Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Primož Ocepek

Izdelava projektno prilagodljive in agilne metodologije razvoja programskih rešitev

magistrsko delo

študijski program druge stopnje
pedagoško računalništvo in informatika

Mentor: prof. dr. Marko Bajec

Ljubljana, 2018

Zahvala

Zahvaljujem se mentorju prof. dr. Marku Bajcu za strokovno svetovanje, potrpežljivost in za spodbudo pri nastajanju magistrskega dela.

Zahvaljujem se tudi podjetju Dewesoft pod vodstvom dr. Jureta Kneza in Andreja Orožna, da so mi omogočili opravljanje magistrskega dela in mi pri tem pomagali. Zahvaljujem se tudi Tilnu Sotlerju, ki mi je predstavil teoretično ozadje, me spoznal s kakovostnim programiranjem in bil delovni mentor v podjetju Dewesoft.

Še posebej pa se zahvaljujem svoji družini, mami Mojci in očetu Jožetu, za skrb in motivacijo ter za vso podporo med študijem. Zahvaljujem se bratu Urošu, ki je vedno z velikim veseljem pomagal in mi svetoval tudi v najtežjih trenutkih.

Doc. dr. Tomažu Petku in Simoni Izgoršek se najlepše zahvaljujem za lektoriranje slovenskega in angleškega besedila.

Primož Ocepek, 2018

みぬがはな. (Domišljija je močnejša od realnosti.)

— Japonski pregovor

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
IEC	International Electrotechnical Commission	Mednarodna komisija za elektrotehniko
ISO	International Organization for Standardization	Mednarodna organizacija za standardizacijo
ME	Method engineering	Razvoj metodologij
SME	Situational method engineering	Situacijski pristop razvoja metodologij

Povzetek

Naslov: Izdelava projektno prilagodljive in agilne metodologije razvoja programskih rešitev

Zaradi potreb po vse hitrejšem razvoju programskih rešitev in zmanjševanju tveganj, da razvita programska oprema ne bi ustrezala končnim uporabnikom, so že pred več leti slapovni razvoj nadomestile lahke in agilne metodologije, ki sledijo principom evolucijskega razvoja. Na voljo so številne agilne metodologije. Med najbolj poznanimi so: Scrum, Kanban, FDD, XP, Crystal . . . . Njihova uvedba v prakso pa je še vedno izziv, saj je način uvajanja zelo odvisen od več lastnosti, kot so: kultura podjetja na področju razvoja programske opreme, kompetence posameznikov, vrste projektov, ki jih podjetje izvaja, itn. V raziskovalni sferi se je za namen uvajanja prilagodljivih in agilnih metodologij uveljavil tako imenovani situacijski pristop razvoja metodologij (v nadaljevanju: SME), ki izhaja iz karakteristik podjetja in projektov ter predlaga korake uvedbe primerne metodologije. V praksi je iz izkušenj z uporabo te metode sorazmerno malo, saj zahteva poglobljen pristop in sodelovanje različnih strokovnih profilov. Cilj magistrskega dela je podrobno preučiti principe SME, jih po potrebi prilagoditi in uporabiti za razvoj konkretne agilne metodologije v podjetju.

Ključne besede

Agilni razvoj programske opreme, Scrum, Kanban, Testno usmerjeni razvoj, Situacijski pristop razvoja metodologij

Abstract

Title: Implementation of a project-adaptable and agile software development methodology

New methodologies of software development were adjusted and implemented because of the needs for faster software development and to limit the risks for unfulfilled goals. Waterfall as a methodology was replaced by the agile methodologies. These follow the principle of evolutional development. There are many agile methodologies. Most known methodologies are Scrum, Kanban, FDD, XP, Crystal ... Their implementation into company is still a challenge because it depends on many factors like: culture of the company in developing software, skills of the employees, types of projects developed by the company and many other factors. In academics for implementing agile methodologies a new approach was defined. The aproach is called situational method engineering. This approach uses all the charasterictics of the company and suggests aproaches that are taken from agile methodologies. There is a lack of this aproach in academics because it needs deep understanding and coorperation with different experts. The goal of the masters thesis is do very precise analyse of the SME, customize and use it for developing an agile methodology for company that develops software.

Keywords

Agile software development, Scrum, Kanban, Test Driven Development, Situational Method Engineering

Poglavje 1

Uvod

1.1 Opis problema in rešitve

Pristopi in načini razvijanja programske opreme so se skozi čas spreminjali. Do sedemdesetih let dvajsetega stoletja so razvijalci razvijali programsko opremo večinoma brez uporabe sistematičnih pristopov. To obdobje je po avtorju Avisonu poimenovano kot obdobje pred pojavom metodologij. Razvijanje je potekalo po principu »čez palec« [1, 2]. Temu obdobju je sledilo zgodnje obdobje metodologij. V tem obdobju so se pojavile prve formalne metodologije. Te so definirale delitve razvoja na posamezne stopnje ali postopke. Poznani primer formalne metodologije je slapovni model [1, 2]. Slapovni model definira zaporedne faze življenjskega cikla razvoja projekta. Obdobju metodologij, ki so definirale zaporedne faze, je sledilo obdobje metodologij, ki uporabljajo iterativne in inkrementalne modele. S takšnimi modeli so se metodologije lažje prilagodile spremembam zahtev strank in uporabnikov [1, 2]. S prilagajanjem na spremembe so metodologije postale obsežnejše in zahtevnejše za razumevanje [1, 2]. Raven zahtevnosti se je povečevala zaradi večjega števila faz, postopkov in aktivnosti, ki se odražajo v dodatnem porabljenem času. V obdobju konca devetdesetih let dvajsetega stoletja se je razširila kritika nad metodologijami, ki so pretirano definirane. Takšne metodologije so si pridobile ime težke metodologije. Teža metodologije je definirana kot zmnožek obsega in gostote metodologije [1]. Obseg metodologije predstavlja število

elementov v metodologiji. Gostota pa predstavlja raven podrobnosti. Ta predstavlja podrobnosti, kako natančno definiramo aktivnosti, vloge, izdelke, priporočila in druge elemente metodologije. Kot nasprotje težkim metodologijam so nastale lahke metodologije. Lahke metodologije v ospredje dajejo implementacijo, testiranje in sprotno prilagajanje zahtevam projekta [1]. Nekatere lahke metodologije lahko označimo kot agilne. Metodologija je agilna, če omogoča hitro prilagajanje obsega metodologije glede na zahteve potreb projekta [1]. Leta 2001 so se avtorji agilnih metodologij sešli in napisali manifest agilnega razvoja programske opreme. V njem so definirali štiri načela in priporočila, ki izhajajo iz teh načel. Načela predstavljajo osnovo agilnih metodologij. Načela agilnega manifesta so [3]:

- Posamezniki in njihova komunikacija so pomembnejši kot sam proces in orodja.
- Delujoča programska oprema je pomembnejša kot popolna dokumentacija.
- Vključevanje (sodelovanje) uporabnika je pomembnejše kot pogajanje na osnovi pogodb.
- Upoštevanje sprememb je pomembnejše od sledenja načrtu.

Priporočila pa predstavljajo pomoč pri gradnji in ovrednotenju metodologij [2]. Ker so si podjetja med seboj različna in se ukvarjajo z različnimi projekti, so raziskovalci odkrili, da ne obstaja metodologija, ki ustreza vsem podjetjem ali projektom. Kot odgovor na te odzive je nastal situacijski pristop razvoja metodologij (Angl. "Situational method engineering") [4]. Namen te metode je vzeti eno ali več metodologij in jih prilagoditi situaciji v podjetju ali celo posameznemu projektu. Iz metodologije so lahko izbrani le posamezni deli, ki so za preučevano podjetje primerni, hkrati pa so vključene tudi navade razvijalcev v podjetju [5]. Glede na raven zrnatosti lahko metodologijo razdrobimo na metode, ali pa na precej manjše dele. Če metodologijo razbijemo na metode, imamo grobo zrnatost. Če metodo razbijemo na precej manjše dele, pa govorimo o fini zrnatosti.
Namen situacijskega pristopa razvoja metodologij je izboljšava procesa razvoja programske opreme v podjetju ali oblikovanje nove metodologije, ki bo prilagojena

potrebam konkretnega projekta [4]. V okviru magistrskega dela bomo uporabili situacijski pristop razvoja metodologij za potrebe izboljšave procesa razvoja v podjetju, v katerem sem zaposlen. Trenutni postopki razvoja so namreč pomanjkljivi in neoptimalni. V podjetju so se odločili za agilne metodologije zaradi prilagodljivosti, ki jih metodologije omogočajo. Implementacija agilnih metodologij s pomočjo situacijskega pristopa razvoja metodologij bo potekala postopoma. Prvi korak je njihova analiza, s čimer želimo prepoznati posamezne segmente ali dele obstoječih metodologij, ki bi bile lahko bile v pomoč pri izboljševanju obstoječega procesa razvoja v podjetju. Na koncu prvega koraka bomo oblikovali repozitorij posameznih metod. Drugi korak je ugotovitev trenutnega stanja v podjetju. To se bo ugotovilo s pomočjo intervjuja dveh zaposlenih v podjetju in z anketiranjem razvijalcev. S tem bomo dobili prvi vpogled v podjetje in zaznali, s katerimi problemi se podjetje spoprijema. Tretji korak bo določitev primernih metod za izboljšavo procesa razvoja v preučevanem podjetju. Metode bodo izbrane iz nabora, ki ga bomo definirali v prvem koraku. Te metode bodo pomagale rešiti probleme v podjetju. Četrti korak bo oblikovanje nove metodologije z vključitvijo teh metod.

1.2 Pregled sestave dela

Magistrsko delo je sestavljeno iz dveh delov. V prvem delu predstavimo, kaj so: proces, metodologija, metoda. Po opredelitvi procesa, metodologije in metode sledi predstavitev situacijskega pristopa razvoja metodologij, ki je v nadaljevanju označen s SME. Predstavljene so posamezne komponente SME in načini uporabe SME. Po opisu situacijskega pristopa razvoja metodologij sledijo opisi metodologij. Opisani so: RUP, Scrum, ekstremno programiranje, Kanban, model Kaos. Po opisu metodologij sledi definicija metod, ki bodo shranjene v repozitoriju posameznih metod. V drugem delu sta predstavljena podjetje in uporaba situacijskega pristopa na izbranem podjetju. Najprej so predstavljene ugotovitve analize trenutnega stanja. Podatki so bili zbrani z razgovori in anketiranjem zaposlenih. Iz intervjuja in anket je razvidno, s katerimi problemi se spoprijema podjetje. Sledi opis uporabe SME za potrebe izboljšave procesa razvoja s poudarkom na delih

procesa, ki so predhodno identificirani kot slabo podprti oziroma neoptimizirani.

1.3 Cilj

Cilj magistrskega dela je preučiti teoretični pristop k izboljševanju in/ali razvoju novih metodologij razvoja programske opreme, ki je v teoriji poznan kot situacijski razvoj metodologij, ter ga aplicirati na konkretnem primeru razvojnega podjetja.

Poglavje 2

Teoretična predstavitev

2.1 Proces razvoja programske opreme, metodologija in metoda

Pred definiranjem situacijskega razvoja metodologije je treba definirati, kaj so proces, metodologija in metoda, saj predstavljajo pomembne elemente v SME. Sledi opis procesa, metodologije in nazadnje tudi metode.

2.1.1 Opis procesa razvoja programske opreme, metodologije in metode

Proces se po Weitzerju ukvarja bolj z razvojnimi pogledi in manj s tehničnim pogledom razvoja programske opreme [2]. Prav tako Weitzer definira namen procesov, ki so podpora projektnemu vodstvu, in obravnava poslovne vidike razvoja programske opreme [2]. Avtor Weitzer po Avisonu povzame definicijo metodologije za razvoj informacijskih sistemov kot zbirko postopkov, tehnik, orodij in izdelkov, ki razvijalcem pomagajo pri razvoju novega informacijskega sistema [2]. Metodologija je po SSKJ-ju definirana kot [6]: skupek metod, ki se uporabljajo pri nekem raziskovanju oz. mišljenju. Po iSlovarju [7] je definirana kot zbirka metod, postopkov in standardov, ki sestavljajo zaključeno celoto inženirskih pristopov k razvoju izdelka. Po avtorju Jayaratna [8] pa metodologija predstavlja študijo o metodah.

Iz teh definicij lahko po Weitzerju povzamemo, da je proces bolj ozko opredeljen in neposredno povezan z razvojem programske opreme. Tako lahko rečemo, da je proces del metodologije, saj metodologija ne vsebuje le procesov, ampak tudi filozofijo in kulturo [2]. Če pogledamo definicije, ki jih predlagajo SSKJ in iSlovar ter Jayaranta, vidimo, da obstaja povezava med metodami in metodologijami. Metoda je v SSKJ-ju definirana kot [9] oblika načrtnega, premišljenega dejanja, ravnanja ali mišljenja za dosego nekega cilja. Po iSlovarju [7] pa predstavlja postopke in pravila za izvajanje določene naloge. Po Brinkkemperju [10] je metoda pristop, po katerem izvajamo projekt, ki temelji na razmišljanju, navodilih, pravilih, hevristikah in je strukturiran na način, da definira aktivnosti, izdelke in vloge. Iz teh razlag terminov lahko predpostavimo, da metodologija predstavlja postopke oziroma načine, kako se spoprijemati z eno ali več zadevami/težavami/aktivnostmi. Ti postopki ali načini predstavljajo metode, ki dejansko predpišejo, kako poteka posamezna izvedba. Metoda pa predstavlja navodila/napotke, ki natačno definirajo, kako izvajati neko aktivnost/zadevo oz. kako rešiti neki problem. Henderson [4] enači pomen metode in metodologije, ker sta si definiciji zelo podobni. V našem magistrskem delu bomo razlikovali med metodami in metodologijami na način, da je metodologija sestavljena iz ene ali več metod. Seveda pa metodologija ni sestavljena le iz metod, ampak vključuje tudi druge elemente, kot so npr. vrednote, filozofija, orodja, standardi, izkušnje, tehnike itn.

2.2 Situacijski pristop razvoja metodologij

Situacijski pristop razvoja metodologij (Angl. "Situational method engineering"), označen s kratico SME, je pristop za oblikovanje metodologije ali posamezne metode glede na posamezno situacijo v podjetju [4]. Metodologija je lahko oblikovana na osnovi obstoječih metodologij ali na temeljih obstoječih procesov, ki se izvajajo v podjetju. Nekatere obstoječe metodologije so opisane v poglavju 2.3. Razvoj SME-a se je začel s spoznanjem, da metodologije ne delujejo po principu »*One size fits all*«. To pomeni, da ne obstaja metodologija, ki bi ustrezala vsem podjetjem niti vsem projektom istega podjetja. Podjetja so si med seboj različna glede

na kulturo dela, število zaposlenih, hierarhijo zaposlenih, način razvoja programske opreme itn. SME pri razvoju metodologije upošteva ta merila in prilagodi metodologijo, tako da ustreza podjetju. Postopek SME je razdeljen na več delov. Na začetku moramo pridobiti repozitorij posameznih metod. Napolnimo ga lahko z metodami, ki lahko izhajajo iz metodologij ali delovnih praks. Repozitorij lahko pridobimo iz drugih virov. Iz napolnjenega repozitorija posameznih metod lahko sestavimo osnovno metodologijo. Ta se v praksi sestavi z zunanjimi izvajalci, ki imajo izkušnje. Osnovna metodologija vsebuje vse elemente, ki so za posamezno podjetje zanimivi, uporabni. Osnovna metodologija vsebuje pravila, ki povedo, pri katerih projektih je posamezen element obvezen, zaželen ali nepotreben. Ob začetku izvajanja konkretnega projekta se analizirajo lastnosti projekta. Z določitvijo pravil oblikujemo osnovno metodologijo glede na potrebe konkretnega podjetja. Druga možnost pa je oblikovati metodologijo s pomočjo zaposlenih, ki imajo dober vpogled v proces razvijanja programske opreme. Novonastala osnovna metodologija se lahko z uporabo spreminja in s tem prilagodi potrebam podjetja. V nadaljevanju bomo predstavili, kaj so metamodeli, ki definirajo posplošitvene metode. Sledila bo predstavitev repozitorija posameznih metod, ki vsebuje metode, ki so bile modelirane na osnovi metamodelov. Nato bodo predstavljeni načini, kako pridobiti metode, in načini oblikovanja nove metodologije [4].

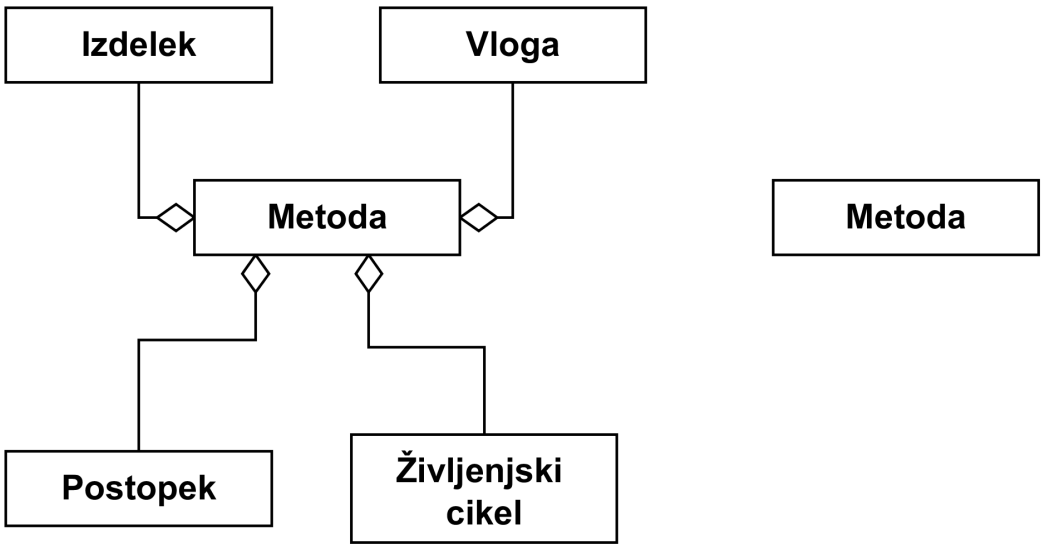
2.2.1 Metamodel

Če želimo metode med seboj primerjati, ovrednotiti ali izboljšati, moramo razumeti, iz česa so sestavljene. To lahko predstavimo z metamodelom metod. Metamodel metode pove, kaj vse metoda zajema, in povezave med njimi. Modeliranje metode na osnovi metamodelov je zelo splošen postopek, kar pomeni, da lahko vsak strokovnjak definira svoj metamodel, v katerem predstavi, na katere elemente in povezave razdeli metodo. Zaradi splošnosti so se v raziskovalnih sferah oblikovali standardi in navodila, ki definirajo modeliranje metod na osnovi metamodelov. Standardi in navodila predstavljajo postopke, kako iz posamezne metode definiramo elemente in povezave. Več o modeliranju na osnovi metamodelov govorimo

v poglavju 2.2.2. Najbolj predlagani metamodeli glede na vira [11, 12] so OPEN [13], SPEM 2.0 [14] in ISO/IEC 24744 [15]. Po analizi strokovnih člankov se za metamodeliranje najpogosteje uporablja standard ISO/IEC 24744. Razlog za to je v tem, ker glavni koncepti standarda temeljijo na ME [11]. ISO/IEC 24744 je standardiziran metamodel, ki ga je oblikovala mednarodna organizacija za standarde. Omenjeni metamodel definira, katere elemente in povezave moramo pridobiti iz posamezne metode. Ko so posamezne metode modelirane na osnovi metamodela, so pripravljene, da se dodajo v repozitorij metod [4].

2.2.2 Modeliranje metod po metamodelu

V prejšnjih poglavjih smo omenjali, da se metoda lahko razdeli na več elementov. Da so metode primerne za repozitorij, je treba posamezne metode modelirati na osnovi metamodelov. Z modeliranjem poskrbimo, da lahko metode med seboj združujemo ali pa jih med seboj primerjamo. Če želimo pravilno modelirati posamezne metode, moramo določiti raven zrnatosti (Angl. “granularity”). Z zrnatostjo definiramo, na kako velike dele razdelimo neko metodo. Henderson-Sellers in et al. [4] navajajo, da so nekateri raziskovalci dali druga imena, ki so vsebinsko enaka pojmu ”deli metod”(npr. po Harmsen in et al. [**?**] poznamo (Angl. “method fragments”), po Rolland in et al. [16] poznamo (Angl. “method chunks”) in po Röstlinger in et al. [17] (Angl. “method components”)). Ta poimenovanja se med seboj razlikujejo po nivoju zrnatosti. Isti avtor [4] v grobem razdeli zrnastost na fino in grobo. Na sliki 2.1 prikazujemo primer fine in grobe zrnatosti. Leva slika prikazuje fino zrnatost, saj prikazuje, katere elemente moramo definirati iz metode. Deli elementov metode so bili povzeti po Weitzerju [2]. Na desni strani pa prikazujemo grobo zrnatost, saj moramo le definirati metodo kot celoto.



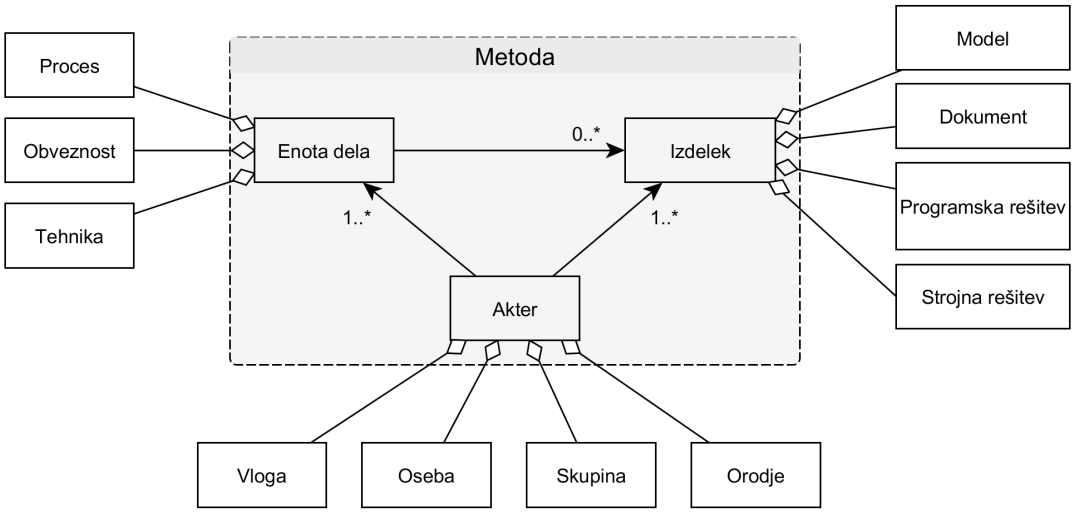
Slika 2.1: Zrnatost metod (leva slika prikazuje fino zrnatost, desna pa grobo)

Nivo zrnatosti in na katere elemente razdelimo posamezno metodo, je definirano v metamodelih. Največkrat uporabljeni metamodelni standard je ISO/IEC 24744 [4]. ISO/IEC 24744 definira, da se posamezna metoda standardizira tako, da se definirajo trije deli. Standard ISO/IEC 24744 razdeli metodo na tri dele, kar pomeni, da je nivo zrnatosti zelo grob. Standard razdeli metodo na:

- (Angl. "Producer"): predstavlja akterja, ki izvaja neko aktivnost. Na primer: pri metodi analiza trenutnega stanja iz slapovnega modela je izvajalec vodja projekta.
- Enota dela (Angl. "Work unit"): predstavlja aktivnost, ki se izvaja v metodi. Na primer, pri metodi analiza trenutnega stanja to predstavlja analizo trenutnega stanja podjetja.
- Izdelek (Angl. "Work produkt"): predstavlja rezultat, ko je opravljeno delo. V našem primeru bi bil izdelek lahko poročilo o trenutnem delu.

Na sliki 2.2 lahko vidimo povezave med omenjenimi elementi metod. Vidimo, da je obvezen element Akter. To pomeni, da moramo pri vsaki metodi, ko jo

modeliramo, definirati, kdo izvaja aktivnost. Z elementom Akter sta v povezavi enota dela in izdelek. Akter lahko opravi eno ali več enot dela. Prav tako akter lahko proizvede enega ali več izdelkov. Med elementoma enote dela in izdelka pa vidimo, da lahko enota dela vpliva na nobenega ali več izdelkov. Enota dela lahko spremeni, odstrani ali doda nove izdelke. Razbitje na takšne elemente ni prisotno samo v standardu ISO/IEC 24744, ampak je podobna razdelitev prav tako prisotna v metamodelih SPEM in OPEN [4]. Ko končamo modeliranje metode po metamodelu, jo lahko dodamo v repozitorij.



Slika 2.2: Razdelitev metode po standardu ISO/IEC 24744 [15]

2.2.3 Repozitorij metod

Repozitorij metod (Angl. "method base") običajno vsebuje metode, ki so bile pridobljene iz metodologij. Lahko pa vsebuje metode, ki so bile oblikovane iz delovnih izkušenj pri opravljanju in opazovanju dela. Vsaka pridobljena metoda ni nujno že pripravljena (ni bila modelirana po metamodelu), da jo vključimo v repozitorij metod. Z modeliranjem poskrbimo, da so metode v repozitoriju enako definirane. To nam olajša, da metode med seboj lažje primerjamo in ugotavljamo njihove povezave. Poleg primerjave in povezave pa nam metamodeliranje omogoča še eno

funkcionalnost. S pomočjo modeliranja lahko ugotovimo, ali je neka metoda primerna za repozitorij posameznih metod. Namreč, ko modeliramo, potrebujemo določene informacije iz te metode (npr. izvajalec, enota dela, izdelek ...). Če metoda nima potrebnih informacij, je ne moremo modelirati po metamodelu. Nemodelirana metoda pa je neprimena za repozitorij. Poleg ugotavljanja nepravilnosti oz. pomanjkanja podatkov o metodi nam modeliranje pomaga pri ugotavljanju podobnosti med metodami. S tem, ko modeliramo metode, jih posplošimo. To pomeni, da iz posamezne metode pridobimo tiste lastnosti, ki so lahko več metodam skupne, kar pomeni, da lahko ugotovimo podobnosti/razlike. Podajmo primer: V poglavju 2.2.2 smo modelirali metodo *Analiza trenutnega stanja* iz slapovnega modela. Ker je metoda že modelirana, jo lahko dodamo v repozitorij metod. Recimo, da si zdaj želimo dodati novo metodo. Metoda, ki jo bomo dodali, izhaja iz metodologije RUP in se glasi *Prepoznava zunanjih entitet*. Najprej moramo metodo modelirati po metamodelu. Iz metode je razvidno, da je *izvajalec* vodja projekta. *Enota dela* je prepoznava vseh zunanjih entitet (ljudi in sistemov), ki bodo sodelovali pri projektu. Pri tem nastane *izdelek*, ki je poročilo o zunanjih entitetah. Zdaj, ko smo končali modeliranje, lahko metodo *Prepoznave zunanjih entitet* primerjamo z metodo *Analize trenutnega stanja*. Iz metod je razvidno, da sta elementa izvajalca enaka, vendar sta elementa enota dela in izdelek različna. To pomeni, da lahko metodo dodamo v repozitorij metod. Tako imamo v repozitoriju dve metodi, ki sta različni in primerni za repozitorij posameznih metod [18]. Metodi sta primerni, ker sta modelirani po metamodelu. Ko imamo zbranih dovolj metod znotraj repozitorija, lahko definiramo novo metodologijo [4].

2.2.4 Pristopi oblikovanja metodologij

Do zdaj smo spoznali, kaj so metode, kako jih modeliramo po metamodelu in pod katerimi pogoji so dodane v repozitoriju metod. V tem delu si bomo pogledali pristope oblikovanja nove metodologije. Po principih SME lahko novo metodologijo oblikujemo na več načinov. Prvi pristop temelji na sestavljanju novih metodologij iz obstoječih metod (Angl. "Asembly-based"). Drugi pristop temelji na sestavljanju novih metodologij z uporabo abstrakcije na metodah, ki jih poznamo, ali na

osnovi metod, ki so oblikovane po metamodelih (Angl. "Paradigm-based"). Tretji in zadnji pristop temelji na razširitvah izbrane metode (Angl. "Extension-based"). V nadaljevanju bosta opisana največkrat uporabljena pristopa za konstruiranje metod [4]: pristop, ki temelji na sestavljanju novih metodologij iz obstoječih metod, ter pristop, ki izhaja iz abstrakcije metodologij in metamodelov.

2.2.4.1 Sestavljanje metodologij s pomočjo obstoječih metod

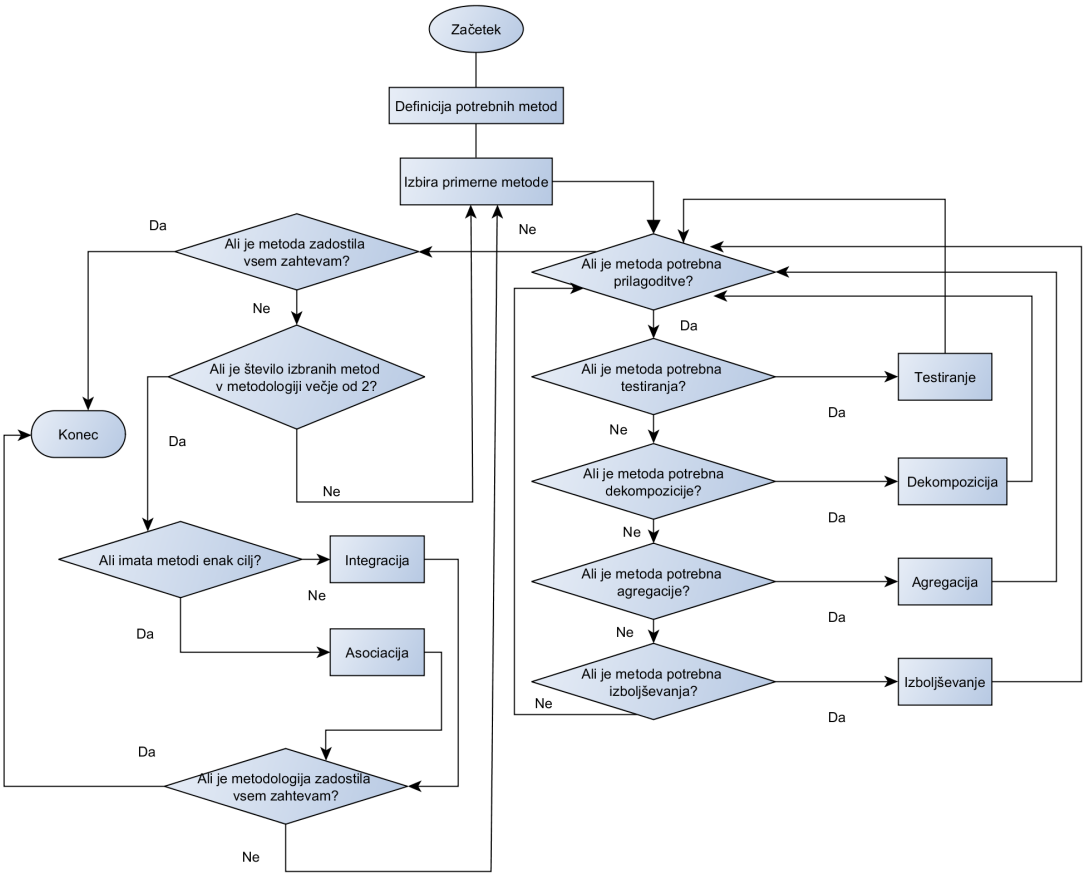
Sestavljalni pristop (Angl. "Asembly-based") je na sliki 2.3 sestavljen iz dveh pomembnih korakov. Prvi korak je izbira metode. Da lahko pravilno izberemo metodo, potrebujemo zahteve. Te so oblikovane na osnovi, kaj si želimo spremeniti v metodologiji, ki je trenutno v podjetju. Če ugotovimo, da je neustrezna/nepravilna posamezna metoda, v zahtevah zapišemo, kakšno metodo si želimo. Tak pristop je po Ralyté poimenovan (Angl. "Intention driven strategy") [19]. Če moramo oblikovati celotni proces, pa v zahtevah zapišemo, katere metode so potrebne za izvedbo posameznega procesa. Ta pristop je po Ralyté poimenovan (Angl. "Process driven strategy") [19]. Pri izdelovanju zahtev je priporočljivo, da so prisotni vsi razvijalci in organizatorji dela. Ko so zahteve sestavljene, se skozi oblikovanje metodologije ne spreminjajo [4]. Z določitvijo zahtev dobimo informacijo o primernosti posamezne metode iz repozitorija. Po izbiri metode iz repozitorija sledi ocenjevanje. Pri tem pogledamo, koliko zahtev izpolnjuje metoda. Če izpolni vse zahteve, to pomeni, da je primerna za metodologijo. Če ne prestane ocenjevanja, obstajajo nadaljnji postopki, ki izboljšajo posamezno metodo:

- testiranje (Angl. "Evalvation strategy") – testiranje, koliko zahtevam zadosti metoda;
- dekompozicija (Angl. "Decomposition strategy") – kadar metoda vsebuje elemente, ki niso potrebni v trenutnih zahtevah;
- agregacija (Angl. "Aggregation strategy") – kadar metoda ne pokriva vseh zahtev;
- izboljšanje (Angl. "Refinement strategy") – iskanje podobne metode iz repozitorija, ki dopolni trenutno izbrano metodo.

Po vsakem izvedenem postopku ponovimo postopek ocenjevanja, dokler niso izpolnjene vse zahteve. Če smo pri zahtevah iskali le metodo, ki bo izboljšala metodologijo, smo končali sestavljanje metodologije. Pri sestavljanju procesa pa, metodo dodamo na seznam primernih metod [19]. Ko imamo primernih več kot eno metodo, so po Ralyté potrebne strategije, ki združujejo metode v metodologijo. Po Ralyté imamo dve strategiji [19]. Prva se imenuje strategija povezovanja (Angl. "Association strategy"). Ta je uporabljena, kadar imamo metode, ki izpolnjujejo različne zahteve, vendar izdelki posamezne metode povezujejo metodi med seboj. To pomeni, da je izdelek prve metode obvezen vhod za začetek izvajanja druge metode. Rezultat te strategije je metoda, ki je sestavljena iz dveh delov. Prvi del je izdelek, ki je rezultat združitve izdelkov primarnih metod. Drugi del je aktivnost, ki je sestavljena iz aktivnosti predhodnih metod. Definirati moramo, v kakšnem zaporedju se aktivnosti izvajajo. Druga strategija temelji na integraciji (Angl. "Integration strategy"). To metodo lahko uporabimo, ko imamo na seznamu metode, ki dosežejo enak cilj, vendar z drugačnim postopkom. Strategija narekuje, da moramo prepoznati skupne elemente metod in jih združiti. Ko z naborom metod zadovoljimo vsem zahtevam, smo končali proces. Na sliki 2.3 si lahko pogledamo diagram poteka sestavljanja metodologije s pomočjo obstoječih metod.

2.2.4.2 Sestavljanje metodologije na osnovi metamodela

Kot smo že navedli pri sestavljalnem pristopu, metodologijo sestavimo iz znanih metod. Pri drugem pristopu pa metodo ali metodologijo sestavimo tako, da modeliramo na osnovi metamodela. Ta je lahko sestavljen iz več delov, toda po Raylet je priporočljivo, da metamodel razdelimo na dva modela [19]. Prvi predstavlja produkt in vse lastnosti, ki so povezane z njim. Drugi model pa predstavlja cilje, aktivnosti in navodila za dosego ciljev in pravilno uporabo aktivnosti [20]. Procesni in produktni model se morata med seboj ujemati. To pomeni, da se mora model, ki predstavlja izdelek, ujemati s procesnim modelom, v katerem definiramo, kakšni izdelki bodo nastali, ali so potrebni za izvedbo aktivnosti [4]. Ko sestavimo procesni in produktni model, lahko glede na lastnosti, ki so predstavljene v mode-



Slika 2.3: Diagram poteka sestavljanja metodologije s pomočjo obstoječih metod

lih, naredimo novo metodo. Če na začetku uporabimo metamodel, ki predstavlja metodologijo in posamezne metode, pa lahko po tem načinu sestavimo novo metodologijo [4].

V tem poglavju smo predstavili teorijo SME, kako so povezane z metodologijami in kako uporabijo znanje SME za oblikovanje novih metodologij. V nadaljevanju bodo predstavljene metodologije in modeli, ki bodo v praktičnem delu uporabljene, saj bomo iz njih črpali metode za oblikovanje nove metodologije.

2.3 Metodologije razvoja programske opreme

V nadaljevanju bomo predstavili nekaj izbranih metodologij za razvoj programske opreme. Najprej bomo predstavili RUP, nato pa še agilne metodologije, kot so: Scrum, ekstremno programiranje in Kanban, ter na koncu metodologijo Kaos. Te metodologije nam bodo koristile pri sestavljanju repozitorija metod.

2.3.1 RUP

RUP je kratica za "Rational Unified Process" in je moderni proces razvoja programske opreme [21]. RUP opisuje postopke razvoja programske opreme, pri čemer se močno opira na uporabo modelirnega jezika UML in tehnik, ki jih ta standard ponuja. RUP je predstavljen s tremi vidiki [21]:

1. dinamični vidik, ki prikazuje faze skozi razvoj;
2. statični vidik, ki prikazuje postopke v procesu razvoja;
3. praktični vidik, ki predstavi dobre prakse.

RUP je fazni model, ki je sestavljen iz štirih diskretnih faz. Faze v RUP-u so: [21]

1. **Začetna faza**: v njej se vzpostavi poslovni pogled za načrtovan sistem. V tej fazi se prepoznajo vse zunanje entitete, ki bodo v stiku z načrtovanim sistemom. Za vsako zunanjo entiteto moramo definirati, na kakšen način

bodo dostopale stranke oziroma na kakšen način bodo stranke uporabljale načrtovani sistem. Začetna faza nam posreduje informacije, kakšen prispevek bo sistem imel v poslovanju. Če bo izdelava sistema imela majhen prispevek, se lahko načrtovanje sistema konča že v tej fazi.

2. **Zbiranje informacij**: cilj faze zbiranja informacij je zgraditi razumevanje problemske domene in vzpostaviti arhitekturo za sistem, ki ga bomo razvijali. V tej fazi se prav tako oblikuje projektni načrt in se odkrijejo potencialni problemi pri razvijanju programske opreme. Ob koncu te faze je oblikovan model potreb, ki je sestavljen iz primerov uporabe UML, opisa programske arhitekture in iz razvojnega načrta za programsko opremo.

3. **Konstrukcija**: faza konstrukcije vključuje oblikovanje sistema, programiranje in testiranje. Deli sistema so razviti paralelno in vključeni v končni sistem. Ob koncu te faze imamo delujoči sistem in natančno dokumentacijo, ki je pripravljena za stranko.

4. **Prevzem**: zadnja faza je prevzem.V njej naš sistem prenesemo iz razvojnega okolja v okolje, ki ga bo uporabila stranka, in poskrbimo za delovanje. Ta faza je največkrat prezrta v drugih procesih razvoja programske kode, vendar je v resnici draga in včasih problematična. Rezultat te faze je dokument, ki vsebuje dokumentacijo razvitega sistema, ki je delujoč.

Statični pogled v RUP-u se osredini na aktivnosti, ki se dogajajo med procesom izdelovanje programske kode. RUP poimenuje te aktivnosti *postopke* (Angl. "Workflow"). RUP predpisuje šest glavnih postopkov in tri podporne postopke. Ko je bil oblikovan model RUP, je bilo vključenih veliko idej iz jezika UML. Tako so postopki v RUP-u velikokrat povezani z objekti UML. Glavni in podporni postopki so predstavljeni v tabeli 2.1 [21].

Prednost v dinamični in statični predstavitvi faz je, da posamezni procesi niso povezani z določenimi postopki. Tako so lahko vsi postopki prisotni v vseh fazah skozi neki proces razvoja programske opreme. V začetnih fazah so najpomembnejši postopki namenjeni oblikovanju poslovnega modela in potreb, proti koncu pa

Postopek	**Opisi postopkov**
Poslovno modeliranje	Modeliranje poslovnih procesov s pomočjo primerov uporabe.
Zajem zahtev	Prepoznava akterjev, ki komunicirajo s sistemom, in oblikovati primer uporabe, ki prikazuje potrebe sistema.
Analiza in oblikovanje	Oblikovanje modela načrta, ki je dokumentiran s pomočjo UML-a.
Implementacija	Implementacija komponent v sistemu.
Testiranje	Testiranje implementiranega sistema.
Namestitev	Dostava nastalega produkta strankam.
Konfiguracija in zamenjava upravljanja	Določevanje in izvajanje sprememb v sistemu.
Upravljanje projekta	Izvajanje pomoči pri razvijanju sistema.
Upravljanje z razvojnim okoljem	Določevanje primerne programske opreme za realizacijo sistema.

Tabela 2.1: Postopki v RUP-u

se pomembnost prenese na testiranje in prenos razvojnega okolja. Praktičen vidik RUP-a opiše dobre prakse, ki pripomorejo k boljšemu razvoju programske opreme. RUP je oblikoval šest glavnih praks [21]:

1. Iterativni razvoj – razvoj pri, katerem ponavljamo faze razvoja programske opreme v krajšem časovnem obdobju za dosego boljšega izdelka.

2. Obvladovanje zahtev – natančna dokumentacija strankinih zahtev in slediti spremembam teh zahtev. Analiziranje sprememb in njihov vpliv na sistem pred implementacijo.

3. Uporaba komponentne arhitekture – strukturiranje sistemske arhitekture s pomočjo komponent.

4. Vizualno modeliranje – uporaba UML-a za predstavitev statičnega in dinamičnega pogleda na programsko opremo.

5. Preverjanje kakovosti – preverjanje, da programska oprema dosega standarde, ki jih predpisuje podjetje.

6. Nadzorovanje sprememb v programski opremi – nadzorovanje sprememb v programski opremi in uporaba sistema za upravljanje sprememb v programski kodi.

RUP ni primeren za vse vrste razvoja programske opreme. Najpomembnejša inovacija v RUP-u je ločevanje faz in postopkov ter prepoznava, da je prenos programske kode iz razvojnega okolja v strankino okolje. Vsaka faza je dinamična in ima svoje cilje. Postopki so statični in so tehnične aktivnosti, ki niso povezane z določeno fazo, ampak se lahko pojavijo v več fazah hkrati skozi celoten proces razvoja programske opreme.

2.3.2 Scrum

Scrum je ena izmed najuporabnejših agilnih metodologij [22]. Scrum je sestavljen iz preprostih pravil in vlog, ki omogočajo agilno razmišljanje [2]. Z upoštevanjem

teh pravil in vlog se v podjetju povečata produktivnost in kakovost izdelka [2]. Scrum ne določa, koliko dela mora opraviti posamezna skupina. Skupine si lahko same določajo, koliko dela bodo opravile in na kakšen način bo delo izvedeno. To pripelje do prijetnega in produktivnega okolja. Metodologija je oblikovana na tak način, da se osredinja na tri procese [2]. Prvi spodbuja organizacijo nalog glede na poslovno korist. To pomeni, da organiziramo naloge tako, da bodo ob koncu nekega kratkega obdobja stranke zadovoljne. Drugi proces se osredinja na dosego čim večjega nivoja uporabnosti nastalega izdelka. Zadnji proces se osredinja na prihodke in koristi izdelka [2]. Metodologija Scrum je prilagodljiva glede na zahteve. To pomeni, da se lahko prilagodi zahtevam, ki nastanejo na novo ali spremenijo trenutne zahteve. Scrum ima prav tako definirane vrednote [23]. Te vrednote so:

- predanost;
- pogum;
- osredinjenost;
- odprtost;
- spoštljivost.

Te vrednote se skupine naučijo skozi uporabo metodologije Scrum. Uspešnost Scruma je odvisna od ljudi, kako hitro sprejmejo te vrednote. Scrum je sestavljen iz treh vlog, ki sodelujejo pri vsakem projektu, treh izdelkov in treh obredov. Te tri vloge so: stranka/lastnik projekta, razvojna skupina in skrbnik procesa. V nadaljevanju opisujemo te vloge.

2.3.2.1 Skrbnik izdelka

Skrbnik izdelka zastopa ciljno stranko, ki ji želimo izdati/prodati naš izdelek. Njegova naloga je maksimirati vrednost produkta. Skrbnik izdelka je glavna oseba, ki je odgovorna za seznam zahtev. Vzdrževanje seznama zahtev vključuje [23]:

- Dodajanje posameznih zahtev na seznam zahtev.

- Uredi posamezne zahteve v seznamu z namenom dosege več ciljev.
- Vrednotenje dela, ki ga opravi razvojna skupina.
- Zagotovitev, da je seznam zahtev viden in vsem jasen. Seznam prav tako prikazuje, kaj bo razvojna skupina razvijala naprej.
- Poskrbi, da razvojna skupina razume posamezne zahteve na seznamu zahtev.

Skrbnik izdelka je samo ena oseba. Njegova glavna naloga je dostava izdelka do stranke. Pri tem se trudi doseči čim večji nivo kakovosti izdelka. Vse spremembe zahtev gredo prek skrbnika izdelka, saj on najbolje pozna seznam zahtev. Da je delo skrbnika izdelka uspešno, mora celotno podjetje spoštovati njegove odločitve. Odločitve so vidne v obliki določanja prioritet za posamezne zahteve in vsebine na seznamu zahtev. Skrbnik izdelka nalaga delo razvojnim skupinam. Razvojne skupine pa se samoorganizirajo ter poskušajo realizirati čim več zahtev iz seznama zahtev.

2.3.2.2 Razvojna skupina

Razvojna skupina, kot že ime pove, se ukvarja z razvojem. Sestavljena je iz strokovnjakov, ki opravljajo delo. Podjetje oblikuje razvojne skupine tako, da so sposobne samoorganicije in samostojnega dela. Rezultat tega je boljša učinkovitost in zmogljivost. Razvojne skupine imajo naslednje lastnosti [23]:

- So sposobni samoorganizacije.
- Razvojna skupina ima dovolj znanj za zaključitev iteracije.
- Scrum ne predpisuje nobenih titul članom razvojne skupine.
- Scrum ne dovoli, da se razvojna skupina razčleni na podskupine.
- Posamezniki imajo lahko specifične sposobnosti, vendar je celotna odgovornost odvisna od skupine in ne od posameznika.

Velikost razvojne skupine ne sme biti ne premajhna ne prevelika. Priporočljivo je, da ima vsaka skupina od tri do devet članov v eni razvojni skupini. Skrbnik izdelka in skrbnik procesa nista vključena, razen če opravljata naloge, ki so na seznamu zahtev.

2.3.2.3 Skrbnik procesa

Zadnja vloga je skrbnik procesa (Angl. "Scrum master"). Njegova odgovornost je, da je metodologija Scrum pravilno razumljena in izvajana. Skrbnik procesa skupaj s skrbnikom izdelka podjetja poskrbi, da *razvojna skupina* razume uporabniške zgodbe. Ob začetku razvoja nekega produkta *Scrum master* nima veliko obveznosti pri razvoju, saj mora poskrbeti, da se *razvojna skupina* drži pravil, ki jih določa metodologija Scrum. Najpomembnejša vloga *Scrum masterja* je, da določa tok dela ob spremembah. *Stranka/Lastnik* podjetja lahko že med razvojem doda uporabniške zgodbe, ki ogrožajo trenutni potek. Naloga *Scrum masterja* pa je, da se dogovori *s stranko/z lastnikom* podjetja o tem, ali je mogoče novo uporabniško zgodbo preložiti na nov sprint ali je potreba po ponovnem zagonu sprinta (iteracije). Če pogledamo vlogo *Scrum masterja* od daleč, vidimo, da je njegova naloga, da skrbi za tempo razvoja in da ob ovirah poskrbi, da se razvoj ne zaplete ali se celo upočasni [2].

2.3.2.4 Načrtovanje sprinta

Sprint je načrtovan na začetku vsakega sprint cikla [2]. Skrbnik izdelka pride na sestanek s seznamom zahtev. Te so urejene glede na prioriteto. Skrbnik izdelka skupaj z razvijalci analizira posamezne zahteve. Razvijalci nato časovno ocenijo, koliko časa se bo razvijala posamezna zahteva. Razvojna skupina potem določi, koliko zahtev bo realizirala v enem sprint ciklu. Rezultat tega sestanka je sprint backlog.

2.3.2.5 Analiza sprinta

Analiza sprinta poteka po koncu vsakega sprint cikla. Na tem sestanku se predstavi, katere zahteve so bile realizirane in na kak način. Sestanek lahko poteka na

način, da pokažemo prototip izdelka. Tega sestanka se lahko udeležijo stranke in tako predstavijo svoj vidik nad prototipom ali izdelkom [2].

2.3.2.6 Dnevni sestanki Scrum

Dnevni sestanki, kot ime že navaja, potekajo dnevno. Ne potekajo več kot 15 minut. Na njih vsak razvijalec pove tri informacije: kaj je dokončal včeraj, na čem bo delal danes in poroča, ali ima kakšne probleme z razvijanjem. S takšnim pogovorom hitreje ugotovimo napredek in probleme, s katerimi se razvijalci spoprijemajo [2].

2.3.2.7 Product backlog

Product backlog je seznam zahtev, ki so urejene. Seznam zahtev mora zajemati vse lastnosti novonastalega produkta. Poduckt backlog je sestavljen iz [2]:

- novih funkcij programske opreme (Angl. "Features");
- napak v programski opremi (Angl. "Bugs");
- tehničnih del (Angl. "Technical work");
- pridobivanj znanj (Angl. "Knowledge acquisition").

Nove funkcije si lahko predstavljamo kot dodatek novega gumba na uporabniški vmesnik. Napake si lahko predstavljamo, da neki program nepravilno izračuna rezultat. Primer tehničnega dela je *posodobitev operacijskega sistema na razvijalskih računalnikih*. Primera pridobivanja znanja sta raziskava programskih knjižnic in ovrednotenje [2].

2.3.2.8 Sprint backlog

Sprint backlog je seznam zahtev iz Produckt backloga, ki so bile ovrednotene in organizirane glede na prioriteto. Razvojna skupina se bo na te zahteve osredinila v tekočem sprintu [2].

2.3.2.9 Burn down chart

Burn down chart je diagram, ki prikazuje, koliko zahtev je bilo doseženih v tekočem sprintu. Z Burn down chartom pridobimo vpogled, kako in s kakšno hitrostjo napreduje projekt [2].

2.3.3 Ekstremno programiranje

Kot velja za Scrum, prav tako velja za *ekstremno programiranje*, da temelji na principih in vrednotah agilnih metodologij [24]. V primerjavi s Scrumom je ekstremno programiranje bolj osredinjeno na proces razvijanja programske opreme. Glavni cilj metodologije Scrum je izboljšava produktivnosti, pri čemer je cilj ekstremnega programiranja izboljšava pisanja programske kode. Ekstremno programiranje se osredini na najpogostejše napake, s katerimi se razvijalci ukvarjajo.

Ekstremno programiranje temelji na 12 praksah. Te so razdeljene na pet kategorij [24]. Prva opisuje programerske prakse. Programerska praksa je razdeljena na dva dela. Prvi priporočada testno vodeni razvoj (Angl. "Test driven development") [24]. Pri takšnem razvoju razvijalec, preden spiše programsko opremo, najprej napiše teste. Na začetku razvijanja projekta testi vračajo negativne rezultate. Tako razvijalec razvija, dokler vsi testi ne postanejo pozitivni. Takšni testi se imenujejo *testi enot* (Angl. "Unit testing") [24]. Enota lahko predstavlja razred, metodo ali funkcijo, nad katero se izvedejo testi. S tem, ko programer napiše teste pred pisanjem programske kode, preprečimo najpogostejše napake in lažje rešimo zahtevnejše probleme, kar se vidi v kodi. Kot druga praksa v programerski praksi je programiranje v parih (Angl. "Pair programing") [24]. Programiranje v paru je videti tako, da dva programerja sedita skupaj pred računalnikom in pišeta programsko opremo. V veliko primerih eden v paru piše programsko kodo drugi pa opazuje programsko kodo in razpravlja o napisani kodi. Podjetja, ki uporabljajo programiranje v parih, med pisanjem programske kode, proizvedejo manj napak. Manj programerskih napak se pojavi, ker isto programsko kodo opazujeta dva programerja. Prav tako programiranje v parih vpliva na utrujenost zapo-

slenih, ker med tem, ko en razvijalec programira, drugi počiva. Če se razvijalec utrudi, se vlogi lahko zamenjata. Učinkovitost dela v parih naraste, kadar se poleg razmišljanja o programerski kodi tudi o njej razpravlja. S programiranjem v paru pa prav tako poskrbimo, da več ljudi pozna sestavo programske opreme z vidika kode.

Druga kategorija opisuje integracijske prakse. Te govorijo o stanju kode in njeni pogostosti graditve kode. Prva praksa je, da posamezna graditev kode ne sme trajati več kot deset minut. Pri graditvi kode sta prav tako vključena testiranje in izdaja poročila o pozitivnih in negativnih testih. Graditev kode omejimo na deset minut, ker dolžina gradnje kode vpliva na pogostost, kolikokrat razvijalci gradijo kodo. Več časa, kot je potrebnega za graditev kode, manj časa imajo razvijalci za pisanje kode. S tem pa je tudi izvedenih manj testov na kodi. Druga praksa pri integraciji je zvezna integracija. V velikih podjetjih več razvijalcev razvija na enakem delu kode hkrati. Ker vsi razvijalci ne morejo hkrati pisati v isto datoteko, se v podjetjih uporabljajo upravitelji različic. Ti omogočajo združevanje konfliktnih datotek. Ker razvijalec ne more vedeti, da se koda pravilno obnaša, je vloga zvezne integracije, da to testira. Tako zagotovimo, da je problemov pri integraciji manj in da se popravijo dovolj zgodaj.

Tretja kategorija so načrtovalne prakse. Kot velja za Scrum, prav tako velja tudi za ekstremno programiranje, da se razvija v ciklih, ki trajajo kratko časovno obdobje. Ekstremno programiranje definira tedenske cikle. V vsakem tedenskem ciklu razvijalci realizirajo neke kratkoročne cilje. Za daljše časovno obdobje pa se uporabljajo četrtletni cikli. Za vsak četrtletni cikel programerska skupina pogleda od daleč in razmišlja o tem, kako bodo združili posamezne zgodbe, ki so bile razvite skozi tedenske cikle. Druga praksa so mlahavi dodatki (Angl. “slacks”) [24]. To so majhne dopolnitve projekta. Če nastanejo problemi pri razvijanju pomembnejših zgodb, se mlahavi dodatki opustijo. Naslednja kategorija so prakse o ekipah. Ekstremno programiranje ne predpostavlja samo praks glede programiranja, ampak tudi to, kako sodelovati med seboj. Prva praksa je, da so razvijalci skupaj (Angl. “sit

together") [24]. Čeprav je razvijanje programske kode na prvi pogled videti kot delo posameznika, ki je izoliran, je v realnosti zelo socialno zahtevna aktivnost. Razvijalci se stalno med seboj posvetujejo o problemih, nasvetih, pristopih. Če so v istem prostoru, je socializiranost naravno spodbujana. Naslednje priporočilo je izobraževalno delovno okolje. Ena izmed najpogostejših tehnik za izboljšanje izobraževalnosti v delovnem okolju je tabla, na kateri so izobešena opravila, ki jih morajo realizirati razvijalci. S takšno tablo razvijalci dobijo občutek o tem, kje so in kaj še morajo razvijati [24].

2.3.4 Kanban

Kanban je agilna metoda, ki se ukvarja z izboljšanjem procesov [25]. Kot velja za Scrum in ekstremno programiranje, tako tudi za Kanban velja, da potrebuje svoj način razmišljanja. Kanban je bil v preteklosti uporabljen kot pojem v industriji. Z letom 2010 pa je postala tehnika zanimiva za razvijalce programske opreme. Fokus Kanbana je povsem drugačen od Scruma in ekstremnega programiranja. Scrum se osredini na vodenje projekta, ekstremno programiranje se osredini na način razvijanja programske opreme, Kanban pa na izboljšanje procesa izdelovanja programske opreme. Tudi Kanban ima principe. Glavni principi so [25]:

- Začni, kar delaš.
- Sledi inkrementalnim in evolucijskim spremembam.
- Spoštuj vloge, dolžnosti in nazive.

Kanban nam ne predstavi, kako voditi projekt, ampak nam predstavi, kako moramo razmišljati, da bomo spremenili proces izdelovanja programske kode. Cilj Kanbana je implementirati majhne spremembe v trenutni sistem razvoja programske kode ob vsakem novem projektu. Glavno orodje, ki se uporablja pri Kanbanu, je kanban tabla. S kanban tablo ponazorimo proces. Razlika med Scrum tabelo in kanban tablo je v tem, da kanban tabla ne prikazuje opravil, ampak samo zgodbe. Prav tako je število stolpcev odvisno od ekipe do ekipe glede na to, s kakšnim

projektom se ukvarjajo. Skozi proces izdelovanja nekega projekta se opravila premikajo od leve proti desni glede nato, kdo je trenutno odgovoren za tisto dolžnost. Skozi potek projekta se vidi, pri katerih dejavnostih se nabirajo zgodbe. Informacija o nabiranju zgodb pove, pri katerih dejavnostih se lahko naredi optimizacija, in tako se pospeši celotni proces razvoja programske opreme[25].

2.3.5 Model Kaos

Pri modelu Kaos se moramo osrediniti na vidik, ki ga vidi programer. Če želimo razumeti, kako poteka proces razvoja programske opreme, se moramo osrediniti, kako programerji pristopajo k razvoju programske opreme. Model Kaos združuje linearno zanko reševanja problemov in fraktale (termin, prevzet iz matematične teorije kaosa), ki predstavljajo kompleksnost razvoja programske opreme. Linearna zanka, ki rešuje programerske probleme, je sestavljena iz štirih različnih stanj [26]. Ta stanja so:

- definicija problema;
- tehnični razvoj;
- dodajaje rešitev;
- status quo.

2.3.5.1 Definicija problema

V stanju definicija problema programer izbere specifičen problem, ki ga bo realiziral. Po izbiri odkrije, s katerimi težavami se bo spoprijemal pri rešitvi tega problema. Včasih stranke točno vedo, kaj si želijo, toda včasih imajo problem z izražanjem svojih želja. Zato je razvijalcu problem reševati probleme, ki niso dobro definirani. Razvijalec mora ob izbiri nekega probleme določiti, kaj bo razvil, da bo uresničil strankino potrebo [26].

2.3.5.2 Tehnični razvoj

V tem stanju razvijalci razvijajo programsko kodo. Če je problem tehničen, ga bodo razvijalci razvili tehnično. Profesionalni razvijalci uporabljajo razvijalska orodja in metodologije, ko je potreba po njih. Toda razvijalci ne morejo uporabiti tehnologije za reševanje netehnoloških problemov. To lahko pripelje do napačne rešitve ali rešitve, ki ni kakovostna. Napačna uporaba tehnologije lahko pripelje do neuspeha projekta [26].

2.3.5.3 Dodajaje rešitev

V tem stanju razvijalci vključijo nastalo programsko kodo v celoten sistem. Poleg integracije programske kode v tem stanju tudi oglašujemo, prodajamo in dostavljamo rešitev do strank. Uporabniki velikokrat zavrnejo ali ignorirajo izdelek, ker ugotovijo, da je bila rešitev napačno implementirana ali je napredek izdelka zanemarljiv [26].

2.3.5.4 Status quo

Status quo predstavlja trenutno stanje sistema. To vključuje vključuje tehnološke ter finančne in socialne okoliščine sodelujočih. Ob prihodu nove tehnološke rešitve se pojavi novi status quo in se zanka ponovi [26].

Linearna zanka za reševanje programerskih problemov se uporabi na več nivojih nekega projekta. Tako je uporabljena na problemih, ki se tičejo celotnega sistema ali samo nekega programerskega dodatka. Življenjski cikel Kaosa odkrije probleme skozi čas projekta. Z uporabo fraktalov model Kaos pokaže, da se faze med seboj prepletajo in niso v nekem zaporedju [26].

2.4 Povezava med agilnimi metodologijami in metodo razvoja

Do zdaj smo predstavili metodologije razvoja in situacijski pristop oblikovanja metodologije. Zdaj bomo uporabili principe situacijskega pristopa oblikovanja metodologije in analizirali opisane metodologije z namenom identificirati zanimive metode za repozitorij. Metode bomo prevzeli iz metodologij in metod, ki smo jih definirali v poglavju 2.3. Vključili bomo metodologije RUP, Scrum, Kanban in Ekstremno programiranje. Metodologija Kaos ima definirane splošne metode, kot so na primer: pisanje programske kode, pogovor s strankami itn. Takšne metode so preveč splošne za našo bazo metod, zato metodologija Kaos ne prispeva veliko k naboru metod.Metode iz teh metodologij bodo napolnile naš repozitorij metod. Na metodah bomo uporabili metamodel in razbili metode na dele. Metode bomo razbili na načine, ki jih definira standard ISO/IEC 24744. To pomeni, da bomo iz vsake posamezne metode pridobili informacije o izvajalcu aktivnosti, enoti dela in o izdelku. S takšno razdelitvijo je nivo zrnatosti širok, saj metodo razdelimo na večje dele.

2.4.1 Predstavitev metod s standardom ISO/IEC 24744

Metode, ki smo jih opisali v prejšnjem poglavju, bomo zdaj predstavili z elementi, ki jih predstavi standard ISO/IEC 24744. To pomeni, da bomo vsako metodo razdelili na tri dele, ki smo jih definirali v podpoglavju 2.2.2. Tabele od 2.2 do 2.6 predstavljajo metode iz RUP. Tabela 2.7 predstavlja metode iz metodologije Scrum. Tabela 2.8 predstavlja metode iz metodologije ekstremno programiranje. Zadnja tabela 2.9 predstavlja metode iz metodologije Kanban. Za vsako metodo definiramo, iz katere metodologije izhaja metoda, akterja, enoto dela in izdelek. Takšna definicija delov je definirana v standardu ISO/IEC 24744. S takšno razdelitvijo metod poskrbimo, da so metode predstavljene enotno v repozitoriju posameznih metod. Z analizo teh metodologij smo pridobili repozitorij metod, iz katerega lahko v nadaljevanju oblikujemo metodologijo. V nadaljevanju bomo uporabili ta repozitorij, da bomo iz njega pridobili ustrezne metode, ki bodo ustrezale podjetju,

njihovim zahtevam in potrebam.

Metoda	**Akter**	**Enota dela**	**Izdelek**
Oblikovanje dokumentacije o obstoječem sistemu	Sistemski analitik/Končni uporabnik	Sistemski analitik in končni uporabnik oblikujeta dokumentacijo o obstoječem sistemu	Dokumentacija o obstoječem sistemu
Oblikovanje dokumentacije o poslovnem sistemu	Sistemski analitik/Končni uporabnik	Sistemski analitik in končni uporabnik oblikujeta dokumentacijo o poslovnem sistemu	Dokumentacija o poslovnem sistemu
Zajem zahtev	Sistemski analitik / končni uporabnik	Sistemski analitik in končni uporabnik zajameta zahteve	Zbirka virov
Definicija funkcionalnih zahtev	Sistemski analitik/Končni uporabnik	Sistemski analitik in končni uporabnik definirata funkcionalne zahteve	Funkcionalne zahteve
Definicija nefunkcionalnih zahtev	Sistemski analitik/Končni uporabnik	Sistemski analitik in končni uporabnik definirata nefunkcionalne zahteve	Nefunkcionalne zahteve
Oblikovanje diagrama primerov uporabe	Sistemski analitik/Končni uporabnik	Sistemski analitik in končni uporabnik oblikujeta diagram primera uporabe	Diagram primera uporabe
Definiranje vmesnikov	Sistemski analitik/Končni uporabnik	Sistemski analitik in končni uporabnik definirata obliko vmesnikov	Definicija oblik vmesnikov
Oblikovanje slovarja izrazov	Sistemski analitik/Končni uporabnik	Sistemski analitik in končni uporabnik oblikujeta slovar izrazov	Slovar izrazov
Ureditev zahtev	Sistemski analitik / končni uporabnik	Sistemski analitik in končni uporabnik uredita zahteve	Specifikacija zahtev
Potrditev zahtev	Sistemski analitik / končni uporabnik	Sistemski analitik in končni uporabnik skupaj potrdita zahteve	Potrdilo o ustreznosti specifikacije zahtev

Tabela 2.2: Metode iz metodologije RUP, 1. del [21, 27]

Metoda	Akter	Enota dela	Izdelek
Izdelava podatkovnega modela	Sistemski analitik/Sistemski arhitekt/Preizkuševalec	Izdelava podatkovnega modela	Poslovni model
Izdelava procesne logike	Sistemski analitik/Sistemski arhitekt/Preizkuševalec	Izdelava procesne logike	Model procesne logike
Izdelava procesnega modela	Sistemski analitik/Sistemski arhitekt/Preizkuševalec	Izdelava procesnega modela	Procesni model
Izdelava prototipa	Sistemski analitik/Sistemski arhitekt/Preizkuševalec	Izdelava prototipa	Prototip
Izdelava predloge tehnične arhitekture	Sistemski analitik/Sistemski arhitekt/Preizkuševalec	Izdelava predloge tehnične arhitekture	Predlog tehnične arhitekture sistema
Oblikovanje strategije testiranja	Sistemski analitik/Sistemski arhitekt/Preizkuševalec	Oblikovanje strategije testiranja	Strategija testiranja
Oblikovanje gradiva za predstavitev analize	Sistemski analitik/Sistemski arhitekt/Preizkuševalec	Oblikovanje gradiva za predstavitev analize	Gradivo za predstavitev analize
Oblikovanje potrdila o ustreznosti analize	Sistemski analitik/Sistemski arhitekt/Preizkuševalec	Oblikovanje potrdila o ustreznosti analize	Potrdilo o ustreznosti analize

Tabela 2.3: Metode iz metodologije RUP, 2. del [21, 27]

Metoda	Akter	Enota dela	Izdelek
Izdelava načrta podatkovne baze	Načrtovalec podatkovne baze	Načrtovalec podatkovne baze izdela načrt podatkovne baze	Načrt podatkovne baze
Izdelava načrta programskih modulov	Načrtovalec aplikacije	Načrtovalec aplikacije izdela načrt programskih modulov	Načrt programskih modulov
Izdelava načrta dokumentacije	Izdelovalec dokumentacije	Izdelovalec dokumentacije izdela načrt dokumentacije	Načrt dokumentacije
Izdelava načrta prehoda na nov sistem	Skrbnik podatkovne baze	Skrbnik podatkovne baze izdela načrt prehoda na nov sistem	Načrt programskih modulov
Izdelava načrta testiranja	Načrtovalec aplikacije/Načrtovalec podatkovne baze	Načrtovalec aplikacije ali načrtovalec podatkovne baze izdela načrt testiranja	Načrt testiranja
Razvijanje programske opreme	Razvijalec aplikacije/Razvijalec podatkovne baze	Razvijalec aplikacije ali razvijalec podatkovne baze razvijata programsko opremo	Programska oprema
Testiranje programske opreme	Razvijalec aplikacije/Razvijalec podatkovne baze/Tester aplikacije/Tester podatkovne baze	Testiranje programske opreme	

Tabela 2.4: Metode iz metodologije RUP, 3. del [21, 27]

Metoda	Akter	Enota dela	Izdelek
Namestitev programske opreme	Načrtovalec podatkovne baze/Načrtovalec aplikacije/Skrbnik podatkovne baze/Sistemski administrator/Skrbnik aplikacije/Postavitveni inženir/Informacijski varnostni inženir	Namestitev programske opreme v strankino okolje	Poročilo o namestitvi programske opreme
Dodelitev pravic za delo s programsko opremo	Načrtovalec podatkovne baze/Uvajalec/Skrbnik podatkovne baze/Končni uporabnik/Sistemski administrator/Skrbnik aplikacije/Postavitveni inženir/Poslovni lastnik/Informacijski varnostni inženir	Dodeljevanje pravic za delo s programsko opremo zaposlenim v strankinem podjetju	Poročilo o pregledu dodeljenih pravic
Prevedba podatkov	Načrtovalec podatkovne baze/Uvajalec/Skrbnik podatkovne baze/Končni uporabnik/Sistemski administrator/Skrbnik aplikacije/Postavitveni inženir/Poslovni lastnik/Informacijski varnostni inženir	Vzpostavitev začetnega stanja podatkov	Poročilo o pravilni prevedbi podatkov
Izvedba potrditvenega testa programske opreme	Načrtovalec podatkovne baze/Uvajalec/ Skrbnik podatkovne baze/Končni uporabnik/Sistemski administrator/Skrbnik aplikacije/Postavitveni inženir/Poslovni lastnik/Informacijski varnostni inženir	Izvajanje potrditvenega testiranja programske opreme	Poročilo o izvedbi potrditvenega testa programske opreme

Tabela 2.5: Metode iz metodologije RUP, 4. del [21, 27]

Metoda	Akter	Enota dela	Izdelek
Izvedba končnega testa programske opreme	Načrtovalec podatkovne baze / Uvajalec/Skrbnik podatkovne baze/Končni uporabnik/Sistemski administrator/Skrbnik aplikacije/Postavitveni inženir/Poslovni lastnik/Informacijski varnostni inženir	Izvajanje končnega testiranje programske opreme	Poročilo o izvedbi končnega testa
Uvajanje uporabnikov in skrbnikov za delo z programsko opremo	Načrtovalec podatkovne baze/Uvajalec/Skrbnik podatkovne baze/Končni uporabnik/Sistemski administrator/Skrbnik aplikacije/Postavitveni inženir/Poslovni lastnik/Informacijski varnostni inženir	Uvajanje uporabnikov in skrbnikov za delo s programsko opremo	Poročilo o ugotovitvah uvajanja
Prehod na novi sistem	Načrtovalec podatkovne baze/Uvajalec/Skrbnik podatkovne baze/Končni uporabnik/Sistemski administrator/Skrbnik aplikacije/Postavitveni inženir/Poslovni lastnik/Informacijski varnostni inženir	Priprava programske opreme za uporabo v produkcijskem okolju	Poročilo o ugotovitvah izvedbe prehoda na nov sistem

Tabela 2.6: Metode iz metodologije RUP, 5. del [21, 27]

Metoda	Akter	Enota dela	Izdelek
Oblikovanje seznama zahtev	Skrbnik izdelka	Skrbnik izdelka oblikuje seznam zahtev, ki jih podjetje mora doseči	Seznam zahtev
Opisovanje posameznih elementov na seznamu zahtev	Skrbnik izdelka	Skrbnik izdelka doda opis k posamezni zahtevi	Opisi zahtev
Vrednotenje in urejanje posameznih elementov na seznamu zahtev	Skrbnik izdelka	Skrbnik izdelka ovrednoti in uredi po pomembnosti posamezne zahteve	Ovrednotene in urejene zahteve
Razlaga zahtev razvojnim skupinam	Skrbnik izdelka	Skrbnik izdelka razloži posamezne zahteve zaposlenim	/
Načrtovanje izdaje	Skrbnik izdelka/Razvijalci	Skrbnik izdelka skupaj z razvijalci načrtujejo izdajo izdelka	Načrt o izdaji izdelka
Ocenitev časovne zahtevnosti	Razvijalec	Razvijalec oceni časovno zahtevnost posamezne zahteve	/
Razvoj programske opreme	Razvijalec	Razvijalec prevzame posamezno zahtevo in jo realizira	Programska oprema
Vodenje dnevnih sestankov	Skrbnik procesa	Skrbnik procesa vodi dnevne sestanke	/
Oblikovanje sprintov	Razvijalci	Razvijalci oblikujejo sprinte	Načrt sprintov
Izobraževanje zaposlenih o Scrumu	Skrbnik procesa	Skrbnik procesa izobražuje zaposlene o Scrumu	/
Pomoč pri odločitvah ekip	Skrbnik procesa	Skrbnik procesa pomaga, ko skupine potrebujejo pomoč	/
Spodbujanje agilnih metod pri razvoju programske opreme	Skrbnik procesa	Skrbnik procesa spodbuja agilne metode pri razvijanju programske opreme	/
Organizacija aktivnosti razvojnih skupin	Skrbnik procesa	Skrbnik procesa organizira aktivnosti ob pokrivanju aktivnosti posamezne skupine	Poročilo aktivnosti razvojnih skupin

Tabela 2.7: Metode iz metodologije Scrum [28, 29, 30]

Metoda	**Akter**	**Enota dela**	**Izdelek**
Pisanje testov enot	Razvijalec	Razvijalec piše enote testa za lažje testiranje programske kode	Enote testov
Programiranje v paru	Razvijalec	Razvijalci programirajo v paru za hitrejše razvijanje programske opreme	/
Pisanje uporabniških zgodb	Vodja projekta	Vodja projekta napiše uporabniške zgodbe glede na dani projekt	Seznam uporabniških zgodb
Časovno načrtovanje uporabniških zgodb	Vodja projekta	Vodja projekta časovno definira posamezno uporabniško zgodbo	Seznam uporabniških zgodb
Organizacija uporabniških zgodb po prioriteti	Vodja projekta	Vodja projekta organizira uporabniške zgodbe po prioriteti	Seznam uporabniških zgodb
Časovno načrtovanje posamezne iteracije	Vodja projekta	Vodja projekta organizira načrtovanje posamezne iteracije, pri čemer določi koliko, se bo naredilo v iteraciji	Načrt posameznih iteracij
Organizacija dnevnih sestankov	Vodja projekta	Vodja projektov organizira in vodi dnevne sestanke	Načrt dnevnega sestanka
Izvajanje zvezne integracije	Razvijalec	Razvijalec razvije in uporablja zvezno integracijo	/
Oblikovanje standarda za slogovni stil programske kode	Razvijalec	Razvijalci oblikujejo standard, ki definira slogovni stil programske kode	Standard, ki predpisuje slogovni stil programske kode
Definiranje obsega dela	Vodja projekta	Vodja projekta definira obseg dela posamezne iteracije	Poročilo o obsegu dela
Definiranje časovne zahteve za zaključitev projekta	Vodja projekta	Vodja projekta definira, koliko časa je potrebnega za dokončanje iteracije ali projekta	Poročilo z definirano časovno zahtevnostjo projekta

Tabela 2.8: Metode iz metodologije ekstremno programiranje [24, 31]

Metoda	**Akter**	**Enota dela**	**Izdelek**
Uporaba kanban table	Vodja projekta/Razvijalec/Tester programske opreme	Vsi udeleženci uporabljajo kanban table	/
Optimiziranje procesov	Vodja projekta/Razvijalec/Tester programske opreme	Vsi udeleženci ob koncu projekta pogledajo, katere procese je treba optimizirati	Optimiziran proces
Sprememba stanja posamezne aktivnosti	Vodja projekta/Razvijalec/tester programske opreme	Vsi udeleženci, ko opravijo svoje delo, spremenijo stanje in tako naslednja skupina prevzame delo	/

Tabela 2.9: Metode iz metodologije Kanban [25, 32]

Poglavje 3

Analiza in izvedba SME v podjetju

V drugem delu bomo predhodni del uporabili v praksi. V nadaljevanju bomo opisali lastnosti podjetja, v katerem bomo izvedli SME. Po predstavitvi podjetja bo predstavljen intervju z dolgoletnima zaposlenima, ki bosta predstavila okvirno sliko, s katerimi težavami se podjetje spoprijema. Kot drugi način poizvedovanja smo uporabili vprašalnik, s katerim smo povprašali razvijalce v podjetju. Te smo povprašali o stanju procesa razvoja programske opreme in na kak način bi izboljšali proces. Pridobljene informacije zaposlenih so pomembne, saj bodo pomagale pri hitrejšem odkrivanju možnosti za izboljšavo procesa razvoja programske opreme v podjetju. Nato bomo predstavili metodologijo, ki je nastala kot rezultat uporabe SME. Ugotovitve, ki jih bomo pridobili s pomočjo ankete in vprašalnikov, nam koristijo kot vhod za prepoznavo primernih metod iz repozitorija metod. Repozitorij metod je predstavljen v poglavju 2.4.1. Izbrane metode uporabimo za izboljšavo trenutne metodologije. Stranski produkt, ki bo nastal ob razvijanju metodologije, bo dokument, ki bo opisoval ter predpisoval vse akterje, njihove aktivnosti, pristope in vloge v novi metodologiji. Nastali dokument bo postal nekakšen začetni vodič za novozaposlene. Dokument bo predstavil, kako poteka proces razvoja programske opreme v podjetju. Ta predstavitev bo pomagala posamezniku k hitrejši vključitvi v delovanje podjetja, kar bo pomagalo podjetju pri hitrejšem razvoju

programske opreme.

3.1 Predstavitev podjetja

S podjetjem, v katero smo implementirali novo metodologijo, bomo najprej opravili analizo. Po njej bo sledilo načrtovanje metodologije. Podjetje se ukvarja z razvojem programske opreme in izdelavo merilnih naprav. Lastnosti dobrih merilnih naprav in njihove programske opreme so natančnost in hkrati hitrost izvajanja meritev. Z natančnostjo in s hitrostjo meritev je podjetje pred konkurenco. Stranke, ki uporabljajo njihove merilne naprave, so zadovoljne. Podjetje se mora držati časovnih rokov, ko razvija nove funkcionalnosti ali ko odpravlja napake v programski opremi. Podjetje ima zaposlenih več kot 70 ljudi. V takšnem podjetju mora biti prisotna dobra komunikacija znotraj ekipe in med ekipami.

3.2 Analiza procesa razvoja programske opreme v podjetju

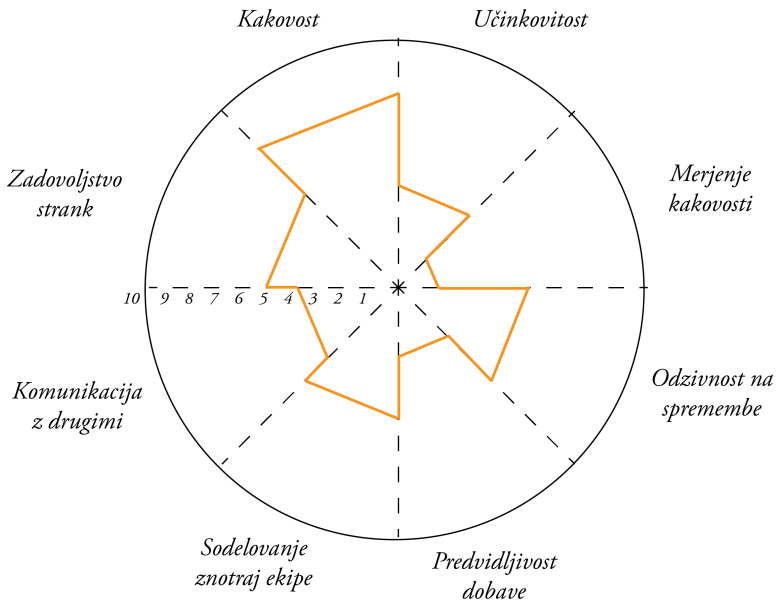
Namen analize procesa razvoja programske opreme je ugotoviti, kje v procesu razvoja programske opreme ali merilnih naprav prihaja do ozkih grl, kar oteži in upočasni razvoj. Za vizualno predstavitev stanja procesa razvoja programske opreme je bilo uporabljeno kolo ravnovesja. To je graf, namenjen prikazovanju vrednosti več spremenljivk hkrati. Vsaka je predstavljena s svojo osjo. Vse osi izvirajo iz iste točke in so označene z vrednostimi od nič do deset. S kolesom ravnovesja smo pokazali, kakšne ocene o posamezni spremenljvki so podali delavci v podjetju, ki smo jih vključili v razgovore oziroma smo jih anketirali [33]. Pri kolesu ravnovesja bodo uporabljene lastnosti/dimenzije, ki so jih izbrali vodje ekip razvoja programske opreme v izbranem podjetju. Izbrali so merila, ki jih uspešno podjetje mora imeti [34]. Izbrane dimenzije so:

- kakovost;
- učinkovitost;

- odzivnost na spremembe;
- predvidevanje roka dobave;
- sodelovanje znotraj ekipe;
- komunikacija z drugimi;
- zadovoljstvo strank;
- ?.

Dimenzija *kakovost* predstavlja stanje kakovosti izdelkov, ki jih podjetje razvija. Dimenzija *učinkovitost* predstavlja, kako učinkovito je podjetje pri razvijanju produkta. Dimenzija *odzivnost na spremembe* predstavlja, kako hitro je podjetje sposobno implementirati nove funkcije/funkcionalnosti v produkt. *Predvidevanje dobave* predstavlja sposobnost podjetja napovedati, koliko časa bo potrebnega za implementacijo neke nove funkcionalnosti v obstoječo programsko okolje ali v merilno napravo. Dimenzija *sodelovanja znotraj ekipe* predstavlja kakovost komunikacije med sodelavci in kako hitro lahko dosežejo neki zaključek v komunikaciji. Dimenzija *komunikacija z drugimi* predstavlja sposobnost podjetja komunikacije z zunanjimi sodelavci in izvajalci. Predzadnja dimenzija je *zadovoljstvo strank*, ki predstavlja mnenja zaposlenih, kako so stranke zadovoljne s podjetjem. Zadnja dimenzija je označena z *?* in predstavlja dimenzijo, ki jo poda vsak posameznik. Ta dimenzija mora biti pomembna posamezniku. Dodatna dimenzija ne sme biti povezana z dimenzijami, ki so že na kolesu ravnovesja. Vsak, ki rešuje kolo ravnovesja, označi na lestvici od 1 do 10, na kakšnem nivoju (po njihovem mnenju) je posamezna dimenzija v podjetju. Da bomo dobili čim boljši vpogled v podjetje, je bilo kolo ravnovesja podano več zaposlenim. Z menedžerjem in vodilnim razvijalcem je bil narejen intervju. Z razvijalci programske opreme pa je bila opravljena anketa za hitrejšo pridobitev informacij. Rezultat rešenih koles ravnovesij so na slikah 3.1, 3.2, 3.3.

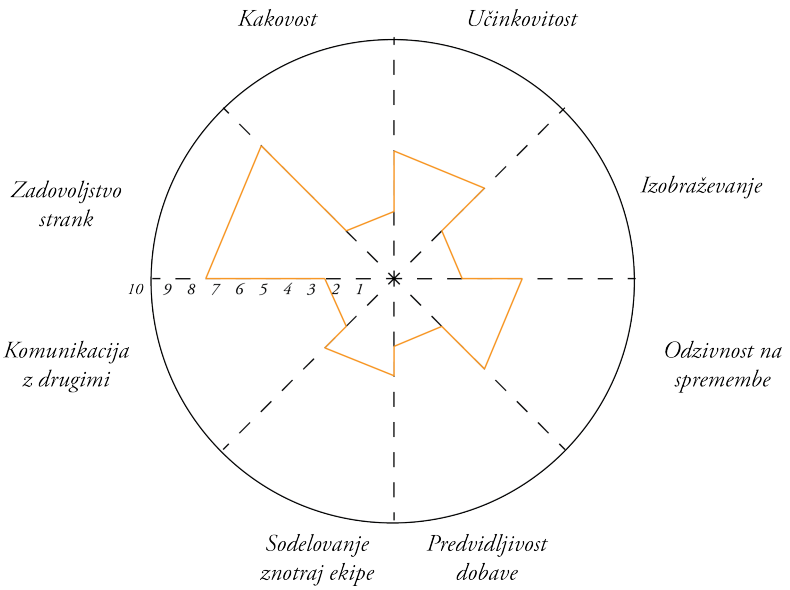
Slika 3.1 prikazuje kolo ravnovesja, ki ga je izpolnil menedžer. Če jo pogledamo



Slika 3.1: Kolo ravnovesij, ki ga je izpolnil menedžer

vidimo, da je menedžer zadovoljen s kakovostjo podjetja, kar pomeni, da z njegovega vidika kakovost ni ogrožena. Učinkovitost je označena na nivoju tri, kar pomeni, da bodo potrebne spremembe na dimenziji učinkovitost. Dimenzija zadovoljstva strank je na nivoju 4, kar pomeni, da je podpovprečna in da bo potrebna natančnejša analiza. Prav tako sta podpovprečni dimenziji komunikacija z drugimi in predvidljivost dobave. Na enakem nivoju sta dimenziji sodelovanje znotraj ekipe in odzivnost na spremembe. Dodatna dimenzija, ki je bila podana, je bila *merjenje kakovosti*, ki predstavlja vrednosti, s katerimi bi lahko prikazovali uspešnost in lahko tudi pozneje primerjali vrednosti z drugimi obdobji.

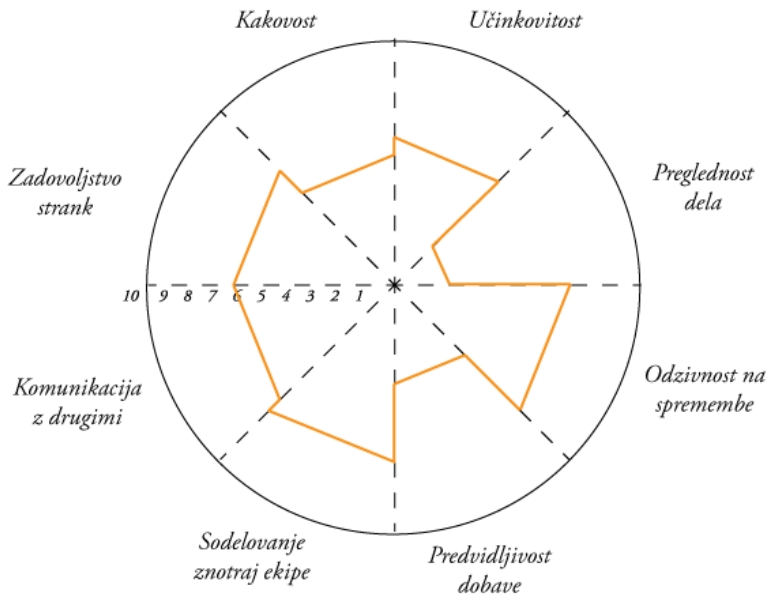
Slika 3.2 prikazuje kolo ravnovesja, ki ga je izpolnil vodilni programer. Druga slika prikazuje drugačno sliko, kot jo prikazuje kolo ravnovesja, ki ga je izpolnil menedžer. V primerjavi z menedžerjem je bolj kritična. Na enakem nivoju so dimenzije učinkovitost, odziv na spremembe in sodelovanje znotraj ekipe. Programer v podjetju je kritičen do kakovosti, predvidljivosti dobave in do komunikacije z drugimi. Kot dodatna dimenzija, ki smo jo dodali pri programerju, je izobraževanje.



Slika 3.2: Kolo ravnovesij, ki ga je izpolnil vodilni programer

Opredeljeno je kot učenje novih znanj in udeležba na sejmih na področju razvoja programske opreme.

Poleg intervjuja z vodilnim razvijalcem je bila prav tako opravljena anketa. Anketirani so bili razvijalci. Z anketo smo pridobili vpogled v to, kako, razvijalci vidijo probleme, saj vodja vedno ne vidi vseh problemov. Kolo ravnovesja od razvijalcev je vidno na sliki 3.3. Ocene predstavljajo povprečje ocen vseh razvijalcev, ki so bili anketirani. Slika prikazuje ujemanja in prav tako razlike med vodilnimi in drugimi razvijalci. Kolesa ravnovesja vodilnega razvijalca in drugih razvijalcev se ujemajo v učinkovitosti, predvidljivosti dobave in v odzivnosti na spremembe. Podobnosti nam povedo, da vodja razvijalcev vidi probleme, ki jih vidijo tudi razvijalci. Največ razlik nastaja v dimenzijah komunikacije z drugimi, sodelovanje znotraj ekipe in kakovosti. Ta razlika pomeni, da razvijalci delujejo kot skupina. Te razlike so pomembne za intervju. Razlike nam podajo informacijo o problemih v dimenziji. Ti problemi obstajajo, čeprav jih vodje ne opazijo. Iz koles ravnovesja, ki sta jih rešila izvajalec in menedžer ter z anketiranjem razvijalcev smo dobili



Slika 3.3: Kolo ravnovesij, ki je rezultat anket

sliko, kje v podjetju so problemi. Glede na odkrite probleme so bila oblikovana vprašanja, ki bodo odkrila globino problemov, kar bo pomagalo pri oblikovanju nove metodologije. V nadaljevanju bo predstavljen intervju z menedžerjem, nato pa bo predstavljen še intervju z vodilnim programerjem.

3.2.1 Primerjava koles ravnovesij

S primerjanjem koles ravnovesij dobimo informacijo o tem, s katerimi problemi se spoprijema posamezna skupina. Če primerjamo menedžerje in programerje, opazimo kar nekaj razlik v kolesu ravnovesja. Če primerjamo kolo ravnovesja, ki predstavlja programerje, in kolo ravnovesja menedžerja, vidimo, da do največjih razlik pride pri kategoriji kakovost. To razložimo s tem, da menedžer nima tako dobrega vpogleda v kakovost izdelkov, kot ga imajo posamični programerji. Nasprotje se pokaže pri zadovoljstvu strank, v kar razvijalci nimajo vpogleda, menedžerji pa ga imajo. Pri primerjanju drugih kategorij opazimo, da ni veliko sprememb, kar pomeni, da se lahko oblikujejo podobna vprašanja, saj vidijo podobne probleme v podjetju. Do zdaj smo primerjali menedžerja in programerja, ki je glavni progra-

mer. Zdaj pa bomo primerjali glavnega programerja in posamezne programerje. Čeprav si lahko mislimo, da ne bo razlik, se na kolesu ravnovesja te vidijo, kar pomeni, da so prav tako vidiki, ki jih vidi glavni programer in posamični programerji, različni. Največje razlike prihajajo pri dveh dimenzijah. Ti sta komunikacija z drugimi in sodelovanje znotraj ekipe. To pomeni, da si vodja programerjev želi še večje sodelovanje znotraj ekipe in komunikacije z drugimi, čeprav so razvijalci v podjetju že zadovoljni s trenutnim stanjem. V preostalih dimenzijah ni velikih razlik, kar je pozitivno, saj to pomeni, da ima vodja programerjev dober vpogled v stanje med programerji. S primerjavo koles ravnovesja smo pridobili vpogled v podjetje, ki nam služi kot osnova za oblikovanje vprašanj, ki jih bomo uporabili v intervjuju z menedžerjem in glavnim programerjem. V nadaljevanju bosta intervjuja z menedžerjem in vodilnim programerjem.

3.2.2 Intervju z menedžerjem

Ker je bila dimenzija predvidljivosti dobave ocenjena najslabše, smo začeli z vprašanjem: "Kaj je razlog za nezadovoljstvo s predvidljivostjo dobave?". Menedžer je odgovoril, da so problemi v tem, da ni natančne organizacije v povezavi s tem, kaj mora nekdo narediti ob določenem času in katere naloge so pomembne. Naslednje vprašanje je bilo: "Kako trenutno programerji določijo, katere naloge so pomembne in katere niso?". Menedžer je odgovoril, da nimajo organizirane določitve, katere naloge imajo prioriteto; tako programerji rešujejo vse naloge, ki so jim podane. Tako nikoli niso rešene nekatere naloge, ker so vedno potisnjene v ozadje, da jih rešijo pozneje. Dodatno je še pojasnil: "Predlagal bi, da bi imeli nekakšen letni načrt, na katerem bi za eno leto vnaprej določili, na kaj se bomo osredinili v prihodnjem letu in koliko časa bi porabili za določeno aktivnost." Nato smo ga vprašali: "Kdo potem dodeljuje delo programerjem?". Menedžer je odgovoril, da stranke neposredno naslovijo razvijalce prek e-pošte, in dodal: "Ker rešujemo problem za tuje stranke, se velikokrat zgodi, da stranke enako e-pošto pošljejo več razvijalcem, kar ustvari zmešnjavo; ne ve se, kateri razvijalec je odgovoren za določeno zadevo in kaj mora posamezni razvijalec narediti." Menedžer je pojasnil, da ker se vse nadaljuje prek e-pošte, nastane dolga (zamudna) veriga izmenjanih sporočil, kar

oteži pregled in delo razvijalcev.

Potem smo vprašali: "Dimenziji komunikacija z drugimi in sodelovanje znotraj ekipe ste označili zelo nizko, kje je razlog za to?". Menedžer je odgovoril: "Dimenziji sem označil slabo, ker zaradi slabega organiziranja dela pride to tega, da se razvijalci med seboj blokirajo. To pomeni, da recimo en razvijalec čaka, da drugi razvijalec nekaj (do)konča. Ker drugi razvijalec ne more dokončati tega dela, ker prihajajo drugi problemi, mora prvi razvijalec na njega počakati. To čakanje povzroči, da se izdelki pozneje dostavijo strankam." Nato smo vprašali: "Kako obravnavate zakasnitve?". Menedžer je odgovoril: "Zakasnitve žal vidimo prepozno, saj nimamo nekega tekočega sledenja projektom. Ker ni sledenja projektom, ne moremo predvidevati zakasnitve, kar povzroči veliko problemov s strankami pa tudi z vodenjem."

Nato smo vprašali: "Kaj bi izboljšali za boljšo učinkovitost v podjetju?". Menedžer je odgovoril: "Glavni problem za neučinkovitost v podjetju je pomanjkanje nadzora nad tem, kaj kdo naredi in koliko časa še potrebuje za določeno delo. S tem, ko bi sodelavci videli, koliko časa bo treba čakati na določeno blokado, si lahko potem bolje organizirajo čas in tako poskrbijo za preostala dela. To bi povečalo učinkovitost podjetja." Menedžer je še dodal: "Določene zaposlene bi razbremenil oziroma bi jih razdelil na skupine. S tem bi imel manjše probleme s kopičenjen z obveznosti, saj bi se razdelile med člani skupine."

Nato smo vprašali: "Dimenzijo odzivnost na spremembe ste označili povprečno. To pomeni, da ste zadovoljni?" Menedžer je odgovoril: "Podjetje se hitro odzove, če pride do sprememb. Kadar stranka javi problem in če problem hitro pride do njega, je ta rešen zelo hitro. Problemi nastanejo, kadar naši prodajalci prodajajo zadeve, ki še niso bile analizirane, in tako ni nekega občutka, kako hitro lahko rešimo neki problem. Takšne zadeve zelo otežijo delo razvijalcem, saj se morajo spoprijemati z novo zadevo, za katero niso imeli časa narediti analize, kar pripelje do večje zakasnitve."

Nato smo vprašali: "V podjetju imate poleg razvijalcev tudi pomoč uporabnikom. Kakšna je njihova naloga in kako pomagajo razvijalcem?" Menedžer je odgovoril: "Pomoč uporabnikom je bila nujna, saj so bili razvijalci zasuti z e-pošto, v kateri so stranke želele nove funkcionalnosti in popravke napak. Pomoč uporabnikom je "prva bojna linija", ki pomaga razvijalcem, da niso preobremenjeni s problemi. Vendar je problem pri pomoči enak kot pri razvijalcih. Vsa komunikacija poteka prek e-pošte, kar upočasni in oteži delo. Na koncu, skozi celotno verigo e-pošte, razvijalec ne ve, ali vse deluje."

Nato smo vprašali: "Kot dodatno dimenzijo ste si izbrali merjenje kakovosti. Kaj ste mislili z njo in zakaj je označena tako nizko?" Menedžer je odgovoril: "Z merjenjem kakovosti sem želel dobiti vpogled, kako hitro se rešijo nekateri problemi. Le tako bi lahko primerjal rezultate iz preteklih let in videl, ali smo kot podjetje napredovali." Nato smo vprašali, katere vrednosti bi ga zanimale. Menedžer je odgovoril, da bi želel rezultate o tem, kako hitro so rešeni problemi v programski kodi. Dodal je: "S tem bi dobil vpogled v to, kako hitro so rešeni problemi določene vrste, in tako dobil okvirno oceno časovne zahtevnosti posameznega problema. Poleg tega bi me zanimale časovne vrednosti posameznega projekta. Tako bi videl, koliko časa je potrebnega za dokončanje posameznega projekta. Razlog je enak kot pri napakah v programski kodi."

3.2.3 Intervju z vodilnim razvijalcem

Če pogledamo kolo ravnovesja, ki ga je rešil vodilni razvijalec, opazimo, da so bile štiri dimenzije zelo nizko označene. S temi dimenzijami smo začeli intervju. Vprašali smo: "Začeli bomo z eno izmed dimenzij, ki ste jo označili najnižje. Zanima nas, zakaj je dimenzija kakovosti tako nizka." Razvijalec je odgovoril, da je razlog v pretežno neorganiziranem delu. Dodal je: "Stranke nas kontaktirajo prek e-pošte, in ker so iz tujine, se velikokrat zgodi, da ne razumemo, kaj si želijo, in tako pride do izdelkov s slabšo kakovostjo." Ker smo želeli razumeti izpostavljeno dilemo, smo preverili svoje razumevanje, in sicer to, ali je glavni razlog za nizko

kakovost v zamudni komunikaciji s tujimi strankami. Razvijalec je pojasnil: "Ne povsem. Problem je, da kot razvijalci ne moremo slediti celotni komunikaciji, ki poteka prek e-pošte. To nam oteži razumevanje problema in tako izgubljamo čas s tem. Poleg tega, ker imamo tako veliko zahtev, nimamo jasno zastavljeno, katere zahteve imajo prioriteto pred drugimi."

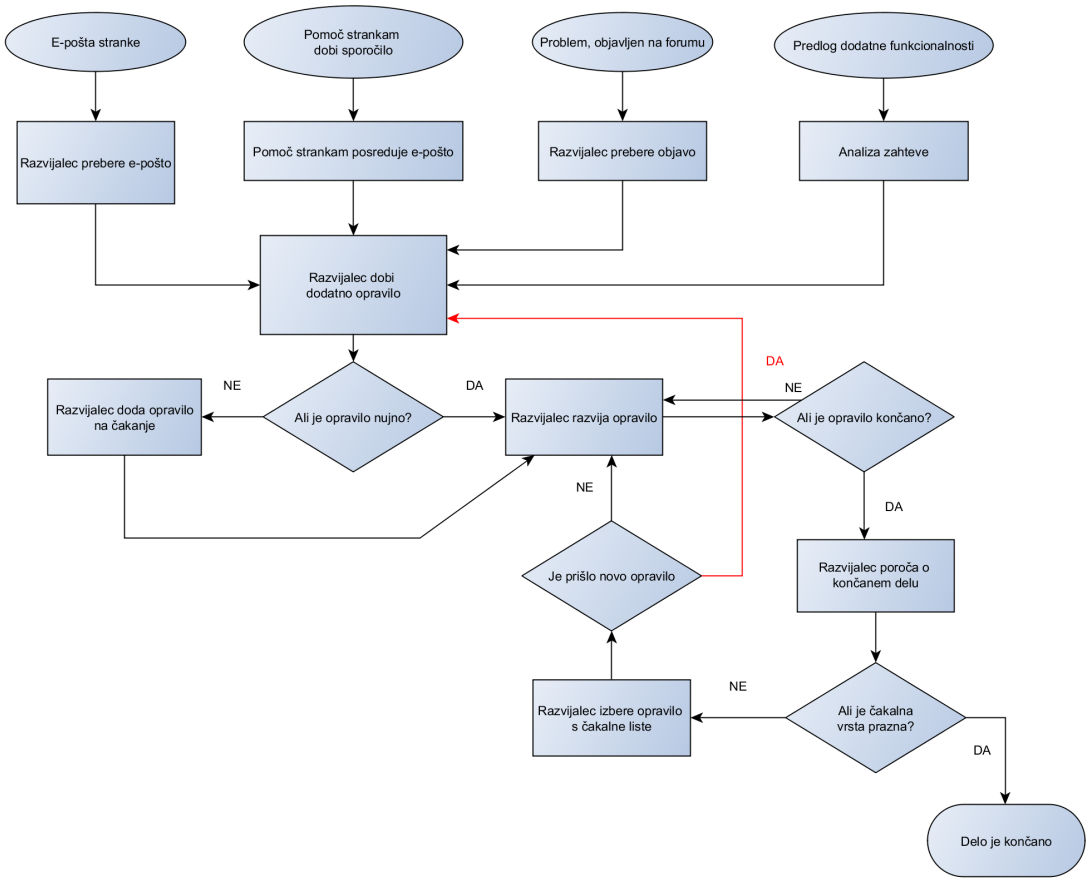
Nato smo vprašali, ali se poslužujejo sestankov. Razvijalec je odgovoril: "Sestanek je vsak ponedeljek, vendar nima neke strukture, tako da bi bil lahko njegov namen bolj izkoriščen. Na sestanku je preveč govora o zadevah, ki niso pomembne, in tako niso rešeni problemi, s katerimi bi se morami spoprijemati." Nadaljevali smo z vprašanjem: "Čemu bi vi namenili pozornost na sestanku?". Razvijalec je odgovoril: "Želim si, da bi na sestanku dobili vpogled nad stanjem preostalih projektov. S tem bi odkrili napake, ki lahko povzročijo zastoje v razvoju programske kode. Hitreje kot se odkrijejo te napake, hitreje se lahko nadaljuje razvoj projekta. Poleg odkritja napak je pomembno vedeti, na katerem projektu je posamezen razvijalec in v kolikšni meri je končal projekt."

Nato smo vprašali: "Glede na to, da isti projekt razvija več razvijalcev, nas zanima, kako poteka zaključek projekta.". Razvijalec je pojasnil: "Na zelo zanimiv način. Naša koda ni testirana sproti. To pripelje do neustreznosti programske kode ob združitvi. Neustreznost povzroči veliko zmešnjavo, saj moramo odpraviti vse napake v kratkem časovnem roku. Namesto da bi se testi izvajali vsak dan oziroma na kratko časovno obdobje, se izvajajo, ko je projekt končan. Enak rezultat se zgodi v primerih, ko si stranka želi spremembe v kodi. Takrat se velikokrat zgodi, da projekt ne prestane testov in nastanejo zakasnitve. Ker morajo biti napake odpravljene hitro, so rešitve teh problemov 'slabe', kar povzroči slabšo kodo. Ker je slabša koda, je posledično tudi težje ohranjati delujočo." Nato smo vprašali, ali testirajo projekte pri strankah. Razvijalec je odgovoril, da njihove stranke testirajo delovanje, vendar se tega ne zavedajo. To pomeni, da ne vedo, da uporabljajo programsko opremo, ki je v stanju testiranja. To povzroča napačna mnenja o programski opremi.

Nato smo vprašali: "Poleg programske opreme razvijate tudi merilne naprave. Kako programska oprema merilnih naprav deluje s programsko opremo na računalniku?" Razložil je: "Ker se programska oprema testira posebej od merilnih naprav, velikokrat pride do napak, ko se naprava priključi in testira. Zato je zmeraj 'zanimivo' videti, ali bo programska oprema prestala priključitev naprav."

3.2.4 Proces razvoja programske opreme, prikazan z diagramom poteka

Na sliki 3.4 vidimo proces razvoja programske opreme za posameznega razvijalca. Ta proces je bil povzet po opisih vodilnega razvijalca. Razvijalec lahko pridobi novo opravilo iz treh virov: foruma, e-pošte ali od oddelka za pomoč strankam. Opravilo predstavlja popravljanje napak na programski opremi ali dodajanje novih funkcionalnosti. Ko je delo predano razvijalcu, je najprej obravnavana njegova pomembnost. Če je delo pomembno, se razvijalec takoj loti dela in prekine delo na predhodni obveznosti. Če delo ni pomembno, je dodano na seznam obveznosti. Razvijalec začne razvijanje z razvijanjem obveznosti. Ko je opravilo končano, poroča o končanem delu. Če je izpraznil čakalno listo, je končal delo, sicer razvijalec izbere drugo delo s čakalne liste. Nato preveri, ali je prišlo novo opravilo. Če je prišla nova obveznost, se ta doda na seznam obveznosti. V takšnem zaporedju je potekal razvoj programske opreme.



Slika 3.4: Trenutni proces razvoja programske opreme

3.3 Analiza intervjujev in anket

Naslednji korak po razgovorih in anketah je analiza intervjujev in anket ter odkrivanje slabosti. Glede na podobnost problemov so se odkrile tri pomanjkljivosti. Prva skupina so slabosti pri sprejetju in obravnavi novih zahtev. Druga skupina so slabosti glede testiranja programske kode. Zadnja skupina so slabosti glede oddelka za pomoč strankam. V nadaljevanju bodo opisane vse slabosti podrobneje.

3.3.1 Pomanjkljivosti pri sprejetju in obravnavi novih zahtev

Skozi razgovor z menedžerjem in razvijalcem se je odkrilo kar nekaj slabosti. Prva slabost je bila, da stranke neposredno komunicirajo z zaposlenimi, kar povzroči zmešnjavo na strani razvijalcev. Veliko pomanjkljivosti glede programske opreme je prišlo k zaposlenim na e-pošto. To je povzročilo, da so se poročila o pomanjkljivosti programske opreme nabirala pri razvijalcu. Tu se je pojavila slabost glede sledenja zahtev strank. Iz takšne množice pomanjkljivosti programske opreme je bilo težje razvidno, katere stranke imajo prioriteto, in tako so morali razvijalci sami presoditi prioritete, kar je tretja slabost. Ker prioritete niso določene, nekatere stranke nikoli ne pridejo v obravnavo. Ker ni bilo postavljenih prioritet, razvijalci niso vedeli, da morajo določene zadeve razviti prednostno. Zadnje pripelje do tega, da morajo razvijalci čakati drug na drugega, kar povzroča zakasnitve. Glavni razlog za to je slaba sledljivost posamezni zahtevi. Če bi bila sledljivost na primerni ravni, bi si lahko razvijalci med seboj pomagali ali ugotovili, da bo prišlo do zakasnitev, in to pravočasno sporočili nadrejenim. Eden izmed razlogov za slabšo sledljivost so neorganizirani sestanki v podjetju. V celoti vidimo slabosti glede sprejemanja zahtev in realizacije sprememb.

3.3.2 Pomanjkljivosti pri nudenju pomoči uporabnikom

Po pogovoru z menedžerjem je bilo razvidno, da imajo probleme pri zagotavljanju pomoči uporabnikom. Slabost, ki se je opazila, je, da ni skupne kontaktne

točke. To pomeni, da lahko stranke kontaktirajo pomoč strankam na več načinov. Skupna kontaktna točka je lahko telefonska številka ali e-poštni naslov oz. informacijski sistem. Glede na to, da podjetje rešuje različne probleme, kot je pomoč pri praktičnih problemih ali odkrivanju napak v programski kodi ter zapisovanje novih funkcionalnosti, bi bil potreben informacijski sistem, ki je sposoben vse probleme zajeti. Naslednji problem, ki je bil opažen, je problem slabe sledljivosti. Ko pride poročilo o napaki na oddelek za pomoč strankam, to napako zaposleni analizirajo in jo sporočijo razvijalcu. Od trenutka, ko je poročilo o napaki posredovano razvijalcu, oddelek za pomoč strankam nima več vpogleda v stanje problema. Kot stanje si lahko predstavljamo, ali je razvijalec že začel na popravljanju napake ali je samo na analizi napake.

3.3.3 Pomanjkljivost postopka testiranja

Iz pogovora z razvijalcem je bilo razvidno, da je testiranje programske kode slabo izvedeno. Če bi se programska oprema testirala sproti, bi bila verjetnost napak na zaključku projekta veliko manjša. S sprotnim testiranjem bi se napake popravile pred roki oddaje programske opreme. Ko razvijalci popravljajo kodo, se kakovost kode vidno poslabša, kar povzroči, da se s časom pojavi še več napak, ki pa jih je težje odpraviti. To povzroči neujemanje programske opreme z napravami. Merilne naprave imajo svojo programsko opremo. Če je programska oprema na računalniku polna napak, bo priključitev naprave povzročilo še več napak. Ker programska koda ni testirana v celoti, nekatere napake prispejo do strank. Stranke opravljajo testiranje programske kode in odkrijejo napake v programski kodi. S tem dobijo mnenje, da je programska koda slaba.

3.4 Analiza procesa razvoja programske opreme

Iz vseh slabosti, ki so bile odkrite, je razvidno, da je problem na več področjih v procesu razvoja programske opreme. Prvo področje je pri organizaciji zaposlenih. Naslednja slabost je problem organizacije pri dodeljevanju nalog. Sledijo slabosti v testiranju in nazadnje se pojavijo tudi slabosti v organizaciji za pomoč strankam.

Slabosti so enako pomembne, ker se prepletajo med seboj. V poglavju 2.4 smo predstavili repozitorij metodologij. Ker je podjetje prilagodljivo in si želi prilagodljive metodologije, imajo agilne metodologije prednost pred drugimi. Najprej bomo pregledali metode v repozitoriju. Poskušali bomo ugotoviti, ali je katera izmed izbranih metod ustrezna za izboljšavo procesa. Izbrane bodo tiste metode, ki bodo izboljšale slabosti v procesu razvoja programske opreme.

3.4.1 Primerne metode za reševaje slabosti pri sprejetju in obravnavi novih zahtev

Za reševanje slabosti pri sprejetju in obravnavi novih zahtev obstaja nekaj metod iz različnih metodologij. Če pogledamo iz metodologije Scrum, vidimo, da je skoraj celotna metodologija sestavljena iz elementov, ki predstavljajo metode, ki so namenjene izboljševanju organizacije dela. S seznamom zahtev predstavimo, katere naloge bo treba izvesti za doseganje posameznega cilja. Pomemben del metodologije Scrum so sprinti, ki trajajo od dveh do štirih tednov. V tem času morajo razvijalci ustvariti izdelek ali neko dodatno funkcionalnost. Poleg posameznih sprintov pa imajo tudi dnevne sestanke, na katerih razvijalci in vodje razvojnih skupin govorijo o napredku projekta. Posamezne naloge si razdelijo člani skupine med seboj. Podobne metode, kot jih ima Scrum, vsebuje tudi metodologija Kanban. Pri Kanbanu v začetku obdobja postavijo cilje, ki jih morajo doseči. To so tako imenovani globalni cilji, ki so sestavljeni iz manjših obveznosti. Katere obveznosti se bodo izvedle, je odvisno od dnevnega sestanka. Na njem določijo, katere obveznosti imajo večjo prioriteto oziroma katere obveznosti se lahko ignorirajo oz. dajo v ozadje [5]. To je velika prednost, ker imajo razvijalci seznam dejavnosti, ki jih morajo opraviti. Tako razvijalci natančno vedo, katero delo morajo opraviti. Pri Kanbanu si vsak razvijalec izbere delo sam. Glavna prednost Kanbana pred Scrumom je stalni vpogled v trenutni tok dela [28]. To pomeni, da se vidi točno, kaj kdo dela in koliko časa opravlja to delo. S tem vpogledom se odkrijejo ozka grla v procesu. S prepoznavo ozkih grl lahko optimiziramo proces. Če primerjamo Kanban in Scrum, se razlikujeta v tem, da Kanban nima tedenskih časovnih omejitev. Ekstremno programiranje nima točno definiranih metod, ki bi

pomagale pri organizaciji dela zaposlenih.

3.4.2 Primerne metode za reševanje slabosti pri testiranju

Metode za testiranje so najbolje predstavljene v metodologiji ekstremno programiranje. Ta predstavlja več priporočil, ki so povezana s testiranjem. Najbolj poznana sta pisanje testov enot in stalna integracija. S tem poskrbimo, da je kvaliteta kode boljša in je tako manjša možnost napak. Drugo priporočilo, ki ga predpisuje ekstremno programiranje, je programiranje v paru. S tem, ko dva razvijalca razvijata isti problem, se verjetnost napak zmanjša in prav tako se tudi pozneje lažje napišejo testi, ki testirajo napake. Kot alternativa programiranju v parih je pregledovanje kode (Angl. "Code review"). To opravlja razvijalec, ki ni razvijal programske kode. S tem se odkrijejo napake ali ideje kako, izboljšati programsko opremo. Na dolgi časovni rok to pripelje do boljše programske kode. Kot naslednje priporočilo, ki ga predpisuje ekstremno programiranje, je stalno izboljševanje kode (Angl. "Refactoring"). To pomeni da se razvijalec na začetku projekta trudi napisati čim boljšo kodo. Toda z povečevanjem projekta postane potreba po prestrukturiranju programske kode večja. [35]. Ena izmed rešitev, da se pojavlja manj napak, je določevanje standardov pisanja kode. S tem, ko ima celotno podjetje določeno, kako pisati programsko kodo in se drži teh standardov, je koda poenotena in lažja za razumevanje.

3.4.3 Primerne metode za reševanje slabosti pri nudenju pomoči strankam

Za reševanje teh slabosti uporabimo kombinacijo že zgoraj navedenih metod. Na primeru vpogleda v delo drugih, ki je določen v Kanbanu, je uporabno ravno pri pomoči uporabnikov, saj vidijo, koliko obveznosti ima določen razvijalec in koliko časa bo potrebnega za popravilo določene zadeve. S tem bo stranka pridobila takojšnjo informacijo o tem, kdaj bo rešen njihov problem in v kateri različici bo prišlo popravilo oz. nadgradnja. Pri nudenju pomoči uporabnikom je vodja razvijalcev odgovoren za natančen opis problema. Z natančnim opisom problema

je lažje predstaviti, kaj je jedro problema, kakšna je zahtevnost odprave problema in koliko časa bo potrebnega za odpravljanje napake [29] [36].

3.4.4 Izbrane metode za reševaje slabosti pri sprejetju in obravnavi novih zahtev

Glavna sprememba, ki bo pomagala pri zboljševanju procesov, so dobri sestanki na kratka obdobja. Zato bomo iz metodologije Scrum uporabili koncept dnevnih sestankov. Na sestankih povedo, kaj so naredili včeraj, kaj bodo naredili danes in s katerimi problemi se spoprijemajo. Če bi se natančno držali metodologije, bi morali predpisati dnevne sestanke, vendar v podjetju ni bilo potrebe po njih. Večino projektov je bilo mogoče spremljati tedensko. Zato so bili dnevni sestanki preoblikovani v tedenske sestanke.

Drugi element, je načrtovanje opravil. Ta je definiran z delovnim letnim načrtom. V letnem načrtu je napisano, kaj se bo v tem letu naredilo in kako bo delo med letom potekalo. Letni načrt predstavlja *izdajo* v metodologiji Scrum. Tako je izdaja izdelka dolga dvanajst mesecev. Posamezen sprint pa je dolg en mesec. V letnem načrtu piše kaj se mora v posameznem mesecu narediti.

Ob koncu leta se naredi analiza, kateri cilji so bili doseženi in, kaj bi se lahko še izboljšalo. Ta element je bil izpeljan iz metodologije Kanban, za katerega je pomembno, da se po vsaki iteraciji optimizirajo procesi. Izboljšan proces uporabimo v naslednji iteraciji.

Za boljšo organizacijo na ravni razvijalcev so potrebne skupine. Zato iz metodologije Scrum uporabimo koncepte razvojne skupine. Vsaka skupina ima vodjo, ki skrbi za vodenje, določi kaj bo kdo naredil in prioriteto posameznih aktivnosti dela. Vodja je izkušen razvijalec, ki ve, komu dati delo in koliko časa lahko pričakuje, da se bo neka obveznost reševala. Prav tako ima zmožnost predvidevanja, kje se lahko zgodijo napake oziroma ozka grla. Sposoben je strokovno pomagati. Lahko tudi prosi druge zaposlene ali pa sam reši problem. Vodja mora

imeti vedno dober vpogled, kaj njegova skupina dela. Če se pojavi blokada (eden izmed članov skupine čaka na drugega), ga vodja zaposli z drugim delom, ki bo časovno trajalo toliko časa, kolikor bo potreboval član za odpravo blokade. Vpogled, v katerem stanju je posameznik, pa ne sme biti omejen samo na voditelja. Vsak razvijalec si lahko ogleda, pri katerem stanju je drugi razvijalec. Tako si lahko organizira svoje delo. S takšnim pristopom se bo delo lažje nadaljevalo. Ob tem pa bodo rešeni tudi drugi problemi. V intervjuju je bil odkrit problem, da tuji prodajalci prodajajo nerazvite funkcije, ki še niso bile analizirane. Ker ima vodja skupina največ izkušenj, lahko oceni, ali je mogoče funkcionalnost izvesti v časovnem roku. Če to ni mogoče, mora opozoriti takoj, da se proces pravočasno ustavi.

3.4.5 Izbrane metode za reševanje slabosti pri testiranju

Iz intervjuja se je opazilo, da je področje testiranja zelo slabo. Zato iz metodologije ekstremno programiranje uporabimo metodo izdelave testov.

Za izboljšavo stanja programske kode je bila uporabljena metoda uporabe stalne integracije, ki je prav tako prisotna v ekstremnem programiranju. Z njo poskrbimo, da se testi na programski opremi izvajajo redno. Redno pomeni, da se testi izvajajo ob vsaki spremembi ali intervalno na vsako uro. Z rednim testiranjem razvijalci odkrijejo napake hitro in tako ne nastanejo problemi ob izidu programske opreme. Odgovornost testerjev je, da se vedno doda več in več testov, s katerimi preprečijo napačno obnašanje aplikacije.

Problem, ki se je pojavil v intervjuju, je problem mrtve kode. To je koda, ki se ne uporablja. Prisotna je le zaradi zgodovinskih razlogov. Takšna koda povzroča probleme na dolgi rok, saj poveča težavnost kode. Takšnim problemom se izognemo s pregledovanjem kode, ki ga predpisuje ekstremno programiranje. Pri pregledovanju kode sodelavec oceni kodo, ki je bila napisana. Že s tem, da v kodo pogleda neki drugi razvijalec in o njej razmišlja, je odraz tega, ali je koda napisana dobro.

Pri ocenjevanju kode se je treba držati navodil. Za to metodologija ekstremno programiranje vsebuje metodo oblikovanja standarda za programsko kodo. Ta navodila opisujejo, kako mora biti koda napisana in kakšnim stilom mora ustrezati. Tak dokument z navodili morajo oblikovati vodje skupin v podjetju.

3.4.6 Izbrane metode za reševanje slabosti pri nudenju pomoči strankam

Iz intervjuja je razvidno, da je pomoči uporabnikom posvečeno premalo pozornosti. Namen pomoči uporabnikom je vez med strankami in razvijalci. Naloga pomoči uporabnikom, je, da od stranke dobi vse informacije glede napak ali novih funkcionalnostih, ki si jih stranke želijo. Te informacije so posredovane so vodjem skupine. Ta pa se lažje odloči, ali bo sprejel posamezno funkcionalnost, in določi prioriteto posamezne napake. Ko vodja skupin določi prioriteto, pomoč strankam dobi povratno informacijo o tem. S to informacijo lahko odgovorijo strankam, koliko časa bo potrebno za realizacijo ali odpravo napake. Tukaj je uporabljena metoda vpogleda v dejavnosti iz metodologije Kanban. Poleg posredovanja napak in dodatnih funkcionalnosti je pomoč uporabnikom pomembna za posredovanje mnenj iz strani strank. Stranke najbolje vedo, kaj si želijo, ter tudi najbolje poznajo programsko in merilno opremo. To je najboljša testna skupina, ki lahko poda verodostojne komentarje, ki so lahko v veliko pomoč.

3.4.7 Dokument vlog in procesa razvoja programske opreme

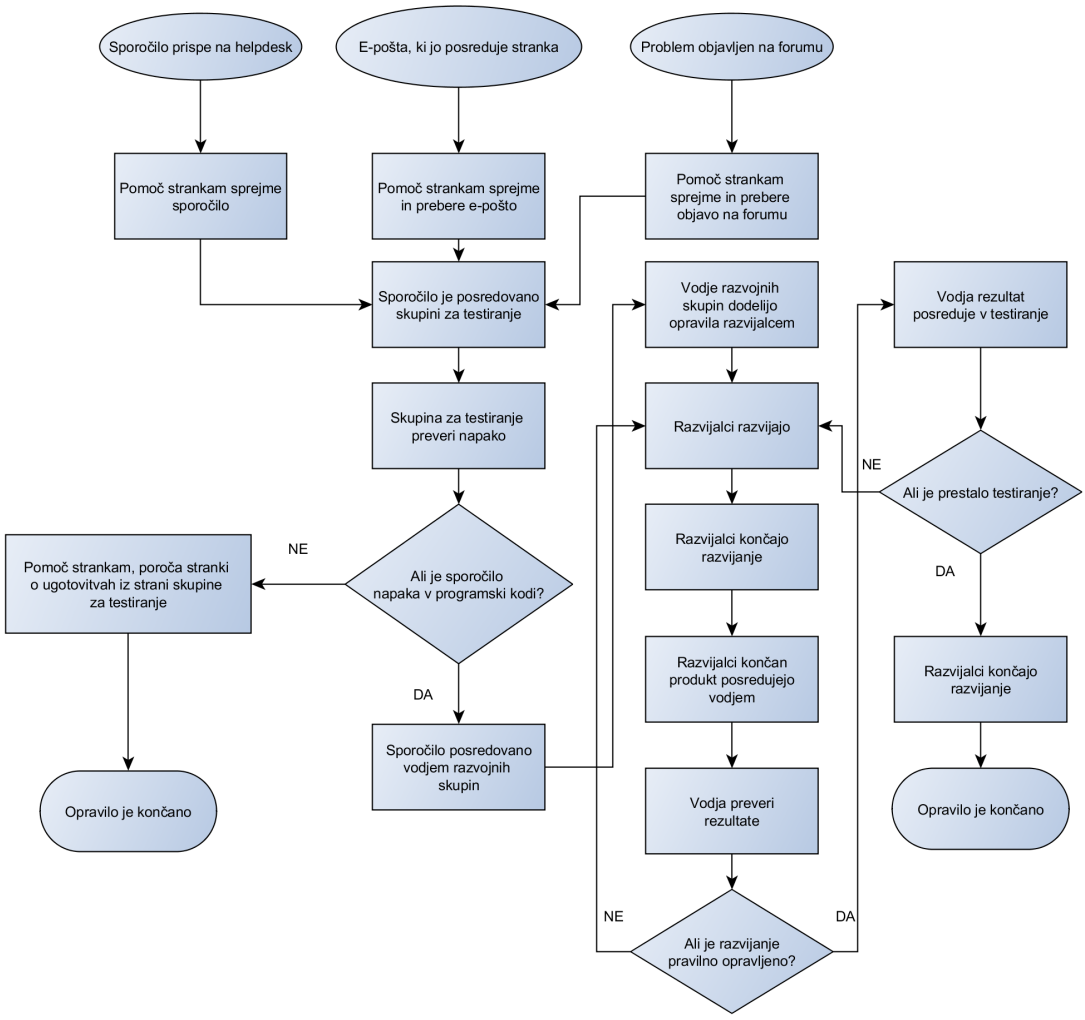
V prejšnjih podpoglavjih smo odkrili slabosti, ki so se pojavile v podjetju. Ti podatki nam bodo pomagali pri oblikovanju dokumenta, ki bo predpisoval aktivnosti, orodja, pristope, discipline in vloge pri posameznem projektu. Ta dokument mora biti agilen. Agilnost bo pri dokumentu prisotna na način, da bodo nekateri pristopi oziroma discipline omogočale prilagodljivost glede na posamično aktivnost. En primer agilnosti je število sodelujočih pri posamičnem projektu. Število sode-

lujočih pri projektu je odvisna od več spremenljivk. Ni nujno določeno na začetku in takšno število ne ostane skozi celotni projekt, ampak se lahko njihovo število skozi projekt povečuje ali zmanjšuje – odvisno od posameznih aktivnosti, ki jih je treba rešiti. V nadaljevanju bo predstavljen dokument, ki bo predpisoval aktivnostim orodja, pristope, discipline in vloge za projekte z vključevanjem agilnosti.

Dokument bo razdeljen na več delov. Najprej bo predstavljen glavni proces, ki prikazuje proces sprejemanja strankinega sporočila in njegovo obdelovanje pri razvijalcih..

3.4.7.1 Dokumentacija glavnega procesa

Glavni proces je prikazan na sliki 3.5. Če pogledamo sliko, vidimo, da pomoč strankam pridobi sporočila iz več virov. Ti viri so sporočila prek foruma ali e-pošte ali prek drugih virov. Skupina za pomoč strankam pridobi sporočilo: To se analizira in ugotovi, ali je namen sporočila napaka v programski kodi ali je sporočilo le vprašanje. Če je ugotovljena napaka v programski kodi, je sporočilo posredovano skupini za testiranje. Namen te skupine je testiranje programske opreme in ugotavljanje napak v programski kodi. Zato ugotovljeno napako testirajo in ugotovijo, ali je to dejanska napaka v programu ali dodatek (Angl. “Feature”) k programu. Če je napaka, se to sporočilo z dodatnimi podatki posreduje voditeljem razvojnih skupin. Naloga voditeljev razvojnih skupin je organizacija dela med razvijalci, ki pripadajo njegovi skupini. Ko je razvijalcu dodeljeno delo, začne razvijanje. Ko razvijalec konča razvijanje, je popravljen izdelek posredovan voditelju razvijanja. Ta bo ocenil kakovost rešitve (Angl. “Code review”). Če je pozitivno ocenjen, voditelj posreduje izdelek v testiranje. Če je negativno ocenjen, mora razvijalec nadaljevati razvijanje. Skupina za testiranje z novonapisanimi testi, testira novo programsko opremo. Če pozitivno prestane testiranje, je razvijalec končal razvijanje. Zaključek opravila je sporočen stranki. Če se testiranje ne konča uspešno, pa mora razvijalec nadaljevati razvijanje in odpravljanje napak.



Slika 3.5: Glavni proces razvoja programske opreme

3.4.7.2 Preslikava vlog iz metodologij v vloge iz podjetja

V poglavjih od 3.4.4 do 3.4.6 smo predstavili metode, ki smo jih uporabili za izboljševanje stanja v procesu razvoja programske opreme. Vloge, ki so prisotne pri teh metodah se ne ujemajo z strukturo zaposlenih v podjetju. Zato je potrebna preslikava vlog. V tabeli 3.1 imamo preslikavo vlog iz metodologij Kanban in Scrum in ekstremno programiranje v vloge, ki so prisotne v podjetju.

Vloge iz Scruma/Kanbana	Vloge v podjetju
Skrbnik izdelka(Scrum)/Vodja Projekta(Kanban)	Višji razvojni inženir za programsko opremo
	Višji aplikacijski inženir
Skrbnik procesa(Scrum)	Vodilni razvojni inženir za organizacijo na področju programske opreme
Vodje skupin(Kanban)	Vodilni razvojni inženir za programsko opremo
	Vodja aplikacijske skupine
	Vodja podpore uporabnikom
	Višji inženir za testiranje programske opreme
Skupina razvijalcev	Razvojni inženir za programsko opremo I
	Razvojni inženir za programsko opremo II
	Aplikacijski inženir
	Inženir za podporo uporabnikom
	Višji inženir za podporo uporabnikom
	Inženir za testiranje programske opreme I
	Inženir za testiranje programske opreme II

Tabela 3.1: Preslikava vlog iz metodologij v vloge iz podjetja

3.4.7.3 Dokumentacija vlog

Vsaka posamezna vloga ima na začetku kratek opis. Opis vloge predstavlja namen posamezne vloge. Opisu vloge bodo sledila priporočena znanja. Z definiranjem priporočenega znanja predstavimo, kakšno znanje morajo imeti zaposleni. To pomeni, da morajo biti znanje in kompetence na takšnem nivoju, da realno predstavlja neko vlogo. Če so znanja na višjem ali nižjem nivoju, kot so realno potrebna, lahko pripelje do napačnih zaključkov in tako posledično do napačnih zahtev, do posameznikov. Priporočena znanja nam prav tako pridejo priročna pri iskanju zaposlitve novega kadra, saj imamo definirana znanja, ki si jih želimo pri novih delavcih. V nadaljevanju bodo predstavljene vloge, ki so v podjetju. Na začetku bodo vloge, ki so povezane s testiranjem programske opreme. Nato bodo sledile

vloge, ki povezane s podporo uporabnikom. Potem sledijo pa vloge aplikacijskega inženirja in razvojnega inženirja za programsko opremo.

3.4.7.4 Inženir za testiranje programske opreme I

Kratek opis vloge

Inženir za testiranje programske opreme je oseba, ki se ukvarja s testiranjem programske opreme.

Priporočena znanja

- Pisanje testov.
- Testiranje izdelkov.
- Iskanje in odpravljanje napak.
- Pripravljanje novih različic programskih paketov.
- Pripravljanje, vodenje in arhiviranje tehnične. dokumentacije

3.4.7.5 Inženir za testiranje programske opreme II

Kratek opis vloge

Inženir za testiranje programske opreme II je napredovanje od inženirja za testiranje programske opreme. Dodeljene so mu dodatne obveznosti.

Priporočena znanja

- Pisanje testnih skript in programov.
- Testiranje izdelkov.
- Iskanje in odpravljanje napak.
- Pripravljanje novih različic programskih paketov.
- Pripravljanje, vodenje in arhiviranje tehnične dokumentacije.

3.4.7.6 Višji inženir za testiranje programske opreme

Kratek opis vloge

Višji inženir za testiranje programske opreme je vodja skupine, ki je odgovorna za testiranje programske opreme. Dodeljene so mu dodatne obveznosti.

Priporočena znanja

- Upravljanje s testno infrastrukturo.
- Pisanje testnih skript in programov.
- Testiranje izdelkov.
- Iskanje in odpravljanje napak.
- Pripravljanje novih različic programskih paketov.
- Pripravljanje, vodenje in arhiviranje tehnične dokumentacije,

3.4.7.7 Inženir za podporo uporabnikom

Kratek opis vloge

Inženir za podporo uporabnikom je odgovoren za komunikacijo z uporabniki in nudenje pomoči uporabnikom.

Priporočena znanja

- Nudenje tehnične pomoči uporabnikom.
- Posredovanje informacij ključnim deležnikom.
- Diagnosticiranje napak.
- Vodenje evidenc in statistik.

3.4.7.8 Višji inženir za podporo uporabnikom

Kratek opis vloge

Višji inženir za podporo je napredovanje inženirja za podporo uporabnikom. Dodeljene so mu dodatne obveznosti.

Priporočena znanja

- Nudenje tehnične pomoči uporabnikom.
- Posredovanje informacij ključnim deležnikom.
- Diagnosticiranje napak.
- Vodenje evidenc in statistik.

3.4.7.9 Vodja podpore uporabnikom

Kratek opis vloge

Vodja podpore uporabnikom je vodja skupine, ki se ukvarja s podporo uporabnikom. Dodeljene so mu dodatne obveznosti kot višjemu inženirju za podporo uporabnikom.

Priporočena znanja

- Načrtovanje, organiziranje, usklajevanje in nadziranje dela.
- Nudenje tehnične pomoči uporabnikom.
- Posredovanje informacij ključnim deležnikom.
- Diagnosticiranje napak.
- Vodenje evidenc, statistik, izdelovanje analiz in pripravljanje poročil.

3.4.7.10 Aplikacijski inženir

Kratek opis vloge

Naloga aplikacijskega inženirja je povezovanje teorije in prakse. Strankam svetuje, kateri izdelki so primerni glede na njihove potrebe.

Priporočena znanja

- Svetovanje kupcem pri izbiri optimalnih tehničnih rešitev.
- Celovito implementiranje rešitev.
- Izobraževanje in usposabljanje kupcev/uporabnikov.
- Seznanjanje kupcev z novosti.
- Sodelovanje pri razvoju izdelkov in rešitev.
- Proaktivno promoviranje družbe.
- Vodenje evidenc, izdelovanje analiz in priprava poročil.

3.4.7.11 Višji aplikacijski inženir

Kratek opis vloge

Višji aplikacijski inženir je napredovanje aplikacijskega inženirja. Dodeljene so mu dodatne obveznosti.

Priporočena znanja

- Zbiranje informacij in analiziranje podatkov s trga in konkurence.
- Prepoznavanje tržnih priložnosti.
- Svetovanje kupcem pri izbiri optimalnih tehničnih rešitev.
- Pripravljanje strategije pridobivanja kupcev, pripravljanje tehničnih predlogov in načrtovanje povpraševanja.

- Seznanjanje kupcev z novostmi.
- Izobraževanje in usposabljanje kupcev/uporabnikov.
- Pripravljanje izobraževalnih vsebin.
- Pripravljanje uporabniških priročnikov.
- Sodelovanje pri razvoju izdelkov in rešitev.
- Proaktivno promoviranje družbe.
- Posredovanje zahtevkov in specifikacij.
- Celovito implementiranje rešitev.

3.4.7.12 Vodja aplikacijske skupine

Kratek opis vloge

Vodja aplikacijske skupine je vodja aplikacijskih inženirjev in se ukvarja s predstavitvijo in prodajo izdelkov ter programske opreme strankam.

Priporočena znanja

- Načrtovanje, organiziranje, usklajevanje in nadziranje dela.
- Zbiranje informacij in analiziranje podatkov s tržišča in konkurence.
- Prepoznavanje tržnih priložnosti.
- Svetovanje kupcem pri izbiri optimalnih tehničnih rešitev.
- Pripravljanje strategije pridobivanja kupcev, pripravljanje tehničnih predlogov in načrtovanje povpraševanja.
- Seznanjanje kupcev z novostmi.
- Izobraževanje in usposabljanje kupcev/uporabnikov.

- Pripravljanje izobraževalnih vsebin.
- Pripravljanje uporabniških priročnikov.
- Sodelovanje pri razvoju izdelkov in rešitev.
- Proaktivno promoviranje družbe.
- Posredovanje zahtevkov in specifikacij.
- Celovito implementiranje rešitev.
- Vodenje evidenc, izdelovanje analiz in priprava poročil.

3.4.7.13 Razvojni inženir za programsko opremo I

Kratek opis vloge

Razvojni inženir za programsko opremo I je odgovoren za razvijanje programskih rešitev.

Priporočena znanja

- Izdelovanje programske kode po specifikacijah.
- Pisanje testov.
- Iskanje in odpravljanje napak.
- Pripravljanje, vodenje in arhiviranje tehnične dokumentacije.

3.4.7.14 Razvojni inženir za programsko opremo II

Kratek opis vloge

Razvojni inženir za programsko opremo II je napredovanje od razvojnega inženirja za programsko opremo I in je odgovoren za razvijanje programskih rešitev ter dodatnih obveznosti.

Priporočena znanja

- Izdelovanje programske kode po uporabniških specifikacijah.
- Pregledovanje in potrjevanje programske kode.
- Pisanje testov.
- Iskanje in odpravljanje napak.
- Pripravljanje, vodenje in arhiviranje tehnične dokumentacije.
- Sodelovanje pri implementaciji rešitev.
- Pripravljanje prototipnih rešitev.
- Nudenje tehnične pomoči uporabnikom.
- Pripravljanje, vodenje in arhiviranje tehnične dokumentacije.

3.4.7.15 Višji razvojni inženir za programsko opremo

Kratek opis vloge

Višji razvojni inženir za programsko opremo je vodja skupin, ki so sestavljeni iz razvojnih inženirjev za programsko opremo II in razvojnih inženirjev za programsko opremo I. Dodeljene so mu dodatne obveznosti.

Priporočena znanja

- Operativno organiziranje, usklajevanje in kontroliranje dela.
- Izdelovanje zaključenih programskih modulov.
- Izdelovanje programske kode po uporabniških specifikacijah.
- Pripravljanje prototipnih rešitev.
- Pregledovanje in potrjevanje programske kode.
- Pisanje testov.

- Sodelovanje pri implementaciji rešitev.
- Iskanje in odpravljanje napak.
- Pripravljanje, vodenje in arhiviranje tehnične dokumentacije.
- Opravljanje drugih del v okviru strokovne usposobljenosti.

3.4.7.16 Vodilni razvojni inženir za programsko opremo

Kratek opis vloge

Vodilni razvojni inženir za programsko opremo je vodja nad višjimi razvojnimi inženirji za programsko opremo in ima vpogled nad dejavnostmi vseh razvojnih inženirjev.

Priporočena znanja

- Načrtovanje, organiziranje, usklajevanje in nadziranje dela.
- Dizajniranje in oblikovanje arhitekture razvoja informacijskih rešitev.
- Izdelovanje zaključenih programskih celot.
- Uvajanje novih standardov, orodij in tehnologij.
- Pripravljanje prototipnih rešitev.
- Pregledovanje in potrjevanje programske kode.
- Pisanje testov.
- Sodelovanje pri implementaciji rešitev.
- Iskanje in odpravljanje napak.
- Pripravljanje, vodenje in arhiviranje tehnične dokumentacije.
- Izdelovanje programske kode po uporabniških specifikacijah.
- Vodenje izobraževanj in usposabljanj.

3.4.7.17 Vodilni razvojni inženir za tehnološki razvoj na področju programske opreme

Kratek opis vloge

Vodilni razvojni inženir za tehnološki razvoj na področju programske opreme je odgovoren za vse zadeve glede razvoja v razvojnih skupinah.

Priporočena znanja

- Načrtovanje, organiziranje, usklajevanje in nadziranje dela.
- Usklajevanje zahtev strank in usmerjanje razvojne ekipe.
- Nadzorovanja in koordiniranje izvajanja nalog razvojnih skupin.
- Uvajanje novih standardov, orodij in tehnologij.
- Sodelovanje pri implementaciji rešitev.
- Vodenje izobraževanj in usposabljanj.
- Pripravljanje, vodenje in arhiviranje tehnične dokumentacije.

3.4.7.18 Vodilni razvojni inženir za organizacijo na področju programske opreme

Kratek opis vloge

Vodilni razvojni inženir za tehnološki razvoj na področju programske opreme je odgovoren za vse zadeve glede organizacije v razvojnih skupinah.

Priporočena znanja

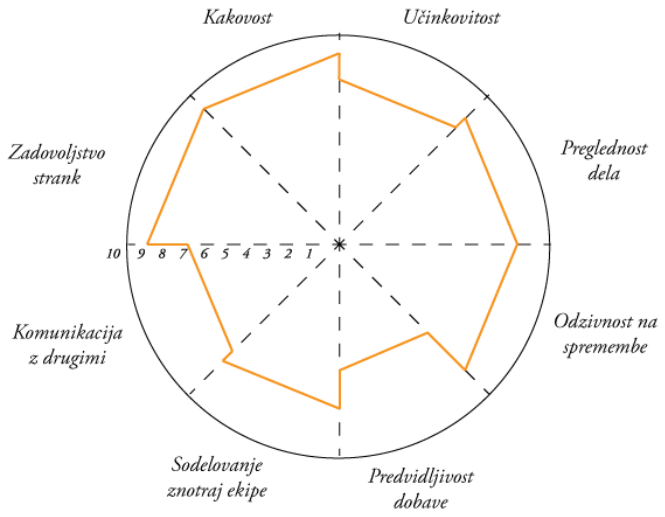
- Načrtovanje, organiziranje, usklajevanje in nadziranje dela.
- Pripravljanje programov mentorstev, njihovo izvajanje in uvajanje novih sodelavcev.
- Načrtovanje kariernega razvoja sodelavcev.

- Optimiziranje obstoječih in uvajanje novih procesov.
- Uvajanje novih standardov, orodij in tehnologij.
- Razvijanje in uvajanje novih postopkov, tehnik in načinov dela.
- Vodenje izobraževanj in usposabljanj.
- Pripravljanje, vodenje in arhiviranje tehnične dokumentacije.

3.4.8 Povratna informacija iz podjetja

V prejšnjih razdelkih smo predstavili dokument, s katerim smo definirali aktivnosti, orodja, pristope, discipline in vloge. Dokument je nastala na osnovi želja, potreb in lastnosti podjetja. Razvijalci so dokument preučili in dali povratno informacijo. Ta je pomembna saj nam da vpogled v to, kje je še potrebna izboljšava, da se bo metodologija še izboljšala in bo posledično vsebovala primernejše metode. Povratna informacija je bila pridobljena s pomočjo vprašalnika. Vprašalnik je bil posredovan zaposlenim tri mesece po uvedbi sprememb s pomočjo SME. Ta je vseboval vprašanja z mersko lestvico (Likertova lestvica) od 0 do 10. Vprašanja zajemajo vse dimenzije kolesa ravnovesja, ki smo ga predstavili v razdelku 3.2. Poleg vprašanj z mersko lestvico je vprašalnik vseboval še eno dodatno vprašanje. Navezovalo se je na najnižjo dimenzijo kolesa ravnovesja. Za to dimenzijo so morali podati rešitev, kako bi nivo dimenzije izboljšali. Vprašalnik so izpolnili vsi, ki so bili vključeni pri oblikovanju dokumenta in nove metodologije. To so vodja razvijalcev, menedžerji in razvijalci. Z vprašalnikom smo dobili celoviti vpogled v smiselnost uporabe predlagane metodologije. Da bi dobili grafično predstavitev povratne informacije, smo pridobljene odgovore dali na skupno kolo ravnovesja. Rezultat, ki ga prikazuje kolo ravnovestja, je na sliki 3.6. Na njej vidimo, da je prišlo do pozitivnega napredka v vseh merilih. Najnižje merilo je učinkovitost. Ker je bilo merilo učinkovitosti najnižja ocenjena dimenzija, je bilo treba pridobiti informacije o tem, kako bi izboljšali učinkovitost. Najpogostejši razlog je bil, da bi razvijalci pisali več testov in posledično sami testirali programsko kodo. S tem bi dosegli boljše stanje obstoječe in tudi nove kode. S tem bi lažje in hitreje zaznali

napake v kodi. Z manj odkritimi napakami pa bi prišlo do večjega zadovoljstva strank in prav tako boljšega zadovoljstva razvijalcev. Vprašalnik so izpolnili vsi, ki so bili vključeni pri oblikovanju dokumenta in nove metodologije. To so vodja razvijalcev, menedžer in razvijalci.



Slika 3.6: Zadnje stanje

Poglavje 4

Zaključek

V magistrskem delu smo preučevali situacijski pristop oblikovanja metodologije in agilne metodologije. V ospredje so bila postavljena podjetja, ki se ukvarjajo z izdelavo programskih rešitev. Podjetje, v katero je bila implementirana metodologija, se je spoprijemalo z velikimi problemi pri organizaciji procesa razvoja programske opreme. Ti so povzročili zakasnjeni razvoj programske opreme. To je privedlo do nezadovoljstva zaposlenih in posledično tudi strank. Da bi podjetje ostalo konkurenčno, je bila potrebna sprememba v podjetju. Informacij o tem, kako implementirati agilne metodologije v podjetje, pri katerih upoštevamo že obstoječe lastnosti, ni veliko. To pomeni, da je največji fokus na implementaciji nove metodologije v podjetje.

Pred implementacijo metodologije je potrebna teoretična analiza. Teoretično smo analizirali situacijski pristop oblikovanja metodologije, ki velja za primeren pristop za oblikovanje metodologije za podjetje. Po teoretični analizi situacijskega pristopa smo analizirali tradicionalne in agilne metodologije. Analiza metodologij je bila potrebna, saj nam je podala informacije o metodah, s katerimi lahko oblikujemo novo metodologijo. Pridobljene metode si lahko predstavljamo kot posamezne dele sestavljanke, ki pa skupaj sestavijo sestavljanko, ki predstavlja novo metodologijo. Posamezen del sestavljanke lahko zamenjamo z metodo iz druge metodologije, če je primernejša za podjetje. Tako naredimo metodologijo primernejšo za posame-

zno podjetje.

Po teoretični analizi je bila potrebna analiza stanja podjetja. Ta informacija nam je pomagala poiskati slabosti, ki se dogajajo v podjetju. Za pridobitev informacij o podjetju je bila uporabljena metoda intervjuja in vprašalnikov. Metoda intervjuja je bila opravljena z vodjo razvijalcev in vodjo menedžerjev. Z metodo vprašalnika pa smo pridobili informacije pri razvijalcih. S pridobljenimi informacijami smo izpostavili slabosti podjetja in navedli, s katerimi metodami iz agilnih metodologij bi jih popravili ter na kak način. V nadaljevanju smo se odločili, katere metode bomo uporabili glede na lastnosti podjetja in zaposlenih.

Nastala metodologija je bila predstavljena podjetju. Magistrsko delo nudi informacije o tem, kako so bile pridobljene informacije o podjetju, katere metode so bile izbrane in implementirane v podjetje. Za povratno informacijo smo v podjetju izvedli anketo, v kateri so zaposleni odgovorili na vprašanja glede metodologije. S tem smo pridobili informacijo o tem, kakšno je njihovo mnenje o nastali metodologiji.

Predvideni cilji, ki so bili postavljeni, so bili doseženi. To pomeni, da je bila na začetku opravljena analiza podjetja, s katero smo pridobili informacijo o tem, v kakšnem stanju je podjetje in kje v procesu razvoja programske opreme so bili prisotni problemi. S to informacijo in pridobljenim znanjem o metodologijah je bila oblikovana nova metodologija. Novonastala metodologija je bila oblikovana glede na situacije, ki se dogajajo v podjetju. S tem magistrskim delom bodo lahko druga podjetja, ki se spoprijemajo s podobnimi problemi implementacije metodologije, dobile vpogled v postopek, kako je bila implementirana agilna metodologija v določenem podjetju. S tem bodo podjetja pridobila pregled nad postopkom implementacije. Ta bo pomagal pri lažji implementaciji podobne metodologije in pravočasnem odpravljanju težav.

Metodologija je živa. Zaradi tega se spreminja skozi čas. To pomeni, da lahko

skozi čas pričakujemo, da bo prišlo do sprememb. Magistrsko delo je definiralo metode za odpravo napak, ki so se pojavile med analizo. Za izboljšavo bi bilo treba analizirati spremembe metodologije skozi daljši čas. S tem vidikom tudi dobimo dragocen pregled nad življenjsko dobo metodologije. Zato je potrebna longitudinalna analiza za daljše obdobje. V prihodnje lahko analiziramo, kako se je metodologija spreminjala in kateri elementi v podjetju so vplivali na njene spremembe.

Literatura

[1] D. Aveson, G. Fitzgerald, Methodologies for developing information systems: A historical perspective (2006).

[2] A. Weitzer, Primerjava in vrednotenje procesov razvoja programske opreme, Master's thesis, Fakulteta za računalništvo in informatiko (February 2010). URL http://eprints.fri.uni-lj.si/1017/

[3] Agile manifest, Dosegljivo: http://agilemanifesto.org/, [Dostopano: 25. 7. 2018] (2001).

[4] Henderson-Sellers, B. and Ralyté, J. and Ågerfalk, P.J. and Rossi, M., Situational Method Engineering, Springer Berlin, 2016.

[5] K. Conboy, S. Coyle, X. Wang, M. Pikkarainen, People over Process: Key Challenges in Agile Development, IEEE Software 28 (4) (2011) 48–57. doi:10.1109/ms.2010.132.

[6] Fran, Dosegljivo: https://tinyurl.com/y9wx2w79, [Dostopano: 10. 9. 2018].

[7] iSlovar, Dosegljivo: http://www.islovar.org/islovar, [Dostopano: 10. 9. 2018].

[8] N. Jayaratna, Understanding and Evaluating Methodologies: NIMSAD, a Systematic Framework, McGraw-Hill, Inc., New York, NY, USA, 1994.

[9] Fran, Dosegljivo: https://tinyurl.com/ycbwm7jk, [Dostopano: 10. 9. 2018].

[10] I. van de Weerd, S. Brinkkemper, J. Souer, J. Versendaal, A situational implementation method for web-based content management system-applications: method engineering and validation in practice, Software Process: Improvement and Practice 11 (5) (2006) 521–538.

[11] C. Gonzalez-Perez, Supporting situational method engineering with ISO/IEC 24744 and the work product pool , in: Situational Method Engineering: Fundamentals and Experiences, Springer, 2007, pp. 7–18.

[12] R. Steenweg, M. Kuhrmann, D. Méndez Fernández, Software Engineering Process Metamodels - A Literature Review (01 2014).

[13] D. G. Firesmith, B. Henderson-Sellers, The OPEN process framework: an introduction, Addison-Wesley, 2002.

[14] OMG, About the Software & Systems Process Engineering Metamodel Specification Version 2.0, Dosegljivo: "https://www.omg.org/spec/SPEM/2.0/", [Dostopano: 25. 8. 2018] (Apr. 2008).

[15] ISO/IEC, ISO/IEC 24744:2014, Dosegljivo: https://www.iso.org/standard/62644.html, [Dostopano: 24. 7. 2018] (2014).

[16] C. Rolland, C. Souveyet, M. Moreno, An approach for defining ways-of-working, Information Systems 20 (4) (1995) 337–359. doi:10.1016/0306-4379(95)00018-y.

[17] A. Röstlinger, G. Goldkuhl, Generisk flexibilitet: På väg mot en komponentbaserad metodsyn, Institutionen för datavetenskap, Universitetet och tekniska högskolan Linköping, 1996.

[18] G. Giray, B. Tekinerdogan, Situational Method Engineering for Constructing Internet of Things Development Methods, Lecture Notes in Business Information Processing Business Modeling and Software Design (2018) 221–239doi:10.1007/978-3-319-94214-8_14.

[19] J. Ralyté, R. Deneckère, C. Rolland, Towards a generic model for situational method engineering, in: International Conference on Advanced Information Systems Engineering, Springer, 2003, pp. 95–110.

[20] M. Cervera, M. Albert, V. Torres, V. Pelechano, Turning Method Engineering Support into Reality, IFIP Advances in Information and Communication Technology Engineering Methods in the Service-Oriented Context (2011) 138–152doi:10.1007/978-3-642-19997-4_14.

[21] P. Kruchten, The rational unified process: an introduction, Addison-Wesley, 2003.

[22] A. Stellman, J. Greene, Learning Agile, OReilly, 2016.

[23] R. C. Martin, Agile software development: principles, patterns, and practices, Pearson Education, 2003.

[24] K. Beck, C. Andres, Extreme programming explained: Second edition, embrace change, Addison-Wesley, 2015.

[25] E. Brechner, Agile project management with Kanban, Microsoft, 2015.

[26] L. B. S. Raccoon, The chaos model and the chaos cycle, ACM SIGSOFT Software Engineering Notes 20 (1) (1995) 55–66. doi:10.1145/225907.225914.

[27] P. N. Robillard, P. DAstous, P. Kruchten, Software engineering process with the UPEDU, Addison Wesley, 2003.

[28] P. Elwer, Agile Project Development at Intel: A Scrum Odyssey, Danube Case Study: Intel Corporation (2008) 1–14.

[29] P. Green, Adobe Premiere Pro Scrum Adoption: How an Agile Approach Enabled Success in a Hyper-competitive Landscape, 2012 Agile Conferencedoi:10.1109/agile.2012.28.

[30] J. V. Sutherland, J. J. Sutherland, Scrum: the art of doing twice the work in half the time, Currency, 2014.

[31] M. Stephens, D. Rosenberg, Extreme programming refactored: the case against XP, Apress, 2003.

[32] D. J. Anderson, Kanban: successful evolutionary change for your technology business, Blue Hole Press, 2010.

[33] N. Walliman, Social research methods: the essentials, SAGE, 2016.

[34] SAFe, Agile Metrics, Dosegljivo: `https://www.scaledagileframework.com/metrics/`, [Dostopano: 26. 7. 2018] (2018).

[35] A. Page, K. Johnston, B. Rollison, L. Finnel, How we test software at Microsoft, Microsoft Press, 2009.

[36] G. Gruver, T. Mouser, Leading the transformation: applying Agile and DevOps principles at scale, IT Revolution, 2015.