do sprememb.

Zaradi večje individualnosti se v podjetju težje širi kolektivno znanje. Osebe, ki so svetovalci in razvijalci hkrati in so v celoti sami odgovorni za določen del programske opreme, imajo ogromno vsebinskega znanja. To znanje prinaša poslovno vrednost izbranemu podjetju. Vendar se zaradi načina dela težje širi med ostale zaposlene.

Tabela 4 predstavlja razvrstitev aktivnosti na podlagi izvedbe analize z metodo dodajanja vrednosti. Aktivnosti so razvrščene v naslednje tri klasifikacije:

- (A) Aktivnosti, ki dodajajo vrednost za stranke (zunanje ali notranje) in so stranke zanje pripravljene plačati.

- (B) Aktivnosti, ki so nujne za izvedbo procesa in dodajajo vrednost za stranke.
- (C) Aktivnosti, ki ne dodajajo vrednosti.

Tabela 4: Razvrstitev aktivnosti v procesu razvoja glede na dodajanje vrednosti

Aktivnost	**Razvrstitev**
[1] Analiziraj vsebino zahtev	A
[2] Dokumentiraj vsebino zahtev	C
[3] Načrtuj informacijsko podporo	B
[4] Načrtuj informacijsko podporo in testne scenarije	B
[5] Dokumentiraj načrtovano informacijsko podporo	C
[6] Potrdi specifikacijo razvojne naloge	/
[7] Implementiraj specifikacije	A
[8] Testiraj spremembe programske opreme	B
[9] Potisni kodo v razvojno vejo repozitorija	C
[10] Potisni kodo v produkcijsko vejo repozitorija	C
[11] Namesti spremembe programske opreme v testno okolje	A ali B
[12] Testiraj načrtovano informacijsko podporo	B
[13] Dokumentiraj ugotovitve pri testiranju	C
[14] Podpiši zapisnik testiranja	C
[15] Dokumentiraj zahteve za naslednjo iteracijo	C

Vir: lastno delo.

Aktivnosti, za katere so stranke pripravljene plačati, sta samo dve: analiza vsebine zahtev (svetovanje) in implementacija zahtev. Kadar podjetje izvaja tudi administrativna dela, so stranke pripravljene plačati tudi izvajanje nadgradenj programske opreme, za vse ostale aktivnosti pa niso pripravljene plačati. Aktivnost [6] je izključena iz analize dodajanja vrednosti, saj jo v nobenem primeru ne izvaja vloga na strani izbranega podjetja.

Aktivnosti [3], [4], [8] in [12] so nujne za izvedbo procesa in dodajajo vrednost za stranke. To so aktivnosti, ki se izvajajo v fazah načrtovanja in testiranja programske opreme. Brez predhodnega načrtovanja se težko izvede implementacija. Z načrtovanjem se izdela koncept

končne rešitve, pripravi se arhitekturna in infrastrukturna zasnova, oblikujejo se uporabniški vmesniki. Vse to pripomore k boljšemu in kvalitetnejšemu produktu, kar pa pomeni dodano vrednost za stranko.

Tudi brez faze testiranja ni mogoče izvesti procesa razvoja programske opreme. Ročno testiranje je ključna aktivnost zagotavljanja kakovosti programske opreme v izbranem podjetju. S testiranjem se dviga kvaliteta programske opreme in posledično tudi vrednost za stranke.

Ostanejo aktivnosti, ki pa ne dodajajo vrednosti za stranke. To so obvladovanje izvorne kode in delo z dokumentacijo. Obvladovanje izvorne kode je integralni del implementacije in se mu ni mogoče izogniti. Postopek dela z dokumentacijo pa je odvisen od narave projekta. Če je pisanje obsežne dokumentacije, njeno podpisovanje ter vodenje sprememb ena izmed zahtev naročnika, se tudi temu ni mogoče izogniti. Pri internih projektih pa se v izbranem podjetju ne piše obsežne dokumentacije. Analiza dodajanja vrednosti procesa razvoja ni identificirala aktivnosti, ki se jim je mogoče izogniti. So pa možnosti za izboljšave pri načinu izvedbe posameznih aktivnosti.

Mlajši zaposleni so prav tako izrazili težave pri razumevanju vsebine in novih zahtev strank. Razloga sta najverjetneje dva: že omenjena individualnost, ki pomeni, da imajo vsebinsko znanje samo določeni posamezniki. Tista vsebina, ki pa je dokumentirana, je večina v tekstovni obliki, ki je težja za prvo razumevanje na konceptualnem nivoju.

Razvijalci so izpostavili težave s preveliko kompleksnostjo določenih delov programske opreme in njeno vse težje obvladovanje, tudi še v času razvoja. To je najverjetneje posledica arhitekturnega tehniškega dolga, ki nastane, ko se vsebinske zahteve močno spremenijo, arhitektura programske opreme pa ostane enaka, saj se prestrukturiranje v tistem trenutku ne izvede. Največkrat zaradi pomanjkanja časa in virov.

Razlog za kompleksnost lastne programske opreme izbranega podjetja je predvsem to, da zagotavlja informacijsko podporo mnogim različnim strankam. Vsak poslovni proces ima svoje specifike in izjeme in vse je potrebno podpreti. Dosedanja rešitev podjetja je bila, da določene dele programske opreme razvija samo za določeno stranko. Tako lahko obvladuje informacijsko podporo zahtevnim poslovnim procesom, brez vpliva na osnovno arhitekturo rešitve, ampak tako pride do podvajanja funkcionalnosti in večanja količine izvorne kode. To pa otežuje obvladovanje in distribucijo sistema.

4.2.4 Predlogi za izboljšave

Uvedba vloge skrbnika temeljnega poslovnega procesa v podjetju. Uvedba skrbništva je osnovni korak k poenotenju načina izvedbe procesa razvoja programske opreme na nivoju celotnega podjetja. Skrbnik procesa bi skrbel za učinkovito in kakovostno izvedbo procesa,

ne glede na naravo in velikost projekta ali sestavo ekipe, ki deluje pri projektu. Naloge skrbnika procesa so (Kovačič, 2019):

- *Modeliranje, analiziranje in izdelovanje predlogov prenove procesa*. Skrbnik je odgovoren za izdelavo modela trenutne izvedbe procesa ter konstantno analiziranje in identificiranje možnosti za izboljšave. S takšnim pristopom bi izbrano podjetje izvedlo prvi korak k poenotenju izvedbe procesa.
- *Sodelovanje pri dolgoročnem načrtovanju procesa in usklajevanje ciljev z lastnikom procesa*. Z boljšim vpogledom v izvajanje svojega temeljnega poslovnega procesa bi izbrano podjetje tudi lažje načrtovalo uvedbo sprememb v procesu. Skrbnik procesa bi prav tako skrbel, da je način izvedbe v skladu s kratkoročnimi in dolgoročnimi poslovnimi cilji izbranega podjetja.
- *Skrb za kakovost in učinkovitost izvedbe procesa ter konstantno izboljševanje značilnosti procesa*. Skrbnik procesa bi v izbranem podjetju skrbel, da se proces razvoja programske opreme izvaja kakovostno in učinkovito, ne glede na sestavo ekipe ali oddelek, ki ga izvaja. S poenotenim načinom dela bi se hitreje prepoznale značilnosti procesa, ki imajo možnost za izboljšavo.
- *Dvigovanje zanesljivosti ter doslednosti izvajanja aktivnosti v procesu*. Zaradi individualnosti v izbranem podjetju je tudi večji odpor do novih sprememb v načinu dela. Skrbnik procesa bi skrbel za spodbujanje zaposlenih k zanesljivi in dosledni izvedbi predvidenih aktivnosti v procesu.

Kot dodaten organizacijski predlog za izboljšavo procesa predlagam, da **pri posameznem projektu ne sodelujejo manj kot tri osebe**. S tem bi izbrano podjetje zmanjšalo individualnost, saj bi zaposleni spoznali tudi druge načine dela. Če v določenem trenutku ni dovolj virov za dosego predloga, predlagam, da je vsaj na začetnem projektnem sestanku prisotnih čim več zaposlenih. Tako se širi splošno zavedanje in informacije o projektih, ki jih trenutno izvaja podjetje.

Prakticiranje XP-tehnike programiranja v paru. Ključen korak k zmanjšanju individualnosti v izbranem podjetju. Določene module programske opreme trenutno razvija in vzdržuje samo ena oseba. Ker je takšna praksa že dlje časa, se je v nekaterih primerih razvil sebičen odnos do programske opreme. S prakticiranjem programiranja v paru bi izbrano podjetje lažje vzpostavilo koncept kolektivnega lastništva in odgovornosti za programsko opremo, katerega ključna ideja je, da je programska oprema v lasti celotne ekipe in ne posameznikov (Weinberg, 1971). Ob zmanjšanju individualnosti bi prakticiranje programiranja v paru izboljšalo tudi:

- *Prenos kolektivnega in vsebinskega znanja v izbranem podjetju*. Še posebej pri sodelovanju parov, kjer je občutna razlika v izkušnjah in znanju posameznikov, saj novinec največ pridobi, kadar ekspert razloži svoj miselni proces. Čeprav se morajo eksperti dodatno potruditi za prenos znanja, je takšna tehnika koristna tudi za njih, saj

jim postavljanje vprašanj novincev pomaga razmišljati o svojih praksah in se tako naučiti iz izkušnje (Plonka, Sharp, Van Der Linden & Dittrich, 2015).

- *Spodbujanje odločitev za prestrukturiranje programske kode*. Če več oseb hkrati razvija ali dograjuje posamezni del programske opreme in je v ekipi vzpostavljeno kolektivno lastništvo, razvijalci hitreje podprejo izziv za prestrukturiranje določenega dela programske kode (Sommerville, 2016).
- *Kakovost programske opreme*. Programiranje v paru deluje tudi kot neformalni pregled programske kode. Posledično se zmanjša število napak in stroški vzdrževanja (Williams, Kessler, Cunningham & Jeffries, 2000).

Uporaba XP-tehnike uporabniških zgodb in BPMN-vizualizacij poslovnih procesov. Na takšen način bi lahko izbrano podjetje izboljšalo razumevanje vsebine in zahtev strank v fazah analize in načrtovanja programske opreme. Uporabniške zgodbe je lažje razumeti kot tradicionalno tekstovno dokumentacijo (Sommerville, 2016), prav tako njihova uporaba izboljšuje produktivnost razvijalcev ter končno kakovost programske opreme (Lucassen, Dalpiaz, van der Werf & Brinkkemper, 2016). Z izdelovanjem BPMN-vizualizacij poslovnih procesov, ki jih izbrano podjetje informacijsko podpira, pa bi se izboljšalo razumevanje konteksta ter vrstni red izvajanja in medsebojna odvisnost uporabniških zgodb (Trkman, Mendling & Krisper, 2016). Modeli poslovnih procesov bi tako komplementarno dopolnjevali uporabniške zgodbe pri analizi novih zahtev strank.

Prakticiranje XP-tehnike prestrukturiranja programske kode. Uporaba te prakse bi imela v izbranem podjetju pozitiven vpliv na produktivnost razvijalcev in končno kakovost programske opreme (Moser, Abrahamsson, Pedrycz, Sillitt & Succi, 2008). Temelja lastnost programske opreme je, da se s spremembami poslabša njena struktura (Lehman, 1996). Posledično je vsako nadaljnjo spremembo težje implementirati. Prakticiranje prestrukturiranja programske kode bi v izbranem podjetju izboljšalo strukturo in berljivost programske opreme, kar bi olajšalo implementacije prihodnjih sprememb (Sommerville, 2016). Ob prakticiranju tehnike prav tako predlagam:

- *Modularen pristop pri izvajanju prestrukturiranja*. Koncept modularnosti je ključnega pomena pri prestrukturiranju programske kode, saj omogoča kasnejši poseg v posamezne dele programske opreme, ne da bi s tem ogrozili delovanje celotnega sistema. Na takšen način bi lahko izbrano podjetje minimiziralo tehniške dolgove na dolgi rok ter privarčevalo pri stroških vzdrževanja programske opreme (Martini, Sikander & Madlani, 2018).
- *Pravočasna odločitev za izvedbo prestrukturiranja*. Predlagam, da izbrano podjetje ne odlaša predolgo z izvedbo prestrukturiranja programske opreme. Torej, da ne čaka na naročilo novih zahtev ali na poslovno neupravičen padec produktivnosti zaradi slabe strukture, ampak da se za prestrukturiranje odloči pravočasno, tudi kadar bi to pomenilo dodaten napor, ki se v tistem trenutku ne zdi poslovno upravičen. V takšnih primerih je predvsem potrebna podpora lastnikov procesa, zato predlagam, da operativna raven

skuša odgovornim osebam predstaviti vse prednosti, ki bi jih prineslo prestrukturiranje (Chen, Xiao, Wang, Osterweil & Li, 2014).

Nov koncept arhitekturne zasnove sistema za informacijsko podporo različnim poslovnim procesom. Na podlagi predlaganega koncepta lahko izbrano podjetje izvede prestrukturiranje obstoječe ali razvoj nove programske opreme za informacijsko podporo različnim poslovnim procesom. Tako bi lahko izbrano podjetje rešilo težave, s katerimi se sooča zaradi kompleksnosti lastne programske opreme.

V nadaljevanju predstavljen koncept omogoča podporo različnim poslovnim procesom znotraj ene distribucije programske opreme, ne glede na specifike in pravila posameznega poslovnega procesa.

Slika 12 prikazuje miselni vzorec koncepta arhitekturne zasnove. Vsako podjetje je organizirano v določeni strukturi. Običajno je sestavljeno iz več enot ali oddelkov, kjer zaposleni opravljajo svoje delo na različnih delovnih mestih. Vsak zaposleni ima svoje zadolžitve znotraj dodeljenih pooblastil. Pooblastilo predstavlja pravico izvajanja dela za določeno entiteto. Na primer oseba, ki skrbi za veleprodajo, je zadolžena za poslovanje s svojimi kupci. Torej ima pooblastila za vlogo prodajalca (delovno mesto) za določene poslovne partnerje (entitete). Način, kako podjetje organizira in zadolžuje svoje zaposlene oziroma način, s katerim dodeli pooblastila zaposlenim za posamezno vlogo, lahko imenujemo shema avtorizacije.

Zaposleni med izvajanjem svojega dela običajno obvladujejo določene podatke. Večinoma jih obvladujejo na naslednja dva načina:

- *Enostavno ažuriranje podatkov*. Običajno se enostavno ažuriranje podatkov izvaja pri matičnih podatkih (na primer poslovni partnerji, artikli in stroškovna mesta) in pri osnovnih šifrantih (na primer države, pošte in naslovi).
- *Izvajanje procesa*. Običajno se izvajanje poslovnega procesa izvaja pri prometnih podatkih različnih vrst entitet, kot so dokumenti (naročilo kupca, prevzem od dobavitelja in pogodba), obračuni plače, inventure in podobno.

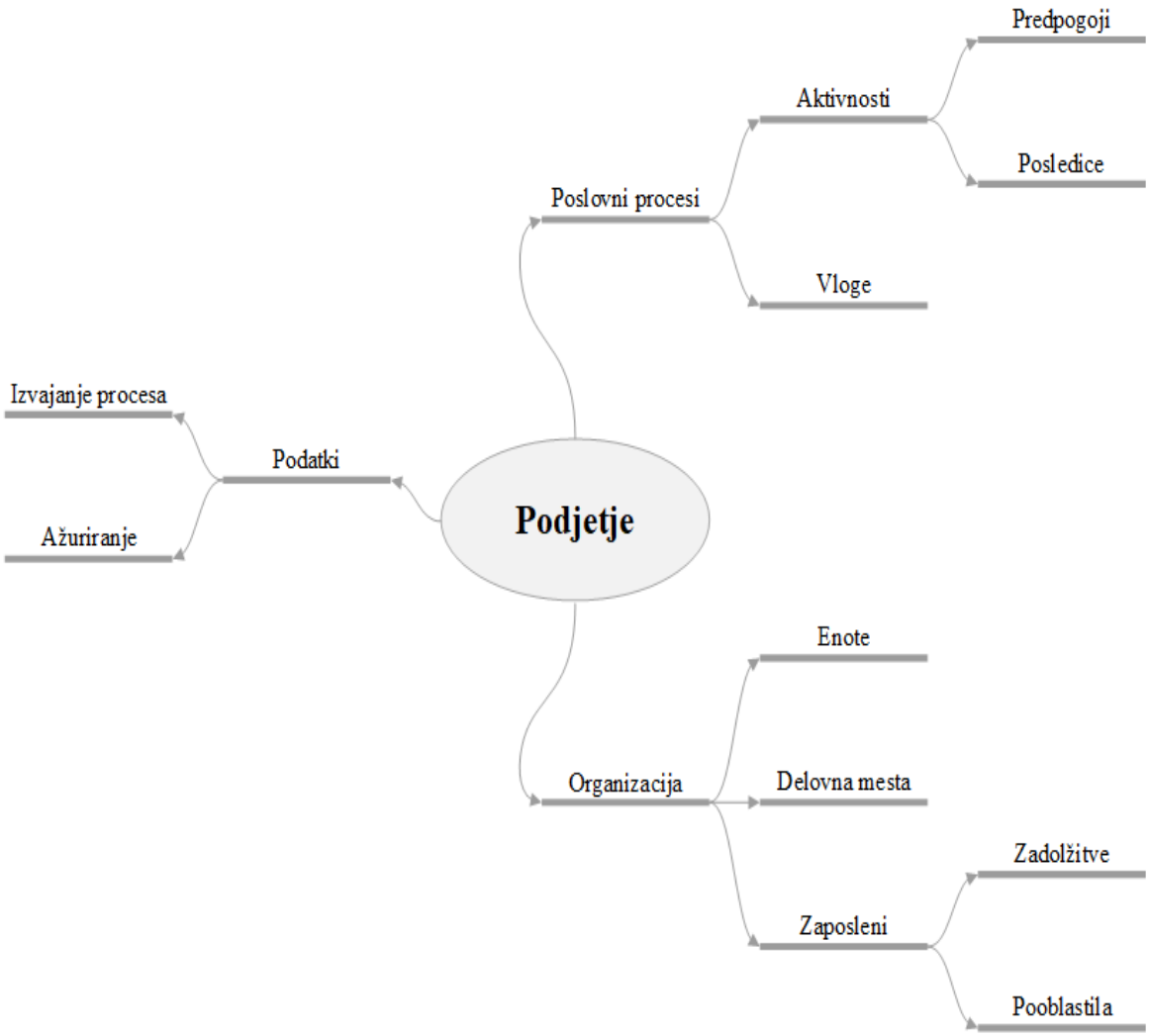
V določenem podjetju se izvaja več različnih poslovnih procesov. Ne glede na vrsto poslovnega procesa (temeljni, vodstveni ali podporni) ima vsak proces opredeljen seznam aktivnosti, ki se izvedejo, in vsaka aktivnost ima opredeljeno vlogo v procesu, ki jo lahko izvaja. Na primer aktivnost potrjevanja naročila kupca potrdi uporabnik v vlogi prodajalca, ki je avtoriziran za delo z omenjenim kupcem.

Prav tako lahko ima določena aktivnost opredeljene predpogoje, ki morajo veljati, preden se lahko izvede aktivnost, ter posledice, ki se zgodijo, ko se uspešno izvede aktivnost. Na primer pred potrjevanjem naročila kupca morajo biti opredeljene vse cene na zahtevanih

artiklih in po uspešnem potrjevanju naročila se samodejno pripravi dokument za izdajo blaga iz skladišča.

Na podlagi opisanih lastnosti koncepta se lahko implementira sistem, ki je zmožen v vsakem trenutku vrniti seznam aktivnosti (akcij), ki jih lahko zaposleni (uporabnik) izvede za podatke (vrsta entitete), za katere ima pooblastilo (shema avtorizacije).

Slika 12: Arhitekturna zasnova sistema za informacijsko podporo različnim procesom



Vir: lastno delo.

4.3 Vzdrževanje programske opreme

Operativno delo v izbranem podjetju je vzdrževanje programske opreme. V procesu vzdrževanja je vključen tudi podproces podpore in svetovanja strankam. Vzdrževanje se praviloma prične s prehodom stranke v produkcijsko uporabo določene programske opreme.

4.3.1 Opredelitev procesa

Tabela 55 opisuje dogodke v procesu vzdrževanja programske opreme.

Tabela 5: Dogodki v procesu vzdrževanja programske opreme

Dogodek	**Vrsta**	**Opis**
Vprašanje / zahteva / problem presledek	Začetni	Proces vzdrževanja programske opreme se začne, kadar njeni uporabniki potrebujejo pomoč pri uporabi. Potrebujejo svetovanje pri reševanju določenega problema, imajo zahtevo za spremembo programske opreme ali pa samo vprašanje glede njenih funkcionalnosti. Dogodek sproži stranka.
Prejeto sporočilo	Vmesni	Uporabniki programske opreme preko sistema za podporo strankam pošiljajo sporočila z različno vsebino. Prejeto sporočilo sproži začetek procesa vzdrževanja programske opreme na strani izbranega podjetja.
Prejet odgovor	Vmesni	Kadar se začetni dogodek ne nadaljuje v dejansko spremembo programske opreme, stranka prejme odgovor z rešitvijo problema. Prejeti odgovor sproži nadaljevanje podprocesa podpore in svetovanja na strani stranke.
Prejeta nadgradnja programske opreme	Vmesni	Kadar začetni dogodek privede do spremembe programske opreme, stranka prejme nadgradnjo, ki vsebuje nove, dopolnjene ali popravljene funkcionalnosti. Prejeta nadgradnja sproži nadaljevanje procesa vzdrževanja programske opreme na strani stranke.
Vprašanje / problem je rešen	Končni	Podproces podpore in svetovanja strankam se zaključi po rešitvi prvotnega problema. Dogodek sproži stranka ob zaključku sporočila.
Sprememba programske opreme je na voljo	Končni	Proces vzdrževanja programske opreme se zaključi, ko je sprememba programske opreme na voljo za namestitev v produkcijsko okolje. Dogodek sproži stranka ob zaključku sporočila.

Vir: lastno delo.

Tabela 6 opisuje organizacijske enote, ki izvajajo proces vzdrževanja programske opreme v izbranem podjetju.

Tabela 6: Vloge, ki izvajajo proces vzdrževanja programske opreme

Vloga	**Opis**
Svetovalec	Izvaja podproces podpore in svetovanja strankam.
Razvijalec	Izvaja temeljni poslovni proces razvoja in vzdrževanja programske opreme.
DBA	Administrator podatkovnih zbirk (angl. Database Administrator). Skrbi za dostavo programske opreme v končna okolja.
Stranka	Naročnik oziroma končni uporabnik programske opreme. Praviloma vedno sproži začetek izvedbe procesa.

Vir: lastno delo.

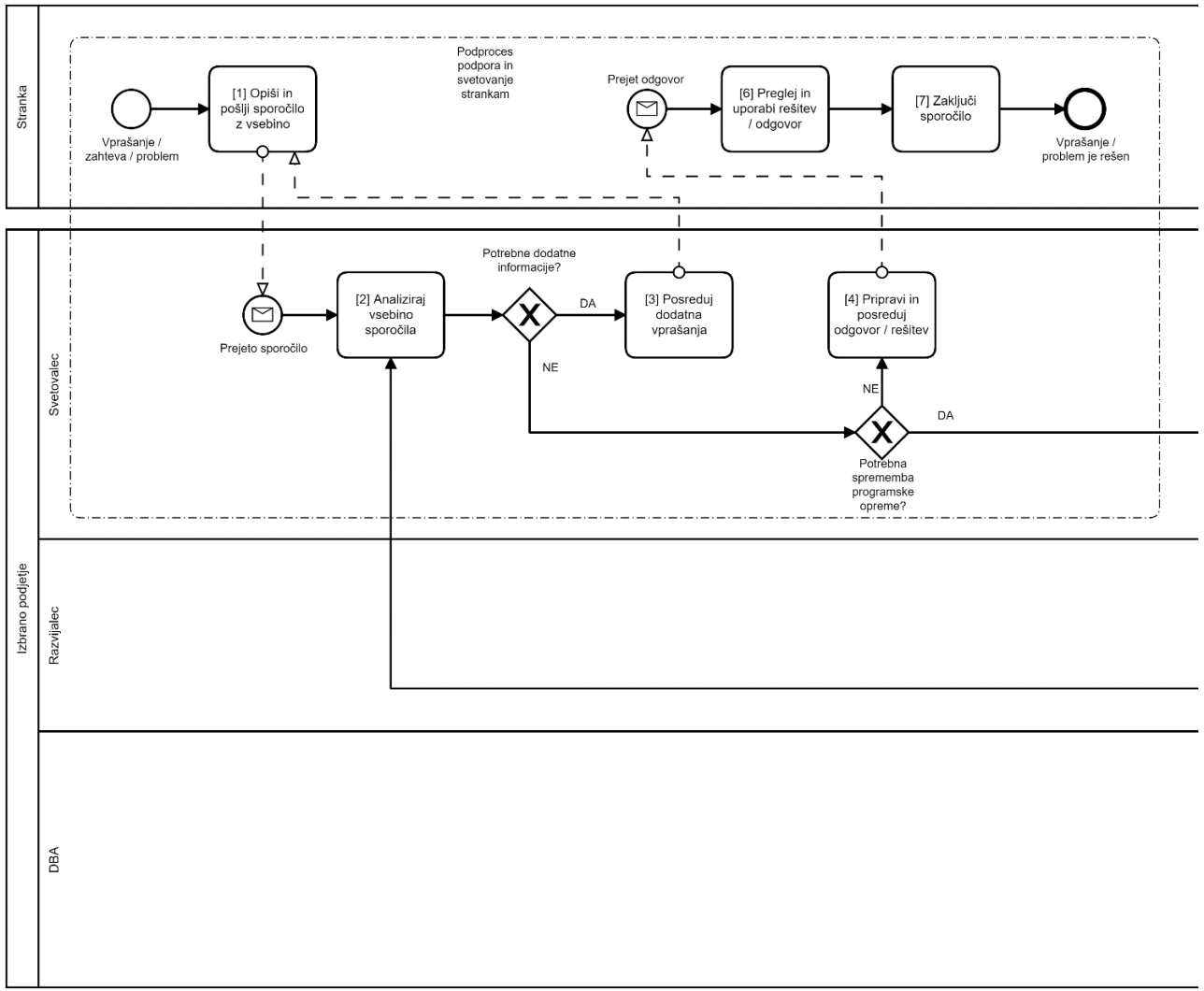
4.3.2 Modeliranje procesa

Na slikah 13 in 14 je prikazana BPMN-vizualizacija procesa vzdrževanja programske opreme v izbranem podjetju. Model predstavlja eno iteracijo, od sporočila, ki prispe v sistem za podporo strankam do pripravljene nadgradnje spremembe programske opreme za namestitev v končno produkcijsko okolje. V eni nadgradnji so lahko vsebovane spremembe iz več prvotnih sporočil.

Iz modeliranja sem izpustil izvedbo nadgradnje v končno produkcijsko okolje, saj je izbrano podjetje vzdržuje programsko opremo tipa SaaP. Torej je odgovornost obvladovanja produkcijskega okolja v celoti na strani stranke. Prav tako sem iz modeliranja izpustil razne poslovne vidike, kot na primer vzdrževalne pogodbe ter spremljanje in vrednotenje dela.

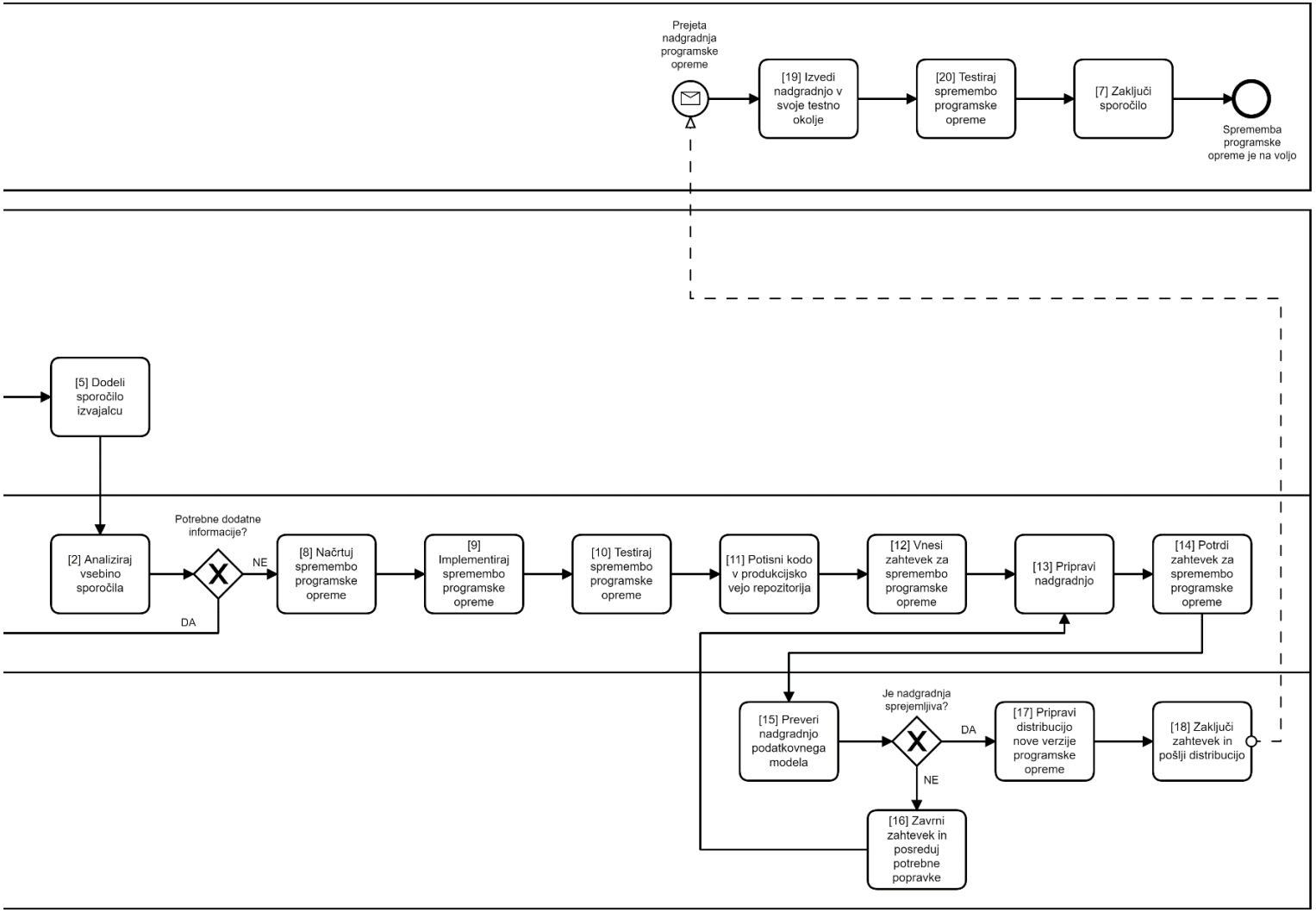
V BPMN-vizualizaciji je označen obseg podprocesa podpore in svetovanja strankam. Praviloma ga izvajata samo stranka in svetovalec. Obstajajo pa primeri izvedbe, kjer svetovalec ni vključen, temveč razvijalec sam komunicira s stranko. V praksi to pomeni, da prevzame sporočilo stranke in si ga sam dodeli. V večini takšnih primerov izvedbe gre za specifično vsebino, kjer je jasno določeno, kdo bo prevzel delo.

Slika 13: BPMN-vizualizacija procesa vzdrževanja



Vir: lastno delo.

Slika 14: BPMN-vizualizacija procesa vzdrževanja (nad.)



Vir: lastno delo.

V nadaljevanju so opisane lastnosti, izvajalci, cilji ter rezultati posameznih aktivnosti, ki se izvajajo v procesu vzdrževanja programske opreme.

[1] Opiši in pošlji sporočilo z vsebino. Začetna aktivnost v procesu vzdrževanja programske opreme, ki se začne s podprocesom podpore in svetovanja strankam. Izvaja jo končni uporabnik oziroma stranka, ki opiše ter odda sporočilo v sistem za podporo strankam. Tip in vsebina sporočila sta različna. V večini primerov gre za vsebinsko vprašanje oziroma problem pri uporabi programske opreme ali pa za zahtevo za spremembo programske opreme. Zahteva je lahko zaradi napake v delovanju programske opreme ali potrebe po dograditvi programske opreme. Potreba lahko nastane zaradi zakonske spremembe (zunanji dejavnik) ali spremembe poslovnega procesa stranke (notranji dejavnik). Izhod aktivnosti je prejeto sporočilo z opredeljenimi zahtevami, pripravljeno za obdelavo s strani izbranega podjetja.

[2] Analiziraj vsebino sporočila. Aktivnost izvajata svetovalec in razvijalec. Analizirata vsebino prejetega sporočila in hkrati ocenita, ali je danih dovolj informacij za pripravo odgovora z rešitvijo problema oziroma za začetek načrtovanja spremembe programske opreme. Za izvedbo aktivnosti se uporablja sistem za podporo strankam. Aktivnost lahko primerjamo z aktivnostjo [1] v procesu razvoja programske opreme, saj gre za opredeljevanje potreb za novo informacijsko podporo ali spremembo obstoječe.

[3] Posreduj dodatna vprašanja. Kadar v sporočilu ni dovolj danih informacij za nadaljevaje procesa, se stranki posreduje dodatna vprašanja. V večini primerov gre za potrebo po natančnejši opredelitvi problema, opis aktivnosti v poslovnem procesu stranke, kjer je nastal problem, ali potrebo po dodatnem gradivu (zaslonske slike, opis napake in podobno). Aktivnost najpogosteje izvaja svetovalec. Za izvedbo aktivnosti se uporablja sistem za podporo strankam.

[4] Pripravi in posreduj odgovor / rešitev. Kadar sporočilo vsebuje vse potrebne informacije za nadaljevanje procesa in ni potrebe za spremembo programske opreme, se pripravi in posreduje odgovor z rešitvijo problema. Aktivnost najpogosteje izvaja svetovalec. Za pripravo rešitve uporablja testna okolja stranke in izbranega podjetja, za posredovanje odgovora pa sistem za podporo strankam.

[5] Dodeli sporočilo izvajalcu. Kadar je potrebna sprememba programske opreme, svetovalec dodeli sporočilo primernemu razvijalcu in določi predviden rok izvedbe. Za izvedbo aktivnosti se uporablja sistem za podporo strankam.

[6] Preglej in uporabi rešitev / odgovor. Kadar se začetno sporočilo stranke ne nadaljuje v spremembo programske opreme, stranka prejme odgovor z rešitvijo problema, ki jo uporabi oziroma izvede v svojem produkcijskem okolju. Za izvedbo aktivnosti se uporablja sistem za podporo strankam in produkcijska okolja stranke.

[7] Zaključi sporočilo. Končna aktivnost v procesu vzdrževanja programske opreme. Najpogosteje jo izvaja stranka. Zaključek sporočila se izvede po uporabi rešitve v produkcijskem okolju oziroma po opravljenem testu sprememb programske opreme. Če se pri izvedbi pojavijo pomanjkljivosti ali potrebe po novih zahtevah, stranka ustvari novo sporočilo in začne se nova izvedba procesa. Za izvedbo aktivnosti se uporablja sistem za podporo strankam.

[8] Načrtuj spremembo programske opreme. Aktivnost izvaja razvijalec. Izvedba aktivnosti je enaka izvedbi aktivnosti [3] v procesu razvoja programske opreme. Razlika je le v obliki vhoda v aktivnost. V procesu vzdrževanja programske opreme je načeloma obseg zahtev manjši, kar pomeni tudi manj načrtovanja.

[9] Implementiraj spremembo programske opreme. Aktivnost izvaja razvijalec. Izvedba aktivnosti je enaka izvedbi aktivnosti [7] v procesu razvoja programske opreme.

[10] Testiraj spremembo programske opreme. Aktivnost izvaja razvijalec. Izvedba aktivnosti je enaka izvedbi aktivnosti [8] v procesu razvoja programske opreme.

[11] Potisni kodo v produkcijsko vejo repozitorija. Aktivnost izvaja razvijalec. Izvedba aktivnosti je enaka izvedbi aktivnosti [10] v proces razvoja programske opreme.

[12] Vnesi zahtevek za spremembo programske opreme. Ko je sprememba programske opreme implementirana in odložena v produkcijsko vejo repozitorija, razvijalec vnese zahtevek za spremembo programske opreme v sistem za vodenje sprememb. V zahtevku opredeli programsko opremo (aplikacijo), ki je bila spremenjena ter stranko (poslovnega partnerja), ki je iniciator spremembe. Prav tako opiše vsebino spremembe oziroma vnese tekst, ki ga bodo končni uporabniki prebrali kot opombo pri izdaji nove verzije programske opreme. Aktivnost najpogosteje izvaja razvijalec, kadar pa gre za večjo spremembo, ki bo trajala dlje časa in jo bo izvajalo več razvijalcev hkrati, zahtevek za spremembo programske opreme lahko vnese tudi svetovalec. Zahtevek tako služi kot povezava in združitev več sprememb hkrati.

[13] Pripravi nadgradnjo. Za vsako spremembo programske opreme je potrebno pripraviti nadgradnjo, ki programsko opremo ustrezno nadgradi z novo (višjo) verzijo. Nadgradnja je sestavljena iz sprememb aplikacije (vmesnik in srednji sloj) in sprememb baze (podatkovna zbirka). Aktivnost izvaja razvijalec. Za pripravo sprememb aplikacije uporablja orodja v integriranem razvojnem okolju, s katerimi kreira potrebne artefakte. Za pripravo sprememb baze uporablja sistem za vodenje sprememb, ki omogoča samodejni vnos podatkovnih objektov, ki so bili spremenjeni v določenem obdobju. Nadgradnjo kompleksnih sprememb podatkovnega modela pripravi ročno. Za posamezno spremembo baze razvijalec označi podatkovni model, v katerem se bo izvedla nadgradnja, prav tako tudi označi, ali naj se nadgradnja izvede izključno za stranko, ki je opredeljena v zahtevku za spremembo programske opreme. Izhod aktivnosti je torej zahtevek za spremembo programske opreme,

ki vsebuje vse potrebne spremembe slojev aplikacije, ki bodo v končnem okolju primerno posodobili artefakte in podatkovni model na novo verzijo programske opreme.

[14] Potrdi zahtevek za spremembo programske opreme. Ko razvijalec končna s pripravo nadgradnje, potrdi zahtevek. S tem posreduje zahtevek v pregled in nadaljnjo obravnavo. Za izvedbo aktivnosti se uporablja sistem za vodenje sprememb.

[15] Preveri nadgradnjo podatkovnega modela. Aktivnost izvaja DBA. Cilj aktivnosti je preverjanje izvedljivosti nadgradnje in hkrati pregledovanje načrtovanih sprememb podatkovnega modela. S pregledom nadgradnje se zagotavlja kakovost programske opreme. Za izvedbo aktivnosti se uporablja sistem za vodenje sprememb in testna okolja.

[16] Zavrni nadgradnjo in posreduj potrebne popravke. Kadar nadgradnja ni sprejemljiva, se zavrne zahtevek in posreduje razvijalcu seznam napak oziroma potrebnih popravkov. Aktivnost izvaja DBA. Za izvedbo aktivnosti se uporabljajo interna orodja za komunikacijo.

[17] Pripravi distribucijo nove verzije programske opreme. Kadar je pripravljena nadgradnja s strani razvijalca sprejemljiva, se pripravi distribucijo nove verzije programske opreme, tako da zapakira vse spremembe vseh slojev aplikacije. Aktivnost izvaja DBA. Za izvajanje se uporablja sistem za vodenje sprememb.

[18] Zaključi zahtevek in pošlji distribucijo. Po preverbi nadgradnje in pripravi distribucije se zaključi zahtevek za spremembo programske opreme. Aktivnosti izvaja DBA. Po zaključku postane distribucija snemljiva s strani sistema za izvajanje nadgradenj v končnem okolju. Za izvajanje aktivnosti se uporablja sistem za vodenje sprememb.

[19] Izvedi nadgradnjo v svoje testno okolje. Aktivnost izvaja stranka. Za izvedbo aktivnosti se uporablja sistem za izvajanje nadgradenj, s katerim stranka najprej prenese distribucijo nove verzije programske opreme v svoje omrežje, nato pa izvede nadgradnjo v testno okolje.

[20] Testiraj spremembo programske opreme. Aktivnost izvaja stranka. Za izvedbo aktivnosti uporablja svoja testna okolja. Cilj aktivnosti je preverjanje, ali so bile zahteve začetnega sporočila izpolnjene še pred izvedbo nadgradnje v produkcijsko okolje. Vendar se ta aktivnost včasih tudi opusti. Kadar obstajajo nepravilnosti ali pomanjkljivosti v spremembi programske opreme, stranka pošlje novo sporočilo s primerno vsebino in proces vzdrževanja se ponovi.

4.3.3 Analiza procesa

Tabela 7 predstavlja razvrstitev aktivnosti na podlagi izvedbe analize z metodo dodajanja vrednosti. Aktivnosti so razvrščene v naslednje tri klasifikacije:

- (A) Aktivnosti, ki dodajajo vrednost za stranke (zunanje ali notranje) in so stranke zanje pripravljene plačati.

- (B) Aktivnosti, ki so nujne za izvedbo aktivnosti in dodajajo vrednost za stranke.
- (C) Aktivnosti, ki ne dodajajo vrednosti.

Tabela 7: Razvrstitev aktivnosti v procesu vzdrževanja glede na dodajanje vrednosti

Aktivnost	**Razvrstitev**
[1] Opiši in pošlji sporočilo z vsebino	/
[2] Analiziraj vsebino sporočila	C
[3] Posreduj dodatna vprašanja	C
[4] Pripravi in posreduj odgovor / rešitev	A
[5] Dodeli sporočilo izvajalcu	C
[6] Preglej in uporabi rešitev / odgovor	/
[7] Zaključi sporočilo	C
[8] Načrtuj spremembo programske opreme	B
[9] Implementiraj spremembo programske opreme	A
[10] Testiraj spremembo programske opreme	B
[11] Potisni kodo v produkcijsko vejo repozitorija	C
[12] Vnesi zahtevek za spremembo programske opreme	C
[13] Pripravi nadgradnjo	C
[14] Potrdi zahtevek za spremembo programske opreme	C
[15] Preveri nadgradnjo podatkovnega modela	C
[16] Zavrni zahtevek in posreduj potrebne popravke	C
[17] Pripravi distribucijo nove verzije programske opreme	C
[18] Zaključi zahtevek in pošlji distribucijo	C
[19] Izvedi nadgradnjo v svoje testno okolje	A ali B
[20] Testiraj spremembo programske opreme	B

Vir: lastno delo.

Aktivnosti, za katere so stranke pripravljene plačati, sta samo dve: pomoč in svetovanje strankam ter implementacija zahtev. Kadar izbrano podjetje opravlja tudi administrativna dela, so stranke pripravljene plačati tudi obvladovanje nadgradenj v končnih okoljih, za vse ostale aktivnosti pa niso pripravljene plačati. Aktivnosti [1] in [6] sta bili izključeni iz analize dodajanja vrednosti, saj jih v nobenem primeru ne izvaja vloga na strani izbranega podjetja.

Aktivnosti, ki so nujne za izvedbe procesa in dodajajo vrednost za stranke, so, enako kot v procesu razvoja, aktivnosti, ki se izvajajo v fazi načrtovanja in testiranja programske opreme. Razlogi so enaki kot pri analizi procesa razvoja.

So pa tudi aktivnosti, ki ne dodajajo vrednosti za stranke. Enako kot pri procesu razvoja se določenim ni mogoče izogniti. Vendar pa v nasprotju z razvojem v procesu vzdrževanja obstajajo takšne aktivnosti, ki bi jih izbrano podjetje lahko izločilo iz izvedbe procesa. Te so:

- [12] Vnesi zahtevek za spremembo programske opreme
- [13] Pripravi nadgradnjo
- [14] Potrdi zahtevek za spremembo programske opreme
- [15] Preveri nadgradnjo podatkovnega modela
- [16] Zavrni zahtevek in posreduj potrebne popravke
- [17] Pripravi distribucijo nove verzije programske opreme
- [18] Zaključi zahtevek in pošlji distribucijo

V nadaljevanju so povzete vse težave in pomanjkljivosti v procesu vzdrževanja in pri izvajanju posameznih aktivnosti, ki nudijo možnosti za izboljšave.

Največ težav, s katerimi se izbrano podjetje srečuje v svojem procesu vzdrževanja programske opreme, nastane ob prehodih v produkcijo. Najpogosteje v obliki regresijskih napak (težave z obstoječimi in nespremenjenimi funkcionalnosti) in napak ob izvedbi nadgradnje programske opreme, večinoma zaradi neustreznih postopkov. To so tudi najpogostejše težave, s katerimi se srečujejo razvojna podjetja v praksi (Humble & Farley, 2010). Vzroki za te težave so (velja za izbrano podjetje in tudi na splošno):

- *Ročna priprava in izvedba nadgradenj*. Priprava večine potrebnih sprememb za nadgradnjo podatkovnih zbirk poteka ročno. Izvedba nadgradnje sicer poteka samodejno, ob nadzoru administratorjev, vendar obstajajo primeri, kjer razvijalci sami ter ročno izvajajo nadgradnje, zato pride do napak. Pomanjkanje avtomatiziranih postopkov in načinov dela je ključen problem pri dostavi programske v izbranem podjetju.
- *Preredki prehodi v produkcijo.* Čas, ki preteče od trenutka uskladitve zahteve za spremembo programske opreme, do trenutka, ko je ta zahteva na voljo v produkcijskem okoljem, je predolg. V redkih primerih gre tudi za enoletna obdobja, kar lahko v povezavi s prejšnjim vzrokom povzroči hude posledice. Potrebno je poudariti, da krivda za

preredke prehode v produkcijo ni povsem na strani izbranega podjetja, saj ta uporablja SaaP poslovni model, kar večinoma pomeni, da je za obvladovanje produkcijskega okolja odgovorna stranka.

Izbrano podjetje sicer v določeni meri prakticira zvezno integracijo, saj razvijalci za operativne spremembe uporabljajo samo glavno vejo repozitorija. Frekvenca sprememb s strani posameznika sicer ni vsakodnevna, je pa na dovolj visokem nivoju, da se zaposleni ne srečujejo s težavami pri integraciji sprememb. To je predvsem posledica tega, da je vsak posameznik odgovoren za svoj del programske opreme. Obstaja pa pomanjkanje avtomatiziranih postopov, ki bi se izvedli ob vsaki integraciji sprememb in samodejno preverili pravilnost trenutnega stanja programske opreme.

Vendar pa je zaradi pomanjkanja dodatnih namenskih vej repozitorija težje vzpostaviti dodatna testna okolja za testiranja samo določenih funkcionalnosti, ki se zahtevajo večmesečno implementacijo. Vzrok je pomanjkanje jasne strategije vejitve repozitorija ter pomanjkanje mehanizmov za ustvarjanje testnih okolij za zahtevo.

Svetovalci in tudi razvijalci so izrazili, da so povratne informacije, ki jih imajo na voljo za reševanje nastalih problemov, slabe. Podjetje sicer ima vzpostavljen mehanizem za samodejno pošiljanje napak v sistem za podporo strankam, vendar je zaradi prevelike količine prejetih podatkov težko povezati javljen problem s pravim seznamom napak. Prav tako se reševanje napak prične šele, ko prispe sporočilo stranke v sistem. Gre torej za pomanjkanje preventivnega preprečevanja problemov in tudi neustrezno obvladovanje toka povratnih informacij.

Zadnja pomanjkljivost, ki sem jo identificiral, je pomanjkanje eksperimentiranja pri izboljšavi operativnega dela v podjetju. Gre za podobno težavo, kot sem jo že opisal v analizi procesa razvoja. Zaposleni (oziroma oddelki) delujejo precej individualno in v svojih ustaljenih načinih dela. Zaradi majhnosti podjetja je tudi težje najti čas in vire za iskanje in zagon morebitnih izboljšav, saj so zaposleni preveč obremenjeni z delom, ki neposredno prinaša poslovno vrednost podjetju.

4.3.4 Predlogi za izboljšave

Ključen predlog za izboljšavo, ki je skupen vsem ostalim, je sprememba načina razmišljanja in pristopa k razvoju in vzdrževanju programske opreme v izbranem podjetju. **Predlagam uporabo DevOps pristopa in načina razmišljanja**, ki nudi večino odgovorov na težave in izzive, s katerimi se sooča izbrano podjetje:

Prakticiranje DevOps prakse zvezne integracije v večjem obsegu. Ob vsakodnevni integraciji sprememb predlagam tudi izdelavo avtomatiziranih postopkov, ki se bodo izvedli ob vsaki integraciji sprememb. Postopki naj preverijo izvedljivost ter pravilnost trenutnega stanja programske opreme. Rezultat avtomatiziranih postopkov naj bodo artefakti

programske opreme, pripravljeni za nadaljnjo obdelavo. Z obsežnejšim prakticiranjem zvezne integracije bi lahko izbrano podjetje:

- Zmanjšalo število napak v produkciji, saj avtomatizirani postopki identificirajo pomanjkljivosti, še preden se te dostavijo v končna okolja (Davis & Daniels, 2016).
- Zvišalo kvaliteto programske opreme in nivo zagotavljanja kakovosti, saj rezultati in posledice avtomatiziranih postopkov spodbujajo dodatno pregledovanje programske kode (Rahman & Roy, 2017).

Minimiziranje človeškega faktorja pri pripravi in izvedbi nadgradenj. Podlaga za uspešno prakticiranje DevOps je uporaba sodobnih orodij in tehnologij za avtomatizacijo opravil, ki so tradicionalno zahtevale človeško posredovanje. Predlagam, da se priprava nadgradenj za vse sloje programske opreme pripravi čimbolj samodejno ter brez dodatnega posredovanja razvijalcev. Prav tako predlagam, da podjetje vzpostavi kulturo, znotraj katere je prepovedano ročno izvajanje nadgradenj v končnih okoljih. Človeško napako je težje ponoviti kot napako znotraj avtomatiziranega postopka. Z minimiziranjem človeškega faktorja lahko izbrano podjetje poveča svojo zmogljivost ter zanesljiveje in hitreje dostavi vrednost svojim strankam, kar posledično veča njegovo poslovno vrednost (Humble & Farley, 2010).

Povečanje in pohitritev pretoka vrednosti do stranke s prakticiranjem DevOps prakse zvezne dostave. Predpogoj za prakticiranje zvezne dostave so ustrezno vzpostavljeni mehanizmi zvezne integracije. Ker se izbrano podjetje sooča s preredkimi prehodi v produkcijo, predlagam nadgradnjo avtomatiziranih postopkov v obliki cevovodov, ki bodo sposobni samodejno dostaviti artefakte programske opreme v različna okolja. Z avtomatizacijo ter visoko frekvenco izvajanja prehodov iz enega v drugo okolje lahko izbrano podjetje minimizira tveganja, ki nastanejo ob redkih ter ročnih posredovanjih razvijalcev (Kim, Debois, Willis & Humble, 2016). Z varnim, zanesljivim in hitrim načinom dostave programske opreme lahko izbrano podjetje tudi spodbudi svoje stranke, da bodo pogosteje izvajale prehode v produkciji. Ker vzpostavitev zvezne dostave v proces zahteva ustrezno podporo in napor, predlagam naslednjo strategijo uvajanja (Chen, 2016):

- Predstavitev vodstvu vseh težav, ki se jim lahko podjetje izogne s prakticiranjem zvezne dostave.
- Tvorjene namenske ekipe iz različnih oddelkov, ki bodo skrbeli za implementacijo zvezne dostave in komunikacijo z ostalimi člani oddelkov.
- Konstantno predstavljanje rezultatov uvajanja zvezne dostave ter omogočen dostop do cevovodov vsem razvijalcem.
- Za začetek uvajanja predlagam izbiro programske opreme, ki prinaša največ poslovne vrednosti podjetju, saj se lahko tako doseže največja podpora vodstva ter dodatni zagon ob prvih rezultatih.

Kot dodatne predloge za izboljšavo predlagam **opredelitev jasne strategije vejitve** repozitorija izvorne kode, zmožnost **ustvarjanja testnih okolij na zahtevo** ter imenovanje **skrbnika novega procesa** (navezava na predlog izboljšave iz prejšnjega poglavja).

Z upoštevanjem vseh opredeljenih predlogov za izboljšavo lahko izbrano podjetje izloči iz procesa vse aktivnosti, ki sem jih identificiral kot možne za odstranitev v analizi dodajanja vrednosti. Ker trenutna informacijska podpora, ki jo podjetja uporablja (sistem za podporo strankam in sistem za vodenje sprememb), ni več primerna za predlagan način izvedbe procesa, priporočam uporabo orodja Azure DevOps.

4.4 Zagotavljanje kakovosti programske opreme

Zavezanost h kakovosti svojega produkta je eden izmed ključnih dejavnikov, ki ločuje zelo uspešno podjetja od manj uspešnih (Peters & Waterman, 1982). V primeru programske opreme obstaja mnogo različnih načinov, s katerimi lahko razvojno podjetje upravlja kakovost (Sommerville, 2016). V nadaljevanju najprej analiziram, kakšno je trenutno stanje v izbranem podjetju na področju upravljanja kakovosti programske opreme, nato identificiram težave in pomanjkljivosti ter opredelim možnosti za izboljšave.

4.4.1 Analiza sedanjega stanja

Na podlagi analize procesa razvoja in vzdrževanja ter izvedbe neformalnih pogovorov sem identificiral naslednjih pet aktivnosti, ki jih izbrano podjetje izvaja za zagotavljanje kakovosti lastne programske opreme:

- *Ročno testiranje*. Seznam ključnih aktivnosti za zagotavljanje kakovosti programske opreme v izbranem podjetju. Ročno testiranje se izvaja znotraj aktivnosti *[8] Testiraj spremembe programske opreme* in *[12] Testiraj načrtovano informacijsko podporo* v procesu razvoja ter aktivnosti *[10] Testiraj spremembo programske opreme*, *[15] Preveri nadgradnjo podatkovnega modela* in *[20] Testiraj spremembo programske opreme* v procesu vzdrževanja programske opreme.
- *Zbiranje napak iz končnih okolij*. Izbrano podjetje ima vzpostavljene kanale, preko katerih pridobiva povratne informacije ob različnih dogodkih. Te informacije se uporabijo za reševanje javljenih problemov.
- *Vodenje sprememb programske opreme*. Sistem za vodenje sprememb nudi izbranemu podjetju informacijsko podporo za izvajanje procesa pri zahtevkih za spremembo programske opreme. Tako ima podjetje pregled in sled prehodov statusov za sklop posameznih sprememb. Iz sistema je razvidno, kdo in kdaj je vnesel zahtevek za spremembo, kdo in kdaj ga je potrdil ter kdaj je bil vključen v distribucijo.
- *Izvajanje izobraževanj za zaposlene.* Izbrano podjetje izvaja interna ter nudi eksterna izobraževanja za svoje zaposlene. Na izobraževanjih razvijalci pridobivajo nova znanja na področju zagotavljanja kakovosti programske opreme.

- *Upravljanje varnosti osebnih podatkov.* DBA skrbi za ustrezno varovanje osebnih podatkov znotraj predpisanih zakonskih okvirjev.

Glavna aktivnost za zagotavljanje kakovosti je torej ročno testiranje. Večinoma ga izvaja razvijalec. Ker razvijalci praviloma skrbijo vsak za svoj del programske opreme, tudi sami ročno testirajo implementirane spremembe funkcionalnosti. To je tudi prva težava, ki sem jo analiziral in identificiral. Premalo je navzkrižnega testiranja oziroma načina dela, kjer bi vsako spremembo programske opreme ročno preverjala oseba, ki ni njen avtor. V nadaljevanju povzemam še ostale težave in pomanjkljivosti, ki sem jih identificiral pri analizi sedanjega stanja:

- V fazi implementacije programske opreme se ne upošteva načelo štirih oči (angl. four eyes principle). Kadar programsko kodo pregleduje samo njen avtor, je njena kakovost dokazano slabša kot pa kakovost programske opreme, ki jo je pregledala druga oseba (Bavota & Russo, 2015). To je posledica že večkrat omenjene individualnosti v podjetju ter posledica načina organiziranosti ob izvedbi novih projektov. Se pa načelo štirih oči upošteva pri izvajanju aktivnosti *[15] Preveri nadgradnjo podatkovnega modela* v procesu vzdrževanja programske opreme, kjer DBA pregleda in preveri pripravljene nadgradnje podatkovnih zbirk. Vendar ta pregled ni obvezen, saj se običajno izvede samo, kadar obstajajo težave pri izvedbi nadgradnje. Prav tako se ne pregleduje celotna izvorna koda, temveč le del, ki bo nadgradil programsko opremo v končnem okolju.
- Posledica neprakticiranja načela štirih oči je opazna razlika v strukturi, razumljivosti in kompleksnosti izvorne kode med njenimi avtorji. Takšne razlike znižujejo kvaliteto programske opreme in zvišujejo stroške vzdrževanja (Sommerville, 2016). Dodatni vzrok je tudi pomanjkanje internih standardov programiranja, ki bi služili kot okvir, znotraj katerega bi implementacija potekala na enak način, ne glede na projekt. Ker standardov ni, se povečujejo čas in stroški vzdrževanja programske opreme, še posebej v primeru, kadar nov zaposleni postane odgovoren za del programske opreme, katerega avtorja ni več v podjetju.
- Ob pomanjkanju navzkrižnega testiranja sem identificiral tudi pomanjkanje testiranja s strani ostalih vlog v procesu, predvsem svetovalcev in oseb, odgovorih za vsebino na strani stranke. Prav tako manjka ustrezna informacijska podpora za ročno testiranje, namreč večina povratnih informacij ob ročnem testiranju se vodi v tekstovnih dokumentih (kadar je to ena izmed zahtev znotraj projekta), v sistemu za podporo strankam ali v e-poštnih sporočilih. Manjka možnost upravljanja povezav med spremembo programske opreme ter povratnimi informacijami ob ročnem testiranju te spremembe.

4.4.2 Predlogi za izboljšave

Vzpostavitev internih standardov programiranja. Predlagam sestavo in zapis internih pravil pri implementaciji programske opreme, ki zajemajo najvišjo skupno modrost, ki jo

ima izbrano podjetje. Interni standardi naj temeljijo na najprimernejših praksah, ki izvirajo iz preteklih izkušenj reševanja težav in napak. Z vzpostavitvijo standardov lahko izbrano podjetje postavi okvir, ki določa merila za merjenje kakovosti programske opreme v izbranem podjetju. Kadar pride do nasprotij v fazi implementacije, lahko interni standardi služijo kot podlaga za odločitev, ali je bil dosežen zahtevan nivo kakovosti. Z vzpostavitvijo internih standardov programiranja lahko izbrano podjetje:

- Ohrani konstanten tok dela, kadar določeno opravilo prevzame druga oseba. Če vsi dosledno upoštevajo predpisane standarde, se lahko zmanjšajo stroški ob zamenjavi osebja (Sommerville, 2016).
- Zviša berljivost in razumljivost programske kode. Dobro definirani standardi vodijo k samodokumentirani izvorni kodi (Pressman, 2010), kar je ključno za razumevanje namena in konteksta posameznega dela programske opreme. Oboje znižuje stroške vzdrževanja (Lee, Lee & In, 2015).
- Skrajša čas izvajanja pregledovanja programske kode, saj vzpostavljeni standardi zmanjšajo potrebo po dodatnih razlagah in utemeljitvah načina izvedbe posamezne rešitve (Ebert, Castor, Novielli & Serebrenik, 2019).

Redno prakticiranje neformalnega pregledovanja programske kode. Predlagam, da vsako spremembo programske opreme pregleda vsaj še en razvijalec. Pregled se lahko izvede kot preprost pogovor med dvema zaposlenima glede uporabljenih rešitev za implementacijo določene funkcionalnosti. Ob dokazano višji kakovosti programske opreme lahko izbrano podjetje tudi:

- Zelo zmanjša možnost za nastanek napak v končnem produkcijskem okolju (McConnell, 2004) ter bistveno poveča berljivost programske kode (Bavota & Russo, 2015).
- Zmanjša prisotnost individualnosti, saj razvijalci bolj sodelujejo med sabo in spoznavajo druge dele obstoječe programske opreme (El Asri, Kerzazi, Uddin, Khomh & Idrissi, 2019).
- Izboljša prenos kolektivnega spomina, saj se med pregledovanjem izmenjuje znanje med posamezniki, ki imajo različne strokovne spodobnosti (Spohrer, Kude, Schmidt & Heinzl, 2016).
- Uravnoteženo porazdeli delo med zaposlene. Če izkušenejši razvijalci nimajo časa za razvoj določene težje funkcionalnosti, lahko sodelujejo samo kot pregledovalci, delo pa lahko prevzamejo manj izkušeni razvijalci, ki bodo skozi izvajanje pregleda pridobili potrebna strokovna znanja (McIntosh, Kamei, Adams & Hassan, 2016).

Prav tako predlagam uporabo boljše in enovite informacijske podpore ročnemu testiranju programske opreme. Orodje Azure DevOps, ki nudi celostno podporo izvajanju DevOps, vsebuje tudi orodje Azure Test Plans, s katerim lahko tester ob testiranju enostavno zajame vse potrebne informacije v trenutku ugotovitve določene pomanjkljivosti programske opreme.

SKLEP

V magistrskem delu raziskujem, kako lahko izbrano podjetje izboljša proces razvoja in vzdrževanja programske opreme ter kako lahko izboljša zagotavljanje kakovosti programske opreme. Odgovor na raziskovalni vprašanji prispeva k razumevanju specifik in možnosti izboljšav razvoja, vzdrževanja in zagotavljanja kakovosti programske opreme v kontekstu manjšega razvojnega podjetja.

Na podlagi analize literature na tem področju in analize sedanjega stanja razvoja, vzdrževanja in zagotavljanja kakovosti v izbranem podjetju nudim naslednje glavne ugotovitve ter možnosti za izboljšave:

- Za poenotenje načina dela predlagam uvedbo skrbnika temeljnega poslovnega procesa.
- Za zmanjšanje individualnosti predlagam prakticiranje XP-tehnike programiranja v paru.
- Za izboljšanje razumevanja vsebine zahtev v fazah analize in načrtovanja predlagam uporabo XP-tehnike uporabniških zgodb in BPMN-vizualizacij poslovnih procesov.
- Za izboljšanje produktivnosti razvijalcev in končne kakovosti programske opreme predlagam prakticiranje XP-tehnike prestrukturiranja programske kode.
- Za izboljšanje modularnosti programske opreme in minimiziranje tehniškega dolga predlagam implementacijo novega sistema za informacijsko podporo različnim poslovnim procesom na podlagi predstavljenega koncepta.
- Za učinkovitejšo operativno delo predlagam uvedbo DevOps kulture in spremembo načina razmišljanja glede razvoja in vzdrževanja programske opreme.
- Za minimiziranje števila napak v produkciji in povečanje kakovosti programske opreme predlagam prakticiranje DevOps prakse zvezne integracije v polnem obsegu.
- Za povečanje pretoka vrednosti do stranke in minimiziranje težav ob prehodih v produkcijo predlagam prakticiranje DevOps prakse zvezne dostave.
- Za povečanje končne kvalitete programske opreme predlagam vzpostavitev internih standardov programiranja in redno izvajanje neformalnega pregleda programske kode.

Z uporabo naštetih predlogov lahko izbrano podjetje, kot tudi katerokoli drugo razvojno podjetje, ki deluje v podobnem kontekstu, zniža stroške vzdrževanja programske opreme in poveča svojo poslovno vrednost.

LITERATURA IN VIRI

1. Arisholm, E., Gallis, H., Dybå, T. & Sjøberg, D. (2007). Evaluating Pair Programming with Respect to System Complexity and Programmer Expertise. *IEEE Transactions on Software Engineering, 33*(2), 65-86.
2. Bavota, G. & Russo, B. (2015). Four Eyes are Better than Two: On the Impact of Code Reviews on Software Quality. *Proceedings of the 31st International Conference on*

Software Maintenance and Evolution (str. 81-90). Bremen: Institute of Electrical and Electronics Engineers.

3. Beck, K. (2004). *Extreme Programming Explained: Embrace Change* (2. izd.). Boston: Addison-Wesley.
4. Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., . . . Sutherland, J. (2001). *The Agile Manifesto*. Pridobljeno 22. septembra 2021 iz http://agilemanifesto.org
5. Bessin, G. (2004). IBM. *The Business Value of Quality*. Pridobljeno 22. septembra 2021 iz https://www.ibm.com/developerworks/rational/library/4995.html
6. Brooks, F. (1995). *The Mythical Man-Month: Essays on Software Engineering* (3. izd.). Boston: Addison-Wesley.
7. Chen, J., Xiao, J., Wang, Q., Osterweil, L. & Li, M. (2014). Refactoring Planning and Practice in Agile Software. *Proceedings of the International Conference on Software and Systems Process* (str. 55-64). Nanjing: Association for Computing Machinery.
8. Chen, L. (2016). Continuous Delivery: Overcoming Adoption Obstacles. *Proceedings of the International Workshop on Continuous Software Evolution and Delivery* (str. 84-84). Austin: iEEE.
9. Cobb, C. (2011). *Making Sense of Agile Project Management: Balancing Control and Agility.* Hoboken: Wiley.
10. Cobb, C. (2015). *The Project Manager's Guide to Mastering Agile: Principles and Practices for an Adaptive Approach.* Hoboken: Wiley.
11. Crookshanks, E. (2015). *Practical Enterprise Software Development Techniques: Tools and Techniques for Large Scale Solutions* (1. izd.). New York: Apress.
12. Dalpiaz, F. & Brinkkemper, S. (2018). Agile Requirements Engineering with User Stories. *Proceedings of the 26th International Requirements Engineering Conference* (str. 506-507). Banff: Institute of Electrical and Electronics Engineers.
13. Davidsen, M. & Krogstie, J. (2010). A Longitudinal Study of Development and Maintenance. *Information and Software Technology, 52*(7), 707–719.
14. Davis, J. & Daniels, R. (2016). *Effective DevOps: Building a Culture of Collaboration, Affinity, and Tooling at Scale.* Sebastopol: O'Reilly Media, Inc.
15. DeMarco, T. & Lister, T. (2013). *Peopleware: Productive Projects and Teams* (3 izd.). Boston: Addison-Wesley.
16. Diebold, P., Theobald, S., Wahl, J. & Rausch, Y. (2018). An agile transition starting with user stories, DoD & DoR. *Proceedings of the International Conference on Software and System Process* (str. 147-156). Gothenburg: Association for Computing Machinery.
17. Ebert, F., Castor, F., Novielli, N. & Serebrenik, A. (2019). Confusion in Code Reviews: Reasons, Impacts, and Coping Strategies. *Proceedings of the 26th International Conference on Software Analysis, Evolution, and Reengineering* (str. 49-60). Hangzhou: Institute of Electrical and Electronics Engineers.

18. El Asri, I., Kerzazi, N., Uddin, G., Khomh, F. & Idrissi, M. (2019). An Empirical Study of Sentiments in Code Reviews. *Information and Software Technology, 114*, 37-54.

19. Erlikh, L. (2000). Leveraging Legacy System Dollars for E-Business. *IT Professional, 2*(3), 17-23.

20. Fagan, M. (1986). Advances in Software Inspections. *IEEE Transactions on Software Engineering, 12*(7), 744-751.

21. Fang, X. (2001). Using a Coding Standard to Improve Program Quality. *Proceedings of the 2nd Asia-Pacific Conference on Quality Software* (str. 73-78). Hong Kong: Institute of Electrical and Electronics Engineers.

22. Fatima, N., Nazir, S. & Chuprat, S. (2020). Software Engineering Wastes: A Perspective of Modern Code Review. *Proceedings of the 3rd International Conference on Software Engineering and Information Management* (str. 93–99). Sydney: Association for Computing Machinery.

23. Gartner. (2020). *Gartner Says Global IT Spending to Decline 8% in 2020 Due to Impact of COVID-19.* Pridobljeno 22. septembra 2021 iz https://www.gartner.com/en/newsroom/press-releases/2020-05-13-gartner-says-global-it-spending-to-decline-8-percent-in-2020-due-to-impact-of-covid19

24. Garvin, D. (1987). Competing on the Eight Dimensions of Quality. *Harvard Business Review*, 101–109.

25. Heričko, M., Živkovič, A. & Rozman, I. (2008). An Approach to Optimizing Software Development Team Size. *Information Processing Letters, 108*(3), 101-106.

26. Herraiz, I., Rodriguez, D., Robles, G. & Gonzalez-Barahona, J. (2013). The Evolution of the Laws of Software Evolution: A Discussion Based on a Systematic Literature Review. *ACM Computing Surveys, 46*(2), 28:1-28:28.

27. Horch, J. (2003). *Practical Guide to Software Quality Management.* Norwood: Artech House.

28. Humble, J. & Farley, D. (2010). *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation.* Boston: Addison-Wesley.

29. Jarke, M. & Prause, C. (2015). Gamification for Enforcing Coding Conventions. *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering* (str. 649-660). Bonn: Association for Computing Machinery.

30. Jones, C. (2006). *The Economics of Software Maintenance in the 21st Century, Version 3.* https://spr.com: Capers Jones.

31. Kemerer, C. & Paulk, M. (2009). The Impact of Design and Code Reviews on Software Quality: An Empirical Study Based on PSP Data. *IEEE Transactions on Software Engineering*, 534-550.

32. Keshta, N. & Morgan, Y. (2017). Comparison between traditional plan-based and agile software processes according to team size & project domain (A systematic literature

review). *Proceedings of the 8th Annual Information Technology, Electronics and Mobile Communication Conference* (str. 567-575). Vancouver: nstitute of Electrical and Electronics Engineers.

33. Kim, G., Behr, K. & Spafford, G. (2013). *The Phoenix Project: A Novel about IT, DevOps, and Helping Your Business Win.* Portland: IT Revolution Press.

34. Kim, G., Debois, P., Willis, J. & Humble, J. (2016). *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations.* Portland: IT Revolution Press.

35. Kovačič, A. (2019). PRENOVA IN INFORMATIZACIJA POSLOVANJA: Strateška izhodišča, prenova in informatizacija poslovnih procesov [prosojnice, stran 36]. Ljubljana: Ekonomska fakulteta.

36. Lee, T., Lee, J. & In, H. (2015). Effect analysis of coding convention violations on readability of post-delivered code. *IEICE Transactions on Information and Systems, E98D*(7), 1286-1296.

37. Lehman, M. (1996). Laws of Software Evolution Revisited. *Proceedings of the 5th European Workshop on Software Process Technology* (str. 108-124). Berlin Heidelberg: Springer-Verlag.

38. Lehman, M. & Ramil, J. (2001). An Approach to a Theory of Software Evolution. *Proceedings of the 4th International Workshop on Principles of Software Evolution* (str. 70-74). New York: Association for Computing Machinery.

39. Lientz, B. & Swanson, B. (1980). *Software Maintenance Management.* Boston: Addison-Wesley.

40. Liskin, O., Pham, R., Kiesling, S. & Schneider, K. (2014). Why we need a granularity concept for user stories. *Proceedings of the 15th International Conference on Agile Software Development* (str. 110-125). Rome: Springer Verlag.

41. Lucassen, G., Dalpiaz, F., van der Werf, J. & Brinkkemper, S. (2016). The use and effectiveness of user stories in practice. *Proceedings of the 22nd International Working Conference on Requirements Engineering: Foundation for Software Quality* (str. 205-222). Gothenburg: Springer Verlag.

42. Martini, A., Sikander, E. & Madlani, N. (2018). A semi-automated framework for the identification and estimation of Architectural Technical Debt: A comparative case-study on the modularization of a software component. *Information and Software Technology, 93*, 264-279.

43. McCall, J., Richards, P. & Walters, G. (1977). Factors in Software Quality. *RADC-TR-77-369*. US Department of Commerce.

44. McConnell, M. (2004). *Code Complete: A Practical Handbook of Software Construction* (2. izd.). Seatle: Microsoft Press.

45. McIntosh, S., Kamei, Y., Adams, B. & Hassan, A. (2016). An Empirical Study of the Impact of Modern Code Review Practices on Software Quality. *Empirical Software Engineering, 21*(5), 2146-2189.

46. Moser, R., Abrahamsson, P., Pedrycz, W., Sillitt, A. & Succi, G. (2008). A Case Study on the Impact of Refactoring on Quality and Productivity in an Agile Team. *Proceedings of the 2nd IFIP TC 2 Central and East European Conference on Software Engineering Techniques,* (str. 252-266). Poznan.

47. Pendharkar, P. & Rodger, J. (2009). The Relationship Between Software Development Team Size and Software Development Cost. *Communications of the ACM, 52*(1), 141-144.

48. Peters, T. & Waterman, R. (1982). *In Search of Excellence.* New York: Harper & Row.

49. Plonka, L., Sharp, H., Van Der Linden, J. & Dittrich, Y. (2015). Knowledge transfer in pair programming: An in-depth analysis. *International Journal of Human Computer Studies*, 66-78.

50. Popic, S., Velikic, G., Jaroslav, H., Spasic, Z. & Vulic, M. (2018). The Benefits of the Coding Standards Enforcement and it's Influence on the Developers' Coding Behaviour: A Case Study on Two Small Projects. *Proceedings of the 26th Telecommunications Forum.* Belgrade: Institute of Electrical and Electronics Engineers.

51. Pressman, R. (2010). *Software Engineering: A Practitioner's Approach* (7. izd.). New York: McGraw-Hill.

52. Rahman, M. & Roy, C. (2017). Impact of Continuous Integration on Code Reviews. *Proceedings of the 14th International Conference on Mining Software Repositories.* Buenos Aires: IEEE.

53. Razzaq, S., Huang, J., Sun, H. & Xie, M. (2019). Analyzing time pressure for software economics: Empirically evaluating team factors as the strategic criteria. *Journal of Enterprise Information Management, 32*(2).

54. Rodríguez, D., Sicilia, M., García, E. & Harrison, R. (2012). Empirical Findings on Team Size and Productivity in Software Development. *Journal of Systems and Software, 85*(3), 562–570.

55. Sharma, S. (2017). *The DevOps Adoption Playbook: A Guide to Adopting DevOps in a Multi-Speed IT Enterprise.* Indianapolis: Wiley.

56. Sommerville, I. (2016). *Software Engineering* (10. izd.). Harlow: Pearson Education.

57. Soriano, J. (2012). *Maximizing Benefits from IT Project Management: From Requirements to Value Delivery.* Boca Raton: CRC Press.

58. Spohrer, K., Kude, T., Schmidt, C. & Heinzl, A. (2016). The Transactive Processes of Social Coding: How Code Review Substitutes for Transactive Memory in Software Development Teams. *Proceedings of the International Conference on Information Systems.* Dublin: Association for Information Systems.

59. Ståhl, D., Mårtensson, T. & Bosch, J. (2017). The continuity of continuous integration: Correlations and consequences. *Journal of Systems and Software, 127*, 150-167.

60. Tessem, B. (2014). Individual empowerment of agile and non-agile software developers in small teams. *Information and Software Technology, 56*(8), 873-889.

61. Testa, L. (2009). *Growing Software: Big Strategies for Managing Small Software Companies.* San Francisco: No Starch Press.

62. Trkman, M., Mendling, J. & Krisper, M. (2016). Using business process models to better understand the dependencies. *Information and Software Technology, 71*(1), 58-76.

63. van Vliet, H. (2007). *Software Engineering: Principles and Practice* (3. izd.). Hoboken: Wiley.

64. Weinberg, G. (1971). *The Psychology of Computer Programming.* New York: Van Nostrand.

65. Williams, L., Kessler, R., Cunningham, W. & Jeffries, R. (2000). Strengthening the Case for Pair Programming. *IEEE Software, 17*(4), 19-25.